Исчерпывающее руководство

Профессиональное программирование

Scala

4-е издание

Scala 2.13 — впервые на русском языке

Мартин Одерски
Лекс Спун
Билл Веннерс

ПИТЕР®
artima

Programming in Scala

Fourth Edition

Martin Odersky, Lex Spoon, Bill Venners

artima
Artima Press
Walnut Creek, California

Профессиональное программирование

Scala

4-е издание

Мартин Одерски
Лекс Спун
Билл Веннерс

ПИТЕР®

Санкт-Петербург • Москва • Минск

2021

ББК 32.973.2-018.1
УДК 004.43
О-41

Одерски М., Спун Л., Веннерс Б.

О-41 Scala. Профессиональное программирование. 4-е изд. — СПб.: Питер, 2021. — 720 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1827-4

«Scala. Профессиональное программирование» — главная книга по Scala, популярному языку для платформы Java, в котором сочетаются концепции объектно-ориентированного и функционального программирования, благодаря чему он превращается в уникальное и мощное средство разработки.

Этот авторитетный труд, написанный создателями Scala, поможет вам пошагово изучить язык и идеи, лежащие в его основе.

Данное четвертое издание полностью обновлено. Добавлен материал об изменениях, появившихся в Scala 2.13, в том числе:

- новая иерархия типов коллекций;
- новые конкретные типы коллекций;
- новые методы, добавленные к коллекциям;
- новые способы определять собственные типы коллекций;
- новые упрощенные представления.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-0981531618 англ.
ISBN 978-5-4461-1827-4

Краткое содержание

Оглавление

Предисловие

Когда в 2004 году, на заре своей карьеры, я сел и выбрал малоизвестный язык под названием Scala, мне и в голову не могло прийти, что впереди меня ждет множество открытий. Поначалу мой опыт использования Scala во многом напоминал работу с другими языками: пробы, ошибки, эксперименты и открытия, заблуждения и наконец просветление. В те дни у нас было очень мало источников информации: никаких учебных пособий, блогов или опытных пользователей, желавших поделиться знаниями. А уж таких ресурсов, как эта книга, у нас точно не было. Все, что мы имели, — язык с прекрасным набором новых инструментов, которыми еще никто не умел как следует пользоваться. Это открывало новые горизонты и в то же время приводило в замешательство! С опытом программирования на Java у меня сформировались определенные ожидания, однако мои *ощущения* от повседневного написания кода на Scala были другими. Я помню свои первые приключения со Scala, когда мы вместе еще с одним человеком работали над небольшим проектом. Решив заняться рефакторингом (ввиду регулярного обнаружения и изучения новых возможностей и методик это случалось довольно часто), я проходил несколько этапов компиляции.

Каждый раз компилятор выдавал мне список ошибок вместе с номерами строк. И всякий раз я переходил к нужному участку кода, пытался понять, что с ним не так, вносил изменения, чтобы исправить ошибки (или, скорее, переместить их в другое место). Но компилятор раз за разом направлял меня прямо к источнику проблемы. Иногда это могло повторяться на протяжении нескольких дней без единой успешной компиляции. Но, постепенно уменьшив количество ошибок с сотни до десяти, а затем и до нуля, я наконец впервые смог скомпилировать и запустить существенно переработанный проект.

И, вопреки ожиданиям, он заработал с первого раза. Я был молодым программистом, который до этого писал только на Java, Perl, Pascal, BASIC, PHP и JavaScript, и это произвело на меня неизгладимое впечатление.

На первой конференции Scala World, которую я организовал в 2015 году, во вступительной презентации Рунара Бьярнасона (Rúnar Bjarnason) была озвучена следующая мысль: «Ограничения освобождают, а свободы сковывают». Нагляднее всего это демонстрирует процесс компиляции в Scala: понимание того, что `scalac` накладывает исчерпывающий набор ограничений, не позволяющих программисту допускать предсказуемые ошибки времени выполнения (хуже которых ничего нет), должно давать ощущение свободы любому программисту. Оно должно придавать уверенность, а это, в свою очередь, позволяет экспериментировать,

исследовать и вносить амбициозные изменения даже при отсутствии полноценного набора тестов.

С тех пор я продолжаю изучать Scala и даже сегодня открываю для себя новые возможности, нюансы и интересные сочетания разных функций. Я не знаю ни одного другого языка, способного увлекать своей глубиной на протяжении стольких лет.

Scala стоит на пороге больших изменений. Следующая версия, 3, будет таким же большим шагом вперед по сравнению со Scala 2, как тот, который я сделал 15 лет назад, когда перешел с Java. Повседневное программирование на Scala во многом останется прежним, но при этом мы получим доступ к широкому спектру новых возможностей, охватывающих все аспекты данного языка.

На момент написания этих строк широкое внедрение Scala 3 ожидается лишь через несколько лет, и в ближайшем будущем Scala 2 останется фактически стандартной версией языка.

В последнем выпуске Scala 2 (версия 2.13), подробно рассматриваемом в данной книге, в стандартной библиотеке появились новые, переработанные и упрощенные коллекции, вобравшие в себя множество уроков, усвоенных с момента последнего крупного изменения (Scala 2.8). Эти новые коллекции поддерживают компиляцию в Scala 2 и 3, формируя основу для большей части кода, который мы будем писать еще и в грядущем десятилетии. Учитывая то, какой интерес у людей вызывает следующий выпуск Scala, самое время обзавестись этой книгой и начать ее изучать!

Джон Претти (Jon Pretty), Краков, Польша,
14 сентября 2019 года

Благодарности

Мы благодарны за вклад в эту книгу многим людям.

Сам язык Scala — плод усилий множества специалистов. Свой вклад в проектирование и реализацию версии 1.0 внесли Филипп Альтер (Philippe Altherr), Винсент Кремет (Vincent Cremet), Жиль Дюбоше (Gilles Dubochet), Бурак Эмир (Burak Emir), Стефан Мишель (Stéphane Micheloud), Николай Михайлов (Nikolay Mihaylov), Мишель Шинц (Michel Schinz), Эрик Стенман (Erik Stenman) и Матиас Зенгер (Matthias Zenger). К разработке второй и текущей версий языка, а также инструментальных средств подключились Фил Багвел (Phil Bagwell), Антонио Куней (Antonio Cunei), Юлиан Драгос (Iulian Dragos), Жиль Дюбоше (Gilles Dubochet), Мигель Гарсиа (Miguel Garcia), Филипп Халлер (Philipp Haller), Шон Макдирмид (Sean McDirmid), Инго Майер (Ingo Maier), Донна Малайери (Donna Malayeri), Адриан Мурс (Adriaan Moors), Хуберт Плоциничак (Hubert Plociniczak), Пол Филлипс (Paul Phillips), Александр Прокопец (Aleksandar Prokopec), Тиарк Ромпф (Tiark Rompf), Лукас Рыц (Lukas Rytz) и Джеффри Уошберн (Geoffrey Washburn).

Следует также упомянуть тех, кто участвовал в работе над структурой языка. Эти люди любезно делились с нами своими идеями в оживленных и вдохновляющих дискуссиях, вносили важные фрагменты кода в открытую разработку и делали весьма ценные замечания по поводу предыдущих версий. Это Гилад Браха (Gilad Bracha), Натан Бронсон (Nathan Bronson), Коаюан (Caoyuan), Эймон Кэннон (Aemon Cannon), Крейг Чамберс (Craig Chambers), Крис Конрад (Chris Conrad), Эрик Эрнст (Erik Ernst), Матиас Феллизен (Matthias Felleisen), Марк Харра (Mark Harrah), Шрирам Кришнамурти (Shriram Krishnamurti), Гэри Ливенс (Gary Leavens), Дэвид Макивер (David MacIver), Себастьян Манит (Sebastian Maneth), Рикард Нильссон (Rickard Nilsson), Эрик Мейер (Erik Meijer), Лалит Пант (Lalit Pant), Дэвид Поллак (David Pollak), Джон Претти (Jon Pretty), Клаус Остерман (Klaus Ostermann), Хорхе Ортис (Jorge Ortiz), Дидье Реми (Didier Rémy), Майлз Сабин (Miles Sabin), Виджей Сарасват (Vijay Saraswat), Даниэль Спивак (Daniel Spiewak), Джеймс Страчан (James Strachan), Дон Симе (Don Syme), Эрик Торребор (Erik Torreborre), Мэдс Торгерсен (Mads Torgersen), Филип Уодлер (Philip Wadler), Джейми Уэбб (Jamie Webb), Джон Уильямс (John Williams), Кевин Райт (Kevin Wright) и Джейсон Зауг (Jason Zaugg). Очень полезные отзывы, которые помогли нам улучшить язык и его инструментальные средства, были получены от людей, подписанных на наши рассылки по Scala.

Джордж Бергер (George Berger) усердно работал над тем, чтобы процесс создания и размещения книги в Интернете протекал гладко. Как результат, в данном проекте не было никаких технических сбоев.

Ценные отзывы о начальных вариантах текста книги были получены нами от многих людей. Наших благодарностей заслуживают Эрик Армстронг (Eric Armstrong), Джордж Бергер (George Berger), Алекс Блевитт (Alex Blewitt), Гилад Браха (Gilad Bracha), Уильям Кук (William Cook), Брюс Экель (Bruce Eckel), Стефан Мишель (Stéphane Micheloud), Тод Мильштейн (Todd Millstein), Дэвид Поллак (David Pollak), Фрэнк Соммерс (Frank Sommers), Филип Уодлер (Philip Wadler) и Матиас Зенгер (Matthias Zenger). Спасибо также представителям Silicon Valley Patterns group за их весьма полезный обзор. Это Дейв Астелс (Dave Astels), Трейси Бялик (Tracy Bialik), Джон Брюер (John Brewer), Эндрю Чейз (Andrew Chase), Брэдфорд Кросс (Bradford Cross), Рауль Дюк (Raoul Duke), Джон П. Эйрих (John P. Eurich), Стивен Ганц (Steven Ganz), Фил Гудвин (Phil Goodwin), Ральф Йочем (Ralph Jocham), Ян-Фа Ли (Yan-Fa Li), Тао Ма (Tao Ma), Джеффри Миллер (Jeffery Miller), Суреш Пай (Suresh Pai), Русс Руфер (Russ Rufer), Дэйв У. Смит (Dave W. Smith), Скотт Торнквест (Scott Turnquest), Вальтер Ваннини (Walter Vannini), Дарлин Уоллах (Darlene Wallach) и Джонатан Эндрю Уолтер (Jonathan Andrew Wolter). Кроме того, хочется поблагодарить Дуэйна Джонсона (Dewayne Johnson) и Кима Лиди (Kim Leedy) за помощь в художественном оформлении обложки, а также Фрэнка Соммерса (Frank Sommers) — за работу над алфавитным указателем.

Хотелось бы выразить особую благодарность и всем нашим читателям, приславшим комментарии. Они нам очень пригодились для повышения качества книги. Мы не в состоянии опубликовать имена всех приславших комментарии, но объявим имена тех читателей, кто прислал не менее пяти комментариев на стадии eBook PrePrint™, воспользовавшись ссылкой Suggest. Отсортируем их имена по убыванию количества комментариев, а затем в алфавитном порядке. Наших благодарностей заслуживают Дэвид Бизак (David Biesack), Дон Стефан (Donn Stephan), Матс Хенриксон (Mats Henricson), Роб Диккенс (Rob Dickens), Блэр Захак (Blair Zajac), Тони Слоан (Tony Sloane), Найджел Харрисон (Nigel Harrison), Хавьер Диас Сото (Javier Diaz Soto), Уильям Хелан (William Heelan), Джастин Фурдер (Justin Forder), Грегор Перди (Gregor Purdy), Колин Перкинс (Colin Perkins), Бьярте С. Карлсен (Bjarte S. Karlsen), Эрвин Варга (Ervin Varga), Эрик Уиллигерс (Eric Willigers), Марк Хейс (Mark Hayes), Мартин Элвин (Martin Elwin), Калум Маклин (Calum MacLean), Джонатан Уолтер (Jonathan Wolter), Лес Прушински (Les Pruszynski), Сет Тисуе (Seth Tisue), Андрей Формига (Andrei Formiga), Дмитрий Григорьев (Dmitry Grigoriev), Джордж Бергер (George Berger), Говард Ловетт (Howard Lovatt), Джон П. Эйрих (John P. Eurich), Мариус Скуртеску (Marius Scurtescu), Джефф Эрвин (Jeff Ervin), Джейми Уэбб (Jamie Webb), Курт Зольман (Kurt Zoglmann), Дин Уэмплер (Dean Wampler), Николай Линдберг (Nikolaj Lindberg), Питер Маклейн (Peter McLain), Аркадиуш Стрыйски (Arkadiusz Stryjski), Шанки Сурана (Shanky Surana), Крейг Бордолон (Craig Bordelon), Александр Пэтри (Alexandre Patry), Филип Моэнс (Filip Moens), Фред Янон (Fred Janon), Джефф Хеон (Jeff Heon), Борис Лорбер (Boris Lorbeer), Джим Менар (Jim Menard), Тим Аццопарди (Tim Azzopardi), Томас Юнг (Thomas Jung), Уолтер Чанг (Walter Chang), Йерун Дийкмейер (Jeroen Dijkmeijer), Кейси Боу-

мен (Casey Bowman), Мартин Смит (Martin Smith), Ричард Даллауэй (Richard Dallaway), Антони Стаббс (Antony Stubbs), Ларс Вестергрен (Lars Westergren), Маартен Хазевинкель (Maarten Hazewinkel), Мэтт Рассел (Matt Russell), Ремигиус Михаловский (Remigiusz Michalowski), Андрей Толопко (Andrew Tolopko), Кертис Стэнфорд (Curtis Stanford), Джошуа Каш (Joshua Cough), Земен Денг (Zemian Deng), Кристофер Родригес Масиас (Christopher Rodrigues Macias), Хуан Мигель Гарсия Лопес (Juan Miguel Garcia Lopez), Мишель Шинц (Michel Schinz), Питер Мур (Peter Moore), Рэндолф Кале (Randolph Kahle), Владимир Кельман (Vladimir Kelman), Даниэль Гронау (Daniel Gronau), Дирк Детеринг (Dirk Detering), Хироаки Накамура (Hiroaki Nakamura), Оле Хогаард (Ole Hougaard), Бхаскар Маддала (Bhaskar Maddala), Дэвид Бернар (David Bernard), Дерек Махар (Derek Mahar), Джордж Коллиас (George Kollias), Кристиан Нордал (Kristian Nordal), Нормен Мюллер (Normen Mueller), Рафаэль Феррейра (Rafael Ferreira), Бинил Томас (Binil Thomas), Джон Нильсон (John Nilsson), Хорхе Ортис (Jorge Ortiz), Маркус Шульте (Marcus Schulte), Вадим Герасимов (Vadim Gerasimov), Камерон Таггарт (Cameron Taggart), Джон-Андерс Тейген (Jon-Anders Teigen), Сильвестр Забала (Silvestre Zabala), Уилл Маккуин (Will McQueen) и Сэм Оуэн (Sam Owen).

Хочется также сказать спасибо тем, кто отправил сообщения о замеченных неточностях после публикации первых двух изданий. Это Феликс Зигрист (Felix Siegrist), Лотар Мейер-Лербс (Lothar Meyer-Lerbs), Диетард Михаэлис (Diethard Michaelis), Рошан Даврани (Roshan Dawrani), Донн Стефан (Donn Stephan), Уильям Утер (William Uther), Франсиско Ревербель (Francisco Reverbel), Джим Балтер (Jim Balter), Фрик де Брюйн (Freek de Bruijn), Амброз Лэнг (Ambrose Laing), Сехар Прабхала (Sekhar Prabhala), Левон Салдамли (Levon Saldamli), Эндрю Бурсавич (Andrew Bursavich), Хьялмар Петерс (Hjalmar Peters), Томас Фер (Thomas Fehr), Ален О'Ди (Alain O'Dea), Роб Диккенс (Rob Dickens), Тим Тейлор (Tim Taylor), Кристиан Штернагель (Christian Sternagel), Мишель Паризьен (Michel Parisien), Джоэл Нили (Joel Neely), Брайан МакКеон (Brian McKeon), Томас Фер (Thomas Fehr), Джозеф Эллиотт (Joseph Elliott), Габриэль да Силва Рибейро (Gabriel da Silva Ribeiro), Томас Фер (Thomas Fehr), Пабло Рипольес (Pablo Ripolles), Дуглас Гейлор (Douglas Gaylor), Кевин Сквайр (Kevin Squire), Гарри-Антон Талвик (Harry-Anton Talvik), Кристофер Симпкинс (Christopher Simpkins), Мартин Витман-Функ (Martin Witmann-Funk), Джим Балтер (Jim Balter), Питер Фостер (Peter Foster), Крейг Бордолон (Craig Bordelon), Хайнц-Питер Гум (Heinz-Peter Gumm), Питер Чапин (Peter Chapin), Кевин Райт (Kevin Wright), Анантан Сринивасан (Ananthan Srinivasan), Омар Килани (Omar Kilani), Дон Стефан (Donn Stephan), Гюнтер Ваффлер (Guenther Waffler).

Лекс хотел бы поблагодарить специалистов, среди которых Аарон Абрамс (Aaron Abrams), Джейсон Адамс (Jason Adams), Генри и Эмили Крутчер (Henry and Emily Crutcher), Джои Гибсон (Joey Gibson), Гунар Хиллерт (Gunnar Hillert), Мэтью Линк (Matthew Link), Тоби Рейлтс (Toby Reyelts), Джейсон Снейп (Jason Snape), Джон и Мелинда Уэзерс (John and Melinda Weathers), и всех представителей Atlanta Scala Enthusiasts за множество полезных обсуждений структуры языка, его математических основ и способов представления языка Scala специалистам-практикам.

Особую благодарность хочется выразить Дэйву Брикчетти (Dave Briccetti) и Адриану Мурсу (Adriaan Moors) за рецензирование третьего издания, а также Маркони Ланна (Marconi Lanna) не только за рецензирование, но и за мотивацию выпустить третье издание, которая возникла после разговора о новинках, появившихся со времени выхода предыдущего издания.

Билл хотел бы поблагодарить нескольких специалистов за предоставленную информацию и советы по изданию книги. Его благодарность заслужили Гэри Корнелл (Gary Cornell), Грег Доенч (Greg Doench), Энди Хант (Andy Hunt), Майк Леонард (Mike Leonard), Тайлер Ортман (Tyler Ortman), Билл Поллок (Bill Pollock), Дейв Томас (Dave Thomas) и Адам Райт (Adam Wright). Билл также хотел бы поблагодарить Дика Уолла (Dick Wall) за сотрудничество над разработкой нашего курса Stairway to Scala, который большей частью основывался на материалах, вошедших в эту книгу. Наш многолетний опыт преподавания курса Stairway to Scala помог повысить ее качество. И наконец, Билл хотел бы выразить благодарность Дарлин Грюндль (Darlene Gruendl) и Саманте Вулф (Samantha Woolf) за помощь в завершении третьего издания.

Наконец, мы хотели бы поблагодарить Жюльена Ричарда-Фоя за работу над обновлением четвертого издания этой книги до версии Scala 2.13, в частности за перепроектирование библиотеки коллекций.

Введение

Эта книга — руководство по Scala, созданное людьми, непосредственно занимающимися разработкой данного языка программирования. Нашей целью было научить вас всему, что необходимо для превращения в продуктивного программиста на языке Scala. Все примеры в книге компилируются Scala версии 2.13.1.

Целевая аудитория

Книга в основном рассчитана на программистов, желающих научиться программировать на Scala. Если у вас есть желание создать свой следующий проект на этом языке, то наша книга вам подходит. Кроме того, она должна заинтересовать программистов, которые хотят расширить кругозор, изучив новые концепции. Если вы, к примеру, программируете на Java, то эта книга раскроет для вас множество концепций функционального программирования, а также передовых мыслей из сферы объектно-ориентированного программирования. Мы уверены: изучение Scala и заложенных в этот язык идей поможет вам повысить свой профессиональный уровень как программиста. Предполагается, что вы уже владеете общими знаниями в области программирования. Scala вполне подходит на роль первого изучаемого языка, однако это не та книга, которая может использоваться для обучения программированию. В то же время вам не нужно быть каким-то особенным знатоком языков программирования. Большинство людей использует Scala на платформе Java, однако наша книга не предполагает, что вы тесно знакомы с языком Java. Но все же мы ожидаем, что Java известен многим читателям, и поэтому иногда сравниваем оба языка, чтобы помочь таким читателям понять разницу.

Как пользоваться книгой

Книгу рекомендуется читать в порядке следования глав, от начала до конца. Мы очень старались в каждой главе вводить читателя в курс только одной темы и объяснять новый материал лишь в понятиях из ранее рассмотренных тем. Поэтому если перескочить вперед, чтобы поскорее в чем-то разобраться, то можно встретить объяснения, в которых используются еще непонятные концепции. Мы считаем, что получать знания в области программирования на Scala лучше постепенно, читая главы в порядке их следования.

Обнаружив незнакомое понятие, можно обратиться к глоссарию. Многие читатели бегло просматривают части книги, и это вполне нормально. Но при встрече с непонятными терминами можно выяснить, что просмотр был слишком поверхностным, и вернуться к справочным материалам.

Прочитав книгу, вы можете в дальнейшем использовать ее в качестве справочника по Scala. Конечно, существует официальная спецификация языка, но в ней прослеживается стремление к точности в ущерб удобству чтения. В нашем издании не охвачены абсолютно все подробности Scala, но его особенности изложены в ней вполне обстоятельно. Так что по мере освоения программирования на этом языке издание вполне может стать доступным справочником.

Как изучать Scala

Весьма обширные познания о Scala можно получить, просто прочитав книгу от начала до конца. Но быстрее и основательнее освоить язык можно с помощью ряда дополнительных действий.

Прежде всего можно воспользоваться множеством примеров программ, включенных в эту книгу. Самостоятельно набирая их, вам придется вдумываться в каждую строку кода. Попытки его разнообразить позволят вам глубже заинтересоваться изучаемой темой и убедиться в том, что вы действительно поняли, как работает этот код.

Затем можно поучаствовать в работе множества онлайн-форумов. Это позволит вам и многим другим приверженцам Scala помочь друг другу в его освоении. Есть множество рассылок, дискуссионных форумов, чатов, вики-источников и несколько информационных каналов, которые посвящены этому языку и содержат соответствующие публикации. Уделите время поиску источников информации, более всего отвечающих вашим запросам. Вы будете гораздо быстрее решать мелкие проблемы, что позволит уделять больше времени более серьезным и глубоким вопросам.

И наконец, получив при чтении книги достаточный объем знаний, приступайте к разработке собственного проекта. Поработайте с нуля над созданием какой-нибудь небольшой программы или разработайте дополнение к более объемной. Вы не добьетесь быстрых результатов одним только чтением.

Условные обозначения

При первом упоминании какого-либо *понятия* или *термина* его название дается курсивом. Для небольших встроенных в текст примеров кода, таких как `x + 1`, используется моноширинный шрифт. Большие примеры кода представлены в виде отдельных блоков, для которых тоже используется моноширинный шрифт:

```
def hello() = {
  println("Hello, world!")
}
```

Когда показывается работа с интерактивной оболочкой, ответы последней выделяются шрифтом на сером фоне:

```
scala> 3 + 4
res0: Int = 7
```

Обзор глав

- Глава 1 «Масштабируемый язык» представляет обзор структуры языка Scala, а также ее логическое обоснование и историю.
- Глава 2 «Первые шаги в Scala» показывает, как в языке выполняется ряд основных задач программирования, не вдаваясь в подробности, касающиеся особенностей работы механизмов языка. Цель этой главы — заставить ваши пальцы набирать и запускать код на Scala.
- Глава 3 «Дальнейшие шаги в Scala» показывает ряд основных задач программирования, помогающих ускорить освоение этого языка. Изучив данную главу, вы сможете использовать Scala для автоматизации простых задач.
- Глава 4 «Классы и объекты» закладывает начало углубленного рассмотрения языка Scala, приводит описание его основных объектно-ориентированных строительных блоков и указания по выполнению компиляции и запуску приложений Scala.
- Глава 5 «Основные типы и операции» охватывает основные типы Scala, их литералы, операции, которые могут над ними проводиться, вопросы работы уровней приоритета и ассоциативности и дает представление об обогащающих оболочках.
- Глава 6 «Функциональные объекты» углубляет представление об объектно-ориентированных свойствах Scala, используя в качестве примера функциональные (то есть неизменяемые) рациональные числа.
- Глава 7 «Встроенные управляющие конструкции» показывает способы использования таких конструкций Scala, как `if`, `while`, `for`, `try` и `match`.
- Глава 8 «Функции и замыкания» углубленно рассматривает функции как основные строительные блоки функциональных языков.
- Глава 9 «Управляющие абстракции» показывает, как усовершенствовать основные управляющие конструкции Scala с помощью определения собственных управляющих абстракций.
- Глава 10 «Композиция и наследование» рассматривает имеющуюся в Scala дополнительную поддержку объектно-ориентированного программирования.

Затрагиваемые темы не столь фундаментальны, как те, что излагались в главе 4, но вопросы, которые в них рассматриваются, часто встречаются на практике.

- Глава 11 «Иерархия Scala» объясняет иерархию наследования языка и рассматривает универсальные методы и низшие типы.
- Глава 12 «Трейты» охватывает существующий в Scala механизм создания композиции примесей. Показана работа трейтов, описываются примеры их наиболее частого использования и объясняется, как с помощью трейтов совершенствуется традиционное множественное наследование.
- Глава 13 «Пакеты и импорты» рассматривает вопросы программирования в целом, включая высокоуровневые пакеты, инструкции импортирования и модификаторы управления доступом, такие как `protected` и `private`.
- Глава 14 «Утверждения и тесты» показывает существующий в Scala механизм утверждений и дает обзор ряда инструментов, доступных для написания тестов в этом языке. Основное внимание уделено средству ScalaTest.
- Глава 15 «Case-классы и сопоставление с образцом» вводит двойные конструкции, поддерживающие ваши действия при написании обычных, неинкапсулированных структур данных. `Case`-классы и сопоставление с образцом будут особенно полезны при работе с древовидными рекурсивными данными.
- Глава 16 «Работа со списками» подробно рассматривает списки, которые, вероятно, можно отнести к самым востребованным структурам данных в программах на Scala.
- Глава 17 «Работа с другими коллекциями» показывает способы использования основных коллекций Scala, таких как списки, массивы, кортежи, наборы и отображения.
- Глава 18 «Изменяемые объекты» объясняет суть изменяемых объектов и синтаксиса для выражения этих объектов, обеспечиваемого Scala. Глава завершается практическим примером моделирования дискретного события, в котором показан ряд изменяемых объектов в действии.
- Глава 19 «Параметризация типов» на конкретных примерах объясняет некоторые из технологических приемов скрытия информации, представленных в главе 13, например создание класса для функциональных запросов. Материалы главы подводят к описанию вариантности параметров типа и способов их взаимодействия с скрытием информации.
- Глава 20 «Абстрактные члены» дает описание всех разновидностей поддерживаемых Scala абстрактных членов — не только методов, но и полей и типов, которые можно объявлять абстрактными.
- Глава 21 «Неявные преобразования и параметры» рассматривает две конструкции, которые могут помочь избавить исходный код от излишней детализации, позволяя предоставить ее самому компилятору.

- Глава 22 «Реализация списков» описывает реализацию класса `List`. В ней рассматривается работа списков в Scala, в которой важно разбираться. Кроме того, реализация списков показывает, как используется ряд характерных особенностей этого языка.
- Глава 23 «Возвращение к выражениям for» показывает, как последние превращаются в вызовы методов `map`, `flatMap`, `filter` и `foreach`.
- Глава 24 «Углубленное изучение коллекций» предлагает углубленный обзор библиотеки коллекций.
- Глава 25 «Архитектура коллекций Scala» показывает устройство библиотеки коллекций и то, как можно реализовывать собственные коллекции.
- Глава 26 «Экстракторы» демонстрирует способы применения поиска по шаблонам в отношении не только `case`-классов, но и любых произвольных классов.
- Глава 27 «Аннотации» показывает, как можно работать с расширениями языка с помощью аннотаций. Описывает несколько стандартных аннотаций и описывает способы создания собственных аннотаций.
- Глава 28 «Работа с XML» объясняет порядок обработки в Scala соответствующих данных. Показаны идиомы для создания XML, проведения синтаксического анализа данных в этом формате и их обработки после анализа.
- Глава 29 «Модульное программирование с использованием объектов» показывает способы применения объектов Scala в качестве модульной системы.
- Глава 30 «Равенство объектов» освещает ряд проблемных вопросов, которые возникают при написании метода `equals`. Вдобавок рассматривается ряд узких мест, которые следует обходить стороной.
- Глава 31 «Сочетание кода на Scala и Java» рассматривает проблемы, возникающие при сочетании совместно используемых кодов на Scala и Java в одном и том же проекте, и предлагает способы, позволяющие находить решения таких проблем.
- Глава 32 «Фьючерсы и многопоточность» описывает способы применения класса `Future`. Для программ на Scala могут использоваться примитивы многопоточных вычислений и библиотеки, применяемые на Java-платформе. Однако фьючерсы помогут избежать возникновения условий взаимных блокировок и состояний гонки, которые мешают реализовывать традиционные подходы к многопоточности в виде потоков и блокировок.
- Глава 33 «Синтаксический разбор с помощью комбинаторов» показывает способы создания парсеров с использованием имеющейся в Scala библиотеки парсер-комбинаторов.
- Глава 34 «Программирование GUI» предлагает краткий обзор библиотеки Scala, упрощающей этот вид программирования с помощью среды Swing.
- Глава 35 «Электронная таблица SCells» связывает все воедино, показывая полнофункциональное приложение электронной таблицы, написанное на языке Scala.

Ресурсы

Самые последние версии Scala, ссылки на документацию и ресурсы сообщества можно найти на сайте www.scala-lang.org.

Исходный код

Исходный код, рассматриваемый в данной книге, выпущенный под открытой лицензией в виде ZIP-файла, можно найти на сайте книги: booksites.artima.com/programming_in_scala_4ed.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Масштабируемый язык

Scala означает «масштабируемый язык» (от англ. scalable language). Это название он получил, поскольку был спроектирован так, чтобы расти вместе с запросами своих пользователей. Язык Scala может решать широкий круг задач программирования: от написания небольших скриптов до создания больших систем[1].

Освоить Scala нетрудно. Он запускается на стандартной Java-платформе и полноценно взаимодействует со всеми Java-библиотеками. Язык очень хорошо подходит для написания скриптов, объединяющих Java-компоненты. Но его сильные стороны проявляются еще лучше при создании больших систем и сред повторно используемых компонентов.

С технической точки зрения Scala — смесь объектно-ориентированной и функциональной концепций программирования в статически типизированном языке. Подобный сплав проявляется во многих аспектах Scala — он, вероятно, может считаться более всеобъемлющим, чем другие широко используемые языки. Когда дело доходит до масштабируемости, два стиля программирования дополняют друг друга. Используемые в Scala конструкции функционального программирования упрощают быстрое создание интересных компонентов из простых частей. Объектно-ориентированные конструкции же облегчают структурирование больших систем и их адаптацию к новым требованиям. Сочетание двух стилей в Scala позволяет создавать новые виды шаблонов программирования и абстракций компонентов. Оно также способствует выработке понятного и лаконичного стиля программирования. И благодаря такой гибкости языка программирование на Scala может принести массу удовольствия.

В этой вступительной главе мы отвечаем на вопрос: «Почему именно Scala?» Мы даем общий обзор структуры Scala и ее обоснование. Прочитав главу, вы получите базовое представление о том, что такое Scala и с какого рода задачами он поможет справиться. Книга представляет собой руководство по языку Scala, однако данную главу нельзя считать частью этого руководства. И если вам не терпится приступить к написанию кода на Scala, то можете сразу перейти к изучению главы 2.

---

[1] Scala произносится как «ска́ла».

1.1. Язык, который растет вместе с вами

Программы различных размеров требуют, как правило, использования различных программных конструкций. Рассмотрим, к примеру, следующую небольшую программу на Scala:

```
var capital = Map("US" -> "Washington", "France" -> "Paris")

capital += ("Japan" -> "Tokyo")

println(capital("France"))
```

Эта программа устанавливает отображение стран на их столицы, модифицирует отображение, добавляя новую конструкцию (`"Japan" -> "Tokyo"`), и выводит название столицы, связанное со страной `France`[1]. В этом примере используется настолько высокоуровневая система записи, что она не загромождена ненужными точками с запятыми и сигнатурами типов. И действительно возникает ощущение использования современного языка скриптов наподобие Perl, Python или Ruby. Одна из общих характеристик этих языков, применимая к данному примеру, — поддержка всеми ими в синтаксисе языка конструкции ассоциативного отображения.

Ассоциативные отображения очень полезны, поскольку помогают поддерживать понятность и краткость программ, но порой вам может не подойти их философия «на все случаи жизни», поскольку вам в своей программе нужно управлять свойствами отображений более тонко. При необходимости Scala обеспечивает точное управление, поскольку отображения в нем не являются синтаксисом языка. Это библиотечные абстракции, которые можно расширять и приспосабливать под свои нужды.

В показанной ранее программе вы получите исходную реализацию отображения `Map`, но ее можно будет без особого труда изменить. К примеру, можно указать конкретную реализацию, такую как `HashMap` или `TreeMap`, или вызвать метод `par` для получения отображения `ParMap`, операции в котором выполняются параллельно. Можно указать для отображения значение по умолчанию или переопределить любой другой метод созданного вами отображения. Во всех случаях для отображений вполне пригоден такой же простой синтаксис доступа, как и в показанном примере.

В нем показано, что Scala может обеспечить вам как удобство, так и гибкость. Язык содержит набор удобных конструкций, которые помогают быстро начать работу и позволяют программировать в приятном лаконичном стиле. В то же время есть гарантии, что вы не сможете перерасти язык. Вы всегда сможете перекроить программу под свои требования, поскольку все в ней основано на библиотечных модулях, которые можно выбрать и приспособить под свои нужды.

---

[1] Пожалуйста, не сердитесь на нас, если не сможете разобраться со всеми тонкостями этой программы. Объяснения будут даны в двух следующих главах.

Растут новые типы

Эрик Рэймонд (Eric Raymond) в качестве двух метафор разработки программных продуктов ввел собор и базар[1]. Под собором понимается почти идеальная разработка, создание которой требует много времени. После сборки она долго остается неизменной. Разработчики же базара, напротив, что-то адаптируют и дополняют каждый день. В книге Рэймонда базар — метафора, описывающая разработку ПО с открытым кодом. Гай Стил (Guy Steele) отметил в докладе о «растущем языке», что аналогичное различие можно применить к структуре языка программирования[2]. Scala больше похож на базар, чем на собор, в том смысле, что спроектирован с расчетом на расширение и адаптацию его теми, кто на нем программирует. Вместо того чтобы предоставлять все конструкции, которые только могут пригодиться в одном всеобъемлющем языке, Scala дает вам инструменты для создания таких конструкций.

Рассмотрим пример. Многие приложения нуждаются в целочисленном типе, который при выполнении арифметических операций может становиться произвольно большим без переполнения или циклического перехода в начало. В Scala такой тип определяется в библиотеке класса `scala.math.BigInt`. Определение использующего этот тип метода, который вычисляет факториал переданного ему целочисленного значения, имеет следующий вид[3]:

```
def factorial(x: BigInt): BigInt =
  if (x == 0) 1 else x * factorial(x - 1)
```

Теперь, вызвав `factorial(30)`, вы получите:

```
265252859812191058636308480000000
```

Тип `BigInt` похож на встроенный, поскольку со значениями этого типа можно использовать целочисленные литералы и операторы наподобие * и –. Тем не менее это просто класс, определение которого задано в стандартной библиотеке Scala[4]. Если бы класса не было, то любой программист на Scala мог бы запросто написать его реализацию, например создав оболочку для имеющегося в языке Java класса `java.math.BigInteger` (фактически именно так и реализован класс `BigInt` в Scala).

---

1 *Raymond E.* The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. — O'Reilly, 1999.

2 *Steele Jr., Guy L.* Growing a Language // Higher-Order and Symbolic Computation, 12:221–223, 1999. Транскрипция речи, произнесенной на OOPSLA в 1998 году.

3 factorial(x), или x! в математической записи, — результат вычисления $1 \times 2 \ldots x$, где для 0! определено значение 1.

4 Scala поставляется со стандартной библиотекой, часть которой будет рассмотрена в книге. За дополнительной информацией можно обратиться к имеющейся в библиотеке документации Scaladoc, доступной в дистрибутиве и в Интернете по адресу www.scala-lang.org.

Конечно же, класс Java можно использовать напрямую. Но результат будет не столь приятным: хоть Java и позволяет вам создавать новые типы, они не производят впечатление получающих естественную поддержку языка:

```
import java.math.BigInteger

def factorial(x: BigInteger): BigInteger =
  if (x == BigInteger.ZERO)
    BigInteger.ONE
  else
    x.multiply(factorial(x.subtract(BigInteger.ONE)))
```

Тип `BigInt` — один из многих других числовых типов: больших десятичных чисел, комплексных и рациональных чисел, доверительных интервалов, полиномов, — и данный список можно продолжить. В некоторых языках программирования часть этих типов реализуется естественным образом. Например, в Lisp, Haskell и Python есть большие целые числа, в Fortran и Python — комплексные. Но любой язык, в котором пытаются одновременно реализовать все эти абстракции, разрастается до таких размеров, что становится неуправляемым. Более того, даже существуй подобный язык, нашлись бы приложения, требующие других числовых типов, которые все равно не были бы представлены. Следовательно, подход, при котором предпринимается попытка реализовать все в одном языке, не позволяет получить хорошую масштабируемость. Язык Scala, напротив, дает пользователям возможность наращивать и адаптировать его в нужных направлениях. Он делает это с помощью определения простых в использовании библиотек, которые производят *впечатление* средств, естественно реализованных в языке.

Растут новые управляющие конструкции

Предыдущий пример показывает, что Scala позволяет добавлять новые типы, использовать которые так же удобно, как и встроенные. Тот же принцип расширения применяется и к управляющим конструкциям. Этот вид расширения иллюстрируется с помощью Akka — API Scala для многопоточного программирования, основанного на использовании акторов.

Поскольку распространение многоядерных процессоров в ближайшие годы продолжится, то для достижения приемлемой производительности все чаще может требоваться более высокая степень распараллеливания ваших приложений. Зачастую это чревато переписыванием кода с целью распределить вычисления по нескольким параллельным потокам. К сожалению, на практике создание надежных многопоточных приложений — весьма непростая задача. Модель потоковой обработки в Java выстроена вокруг совместно используемой памяти и блокировок и часто трудно поддается осмыслению, особенно по мере масштабирования систем и увеличения их объема и сложности. Обеспечить отсутствие состояния гонок или скрытых взаимных блокировок, не выявляемых в ходе тестирования, но способных проявиться в ходе эксплуатации, довольно трудно. Возможно, более безопасным альтернативным решением станет использование архитектуры

передачи сообщений, подобной применению акторов в языке программирования Erlang.

Java поставляется с богатой библиотекой параллельных вычислений, основанной на применении потоков. Программы на Scala могут задействовать ее точно так же, как и любой другой API Java. Но есть еще Akka — дополнительная библиотека Scala, реализующая модель акторов, похожую на ту, что используется в Erlang.

Акторы — абстракции параллельных вычислений, которые могут быть реализованы в качестве надстроек над потоками. Они обмениваются данными, отправляя друг другу сообщения. Актор может выполнить две основные операции: отправку сообщения и его получение. Операция отправки, обозначаемая восклицательным знаком (`!`), отправляет сообщение актору. Пример, в котором фигурирует актор с именем `recipient`, выглядит следующим образом:

```
recipient ! msg
```

Отправка осуществляется в асинхронном режиме, то есть отправляющий сообщение актор может продолжить выполнение кода, не дожидаясь получения и обработки сообщения. У каждого актора имеется *почтовый ящик*, в котором входящие сообщения выстраиваются в очередь. Актор обрабатывает сообщения, поступающие в его почтовый ящик, в блоке получения `receive`:

```
def receive = {
  case Msg1 => ... // обработка Msg1
  case Msg2 => ... // обработка Msg2
  // ...
}
```

Блок `receive` состоит из нескольких инструкций выбора `case`, в каждой из которых содержится запрос к почтовому ящику на соответствие шаблону сообщения. В ящике выбирается первое же сообщение, которое соответствует условию какой-либо инструкции выбора, и в отношении него выполняется какое-либо действие. Когда в почтовом ящике не оказывается сообщений, актор приостанавливает работу и ожидает их дальнейшего поступления.

В качестве примера рассмотрим простой Akka-актор, реализующий сервис подсчета контрольной суммы:

```
class ChecksumActor extends Actor {
  var sum = 0
  def receive = {
    case Data(byte) => sum += byte
    case GetChecksum(requester) =>
      val checksum = ~(sum & 0xFF) + 1
      requester ! checksum
  }
}
```

Сначала в акторе определяется локальная переменная `sum` с начальным нулевым значением. Затем — блок получения `receive`, который будет обрабатывать сообщения. При получении сообщения `Data` он прибавляет значение содержащегося в нем аргумента `byte` к значению переменной `sum`. При получении сообщения `GetChecksum`

он вычисляет контрольную сумму из текущего значения переменной `sum` и отправляет результат обратно в адрес пославшего запрос `requester`, используя код отправки сообщения `requester ! checksum`. Поле `requester` отправляется в сообщении `GetChecksum`, обычно оно ссылается на актор, пославший запрос.

Мы не ожидаем, что прямо сейчас вы полностью поймете пример использования актора. Скорее, существенным обстоятельством для темы масштабируемости в этом примере служит тот факт, что ни блок `receive`, ни инструкция отправки сообщения (`!`) не являются встроенными операциями Scala. Даже притом, что внешний вид и работа блока `receive` во многом напоминают встроенную управляющую конструкцию, на самом деле это метод, определенный в библиотеке акторов Akka. Аналогично даже притом, что оператор `!` похож на встроенный оператор, он также является всего лишь методом, определенным в библиотеке акторов Akka. Обе конструкции абсолютно независимы от языка программирования Scala.

Синтаксис блока `receive` и оператора отправки (`!`) выглядят в Scala во многом так же, как в Erlang, но в последнем эти конструкции встроены в язык. Кроме того, в Akka реализуется большинство других конструкций многопоточного программирования, имеющихся в Erlang, например конструкции отслеживания сбойных акторов и истечения времени ожидания. В целом модель акторов показала себя весьма удачным средством для выражения многопоточных и распределенных вычислений. Несмотря на то что акторы должны быть определены в библиотеке, они могут рассматриваться как составная часть языка Scala.

Пример иллюстрирует возможность наращивания языка Scala в новых направлениях, даже в таких специализированных, как программирование многопоточных приложений. Разумеется, для этого понадобятся квалифицированные разработчики и программисты. Но важен сам факт наличия такой возможности: вы можете проектировать и реализовывать в Scala абстракции, которые относятся к принципиально новым прикладным областям, но при использовании воспринимаются как естественно поддерживаемые самим языком.

1.2. Что делает Scala масштабируемым языком

На возможность масштабирования влияет множество факторов, от особенностей синтаксиса до структуры абстрактных компонентов. Но если бы потребовалось назвать всего один аспект Scala, который способствует масштабируемости, то мы бы выбрали присущее этому языку сочетание объектно-ориентированного и функционального программирования (мы немного слукавили, на самом деле это два аспекта, но они взаимосвязаны).

Scala в объединении объектно-ориентированного и функционального программирования в однородную структуру языка пошел дальше всех остальных широко известных языков. Например, там, где в других языках объекты и функции — два разных понятия, в Scala функция по смыслу *является* объектом. Функциональные типы являются классами, которые могут наследоваться подклассами. Эти особенности могут показаться не более чем теоретическими, но имеют весьма серьезные

последствия для возможностей масштабирования. Фактически ранее упомянутое понятие актора не может быть реализовано без этой унификации функций и объектов. В данном разделе дается обзор примененных в Scala способов смешивания объектно-ориентированной и функциональной концепций.

Scala — объектно-ориентированный язык

Развитие объектно-ориентированного программирования шло весьма успешно. Появившись в языке Simula в середине 1960-х годов и в Smalltalk в 1970-х, оно теперь доступно в подавляющем большинстве языков. В некоторых областях все полностью захвачено объектами. Точного определения «объектной ориентированности» нет, однако объекты явно чем-то привлекают программистов.

В принципе, мотивация для применения объектно-ориентированного программирования очень проста: все, за исключением самых простых программ, нуждается в определенной структуре. Наиболее понятный путь достижения желаемого результата заключается в помещении данных и операций в своеобразные контейнеры. Основной замысел объектно-ориентированного программирования состоит в придании этим контейнерам полной универсальности, чтобы в них могли содержаться не только операции, но и данные и чтобы сами они также были элементами, которые могли бы храниться в других контейнерах или передаваться операциям в качестве параметров. Подобные контейнеры называются объектами. Алан Кей (Alan Kay), изобретатель языка Smalltalk, заметил, что таким образом простейший объект имеет принцип построения, аналогичный полноценному компьютеру: под формализованным интерфейсом данные в нем сочетаются с операциями[1]. То есть объекты имеют непосредственное отношение к масштабируемости языка: одни и те же технологии применяются к построению как малых, так и больших программ.

Хотя долгое время объектно-ориентированное программирование преобладало, немногие языки стали последователями Smalltalk по части внедрения этого принципа построения в свое логическое решение. Например, множество языков допускают использование элементов, не являющихся объектами, — можно вспомнить имеющиеся в языке Java значения примитивных типов. Или же в них допускается применение статических полей и методов, не входящих в какой-либо объект. Эти отклонения от чистой идеи объектно-ориентированного программирования на первый взгляд выглядят вполне безобидными, но имеют досадную тенденцию к усложнению и ограничению масштабирования.

В отличие от этого, Scala — объектно-ориентированный язык в чистом виде: каждое значение является объектом и каждая операция — вызовом метода. Например, когда в Scala речь заходит о вычислении `1 + 2`, фактически вызывается метод по имени `+`, который определен в классе `Int`. Можно определять методы с именами, похожими на операторы, а клиенты вашего API смогут с помощью этих методов записать

[1] *Kay A. C.* The Early History of Smalltalk // History of programming languages. — II, P. 511–598. — New York: ACM, 1996 [Электронный ресурс]. — Режим доступа: http://doi.acm.org/10.1145/234286.1057828.

операторы. Именно так API акторов Akka позволяет вам воспользоваться выражением `requester ! checksum`, показанным в предыдущем примере: `!` — метод класса `Actor`.

Когда речь заходит о составлении объектов, Scala проявляется как более совершенный язык по сравнению с большинством других. В качестве примера приведем имеющиеся в Scala *трейты*. Они подобны интерфейсам в Java, но могут содержать также реализации методов и даже поля[1]. Объекты создаются путем *композиции примесей*, при котором к членам класса добавляются члены нескольких трейтов. Таким образом, различные аспекты классов могут быть инкапсулированы в различных трейтах. Это выглядит как множественное наследование, но есть разница в конкретных деталях. В отличие от класса трейт может добавить в суперкласс новые функциональные возможности. Это придает трейтам более высокую степень подключаемости по сравнению с классами. В частности, благодаря этому удается избежать возникновения присущих множественному наследованию классических проблем «ромбовидного» наследования, которые возникают, когда один и тот же класс наследуется по нескольким различным путям.

Scala — функциональный язык

Наряду с тем, что Scala является чистым объектно-ориентированным языком, его можно назвать и полноценным функциональным языком. Идеи функционального программирования старше электронных вычислительных систем. Их основы были заложены в лямбда-исчислении Алонзо Чёрча (Alonzo Church), разработанном в 1930-е годы. Первым языком функционального программирования был Lisp, появление которого датируется концом 1950-х. К другим популярным функциональным языкам относятся Scheme, SML, Erlang, Haskell, OCaml и F#. Долгое время функциональное программирование играло второстепенные роли — будучи популярным в научных кругах, оно не столь широко использовалось в промышленности. Но в последние годы интерес к его языкам и технологиям растет.

Функциональное программирование базируется на двух основных идеях. Первая заключается в том, что функции являются значениями первого класса. В функциональных языках функция есть значение, имеющее такой же статус, как целое число или строка. Функции можно передавать в качестве аргументов другим функциям, возвращать их в качестве результатов из других функций или сохранять в переменных. Вдобавок функцию можно определять внутри другой функции точно так же, как это делается при определении внутри функции целочисленного значения. И функции можно определять, не присваивая им имен, добавляя в код функциональные литералы с такой же легкостью, как и целочисленные, наподобие `42`.

Функции как значения первого класса — удобное средство абстрагирования, касающееся операций и создания новых управляющих конструкций. Эта универсальность функций обеспечивает более высокую степень выразительности, что зачастую приводит к созданию весьма разборчивых и кратких программ. Она также

[1] Начиная с Java 8, у интерфейсов могут быть реализации методов по умолчанию, но они не предлагают всех тех возможностей, которые есть у трейтов языка Scala.

играет важную роль в обеспечении масштабируемости. В качестве примера библиотека тестирования ScalaTest предлагает конструкцию `eventually`, получающую функцию в качестве аргумента. Данная конструкция используется следующим образом:

```
val xs = 1 to 3
val it = xs.iterator
eventually { it.next() shouldBe 3 }
```

Код внутри `eventually`, являющийся утверждением, `it.next() shouldBe 3`, включает в себя функцию, передаваемую невыполненной в метод `eventually`. Через настраиваемый период времени `eventually` станет неоднократно выполнять функцию до тех пор, пока утверждение не будет успешно подтверждено.

В отличие от этого в большинстве традиционных языков функции не являются значениями. Языки, в которых есть функциональные значения, относят их статус ко второму классу. Например, указатели на функции в C и C++ не имеют в этих языках такого же статуса, как другие значения, не относящиеся к функциям: указатели на функции могут только ссылаться на глобальные функции, не позволяя определять вложенные функции первого класса, имеющие некоторые значения в их окружении. Они также не позволяют определять литералы безымянных функций.

Вторая основная идея функционального программирования заключается в том, что операции программы должны преобразовать входные значения в выходные, а не изменять данные на месте. Чтобы понять разницу, рассмотрим реализацию строк в Ruby и Java. В Ruby строка является массивом символов. Символы в строке могут быть изменены по отдельности. Например, внутри одного и того же строкового объекта символ точки с запятой в строке можно заменить точкой. А в Java и Scala строка — последовательность символов в математическом смысле. Замена символа в строке с использованием выражения вида `s.replace(';', '.')` приводит к возникновению нового строкового объекта, отличающегося от `s`. То же самое можно сказать по-другому: в Java строки неизменяемые, а в Ruby — изменяемые. То есть, рассматривая только строки, можно прийти к выводу, что Java — функциональный язык, а Ruby — нет. Неизменяемая структура данных — один из краеугольных камней функционального программирования. В библиотеках Scala в качестве надстроек над соответствующими API Java определяется также множество других неизменяемых типов данных. Например, в Scala имеются неизменяемые списки, кортежи, отображения и множества.

Еще один способ утверждения второй идеи функционального программирования заключается в том, что у методов не должно быть никаких *побочных эффектов*. Они должны обмениваться данными со своим окружением только путем получения аргументов и возвращения результатов. Например, под это описание подпадает метод `replace`, принадлежащий Java-классу `String`. Он получает строку и два символа и выдает новую строку, где все появления одного символа заменены появлениями второго. Других эффектов от вызова `replace` нет. Методы, подобные `replace`, называются *ссылочно прозрачными*. Это значит, что для любого заданного ввода вызов функции можно заменить его результатом, при этом семантика программы остается неизменной.

Функциональные языки заставляют применять неизменяемые структуры данных и ссылочно прозрачные методы. В некоторых функциональных языках это выражено в виде категоричных требований. Scala же дает возможность выбрать. При желании можно писать программы в *императивном* стиле — так называется программирование с изменяемыми данными и побочными эффектами. Но при необходимости в большинстве случаев Scala позволяет с легкостью избежать использования императивных конструкций благодаря существованию хороших функциональных альтернатив.

1.3. Почему именно Scala

Подойдет ли вам язык Scala? Разбираться и принимать решение придется самостоятельно. Мы считаем, что, помимо хорошей масштабируемости, существует еще множество причин, по которым вам может понравиться программирование на Scala. В этом разделе будут рассмотрены четыре наиболее важных аспекта: совместимость, лаконичность, абстракции высокого уровня и расширенная статическая типизация.

Scala — совместимый язык

Scala не требует резко отходить от платформы Java, чтобы опередить на шаг этот язык. Scala позволяет повышать ценность уже существующего кода, то есть опираться на то, что у вас уже есть, поскольку он был разработан для достижения беспрепятственной совместимости с Java[1]. Программы на Scala компилируются в байт-коды виртуальной машины Java (JVM). Производительность при выполнении этих кодов находится на одном уровне с производительностью программ на Java. Код Scala может вызывать методы Java, обращаться к полям этого языка, поддерживать наследование от его классов и реализовывать его интерфейсы. Для всего перечисленного не требуются ни специальный синтаксис, ни явные описания интерфейса, ни какой-либо связующий код. По сути, весь код Scala интенсивно использует библиотеки Java, зачастую даже без ведома программистов.

Еще один аспект полной совместимости — интенсивное заимствование в Scala типов данных Java. Данные типа `Int` в Scala представлены в виде имеющегося в Java примитивного целочисленного типа `int`, соответственно `Float` представлен как `float`, `Boolean` — как `boolean` и т. д. Массивы Scala отображаются на массивы Java. В Scala из Java позаимствованы и многие стандартные библиотечные типы. Например, тип строкового литерала `"abc"` в Scala фактически представлен классом `java.lang.String`, а исключение должно быть подклассом `java.lang.Throwable`.

[1] Изначально существовала реализация Scala, запускаемая на платформе .NET, но она больше не используется. В последнее время все большую популярность набирает реализация Scala под названием Scala.js, запускаемая на JavaScript.

Java-типы в Scala не только заимствованы, но и «принаряжены» для придания им привлекательности. Например, строки в Scala поддерживают такие методы, как `toInt` или `toFloat`, которые преобразуют строки в целое число или число с плавающей точкой. То есть вместо `Integer.parseInt(str)` вы можете написать `str.toInt`. Как такое возможно без нарушения совместимости? Ведь в Java-классе `String` нет метода `toInt`! По сути, в Scala имеется универсальное решение для устранения противоречия между сложной структурой библиотеки и совместимостью. Scala позволяет определять *неявные преобразования*, которые всегда применяются при несовпадении типов или выборе несуществующих элементов. В рассматриваемом случае при поиске метода `toInt` для работы со строковым значением компилятор Scala не найдет такого элемента в классе `String`. Однако он найдет неявное преобразование, превращающее Java-класс `String` в экземпляр Scala-класса `StringOps`, в котором такой элемент определен. Затем преобразование будет автоматически применено, прежде чем будет выполнена операция `toInt`.

Код Scala также может быть вызван из кода Java. Иногда при этом следует учитывать некоторые нюансы. Scala — более богатый язык, чем Java, и потому некоторые расширенные функции Scala должны быть закодированы, прежде чем могут быть отображены на код Java. Подробности объясняются в главе 31.

Scala — лаконичный язык

Программы на Scala, как правило, отличаются краткостью. Программисты, работающие с данным языком, отмечают сокращение количества строк почти на порядок по сравнению с Java. Но это можно считать крайним случаем. Более консервативные оценки свидетельствуют о том, что обычная программа на Scala должна умещаться в половину тех строк, которые используются для аналогичной программы на Java. Меньшее количество строк означает не только сокращение объема набираемого текста, но и экономию сил при чтении и осмыслении программ, а также уменьшение количества возможных недочетов. Свой вклад в сокращение количества строк кода вносят сразу несколько факторов.

В синтаксисе Scala не используются некоторые шаблонные элементы, отягощающие программы на Java. Например, в Scala не обязательно применять точки с запятыми. Есть и несколько других областей, где синтаксис Scala менее зашумлен. В качестве примера можно сравнить, как записывается код классов и конструкторов в Java и Scala. В Java класс с конструктором зачастую выглядит следующим образом:

```
// Это Java
class MyClass {

    private int index;
    private String name;

    public MyClass(int index, String name) {
        this.index = index;
        this.name = name;
    }
}
```

А в Scala, скорее всего, будет использована такая запись:

```
class MyClass(index: Int, name: String)
```

Получив указанный код, компилятор Scala создаст класс с двумя приватными переменными экземпляра (типа `Int` по имени `index` и типа `String` по имени `name`) и конструктор, который получает исходные значения для этих переменных в виде параметров. Код данного конструктора проинициализирует две переменные экземпляра значениями, переданными в качестве параметров. Короче говоря, в итоге вы получите ту же функциональность, что и у более многословной версии кода на Java[1]. Класс в Scala быстрее пишется и проще читается, а еще — и это наиболее важно — допустить ошибку при его создании значительно труднее, чем при создании класса в Java.

Еще один фактор, способствующий лаконичности, — используемый в Scala вывод типов. Повторяющуюся информацию о типе можно отбросить, и тогда программы избавятся от лишнего и их легче будет читать.

Но, вероятно, наиболее важный аспект сокращения объема кода — наличие кода, не требующего внесения в программу, поскольку это уже сделано в библиотеке. Scala предоставляет вам множество инструментальных средств для определения эффективных библиотек, позволяющих выявить и вынести за скобки общее поведение. Например, различные аспекты библиотечных классов можно выделить в трейты, которые затем можно перемешивать произвольным образом. Или же библиотечные методы могут быть параметризованы с помощью операций, позволяя вам определять конструкции, которые, по сути, являются вашими собственными управляющими конструкциями. Собранные вместе, эти конструкции позволяют определять библиотеки, сочетающие в себе высокоуровневый характер и гибкость.

Scala — язык высокого уровня

Программисты постоянно борются со сложностью. Для продуктивного программирования нужно понимать код, над которым вы работаете. Чрезмерно сложный код был причиной краха многих программных проектов. К сожалению, важные программные продукты обычно бывают весьма сложными. Избежать сложности невозможно, но ею можно управлять.

Scala помогает управлять сложностью, позволяя повышать уровень абстракции в разрабатываемых и используемых интерфейсах. Представим, к примеру, что есть переменная `name`, имеющая тип `String`, и нужно определить, наличествует ли в этой строковой переменной символ в верхнем регистре. До выхода Java 8 приходилось создавать следующий цикл:

---

[1] Единственное отличие заключается в том, что переменные экземпляра, полученные в случае применения Scala, будут финальными (final). Как сделать их не финальными, рассказывается в разделе 10.6.

```
boolean nameHasUpperCase = false; // Это Java
for (int i = 0; i < name.length(); ++i) {
    if (Character.isUpperCase(name.charAt(i))) {
        nameHasUpperCase = true;
        break;
    }
}
```

А в Scala можно написать такой код:

```
val nameHasUpperCase = name.exists(_.isUpper)
```

Код Java считает строки низкоуровневыми элементами, требующими посимвольного перебора в цикле. Код Scala рассматривает те же самые строки как высокоуровневые последовательности символов, в отношении которых можно применять запросы с *предикатами*. Несомненно, код Scala намного короче и — для натренированного глаза — более понятен, чем код Java. Следовательно, код Scala значительно меньше влияет на общую сложность приложения. Кроме того, уменьшается вероятность допустить ошибку.

Предикат `_.isUpper` — пример используемого в Scala функционального литерала[1]. В нем дается описание функции, которая получает аргумент в виде символа (представленного знаком подчеркивания) и проверяет, не является ли этот символ буквой в верхнем регистре[2].

В Java 8 появилась поддержка *лямбда-выражений* и *потоков* (streams), позволяющая выполнять подобные операции на Java. Вот как это могло бы выглядеть:

```
boolean nameHasUpperCase = // Это Java 8
    name.chars().anyMatch(
        (int ch) -> Character.isUpperCase((char) ch)
);
```

Несмотря на существенное улучшение по сравнению с более ранними версиями Java, код Java 8 все же более многословен, чем его эквивалент на языке Scala. Излишняя тяжеловесность кода Java, а также давняя традиция использования в этом языке циклов может натолкнуть многих Java-программистов на мысль о необходимости новых методов, подобных `exists`, позволяющих просто переписать циклы и смириться с растущей сложностью кода.

В то же время функциональные литералы в Scala действительно воспринимаются довольно легко и задействуются очень часто. По мере углубления знакомства со Scala перед вами будут открываться все больше и больше возможностей для определения и использования собственных управляющих абстракций. Вы поймете, что это поможет избежать повторений в коде, сохраняя лаконичность и чистоту программ.

---

[1] Функциональный литерал может называться предикатом, если результирующим типом будет Boolean.

[2] Такое использование символа подчеркивания в качестве заместителя для аргументов рассматривается в разделе 8.5.

Scala — статически типизированный язык

Системы со статической типизацией классифицируют переменные и выражения в соответствии с видом хранящихся и вычисляемых значений. Scala выделяется как язык своей совершенной системой статической типизации. Обладая системой вложенных типов классов, во многом похожей на имеющуюся в Java, этот язык позволяет вам проводить параметризацию типов с помощью средств *обобщенного программирования*, комбинировать типы с использованием *пересечений* и скрывать особенности типов, применяя *абстрактные типы*[1]. Так формируется прочный фундамент для создания собственных типов, который дает возможность разрабатывать безопасные и в то же время гибкие в использовании интерфейсы.

Если вам нравятся динамические языки, такие как Perl, Python, Ruby или Groovy, то вы можете посчитать немного странным факт, что система статических типов в Scala упоминается как одна из его сильных сторон. Ведь отсутствие такой системы часто называют основным преимуществом динамических языков. Наиболее часто, говоря о ее недостатках, приводят такие аргументы, как присущая программам многословность, воспрепятствование свободному самовыражению программистов и невозможность применения конкретных шаблонов динамических изменений программных систем. Но зачастую эти аргументы направлены не против идеи статических типов в целом, а против конкретных систем типов, воспринимаемых как слишком многословные или недостаточно гибкие. Например, Алан Кей, автор языка Smalltalk, однажды заметил: «Я не против типов, но не знаю ни одной беспроблемной системы типов. Так что мне все еще нравится динамическая типизация»[2].

В этой книге я надеюсь убедить вас в том, что система типов в Scala далека от проблемной. На самом деле она вполне изящно справляется с двумя обычными опасениями, связываемыми со статической типизацией: многословия удается избежать за счет логического вывода типов, а гибкость достигается благодаря сопоставлению с образцом и ряду новых способов записи и составления типов. По мере устранения этих препятствий к классическим преимуществам систем статических типов начинают относиться намного более благосклонно. Среди наиболее важных преимуществ можно назвать верифицируемые свойства программных абстракций, безопасный рефакторинг и более качественное документирование.

Верифицируемые свойства. Системы статических типов способны подтверждать отсутствие конкретных ошибок, выявляемых в ходе выполнения программы. Это могут быть следующие правила: булевы значения никогда не складываются с целыми числами; приватные переменные недоступны за пределами своего класса;

---

[1] Обобщенные типы рассматриваются в главе 19, пересечения (например, A с B и с C) — в главе 12, а абстрактные типы — в главе 20.

[2] *Kay A. C.* Письмо, адресованное Стефану Раму, с описанием термина «Объектно-ориентированное программирование». Июль 2003 [Электронный ресурс]. — Режим доступа: www.purl.org/stefan_ram/pub/doc_kay_oop_en.

функции применяются к надлежащему количеству аргументов; во множество строк можно добавлять только строки.

Существующие в настоящее время системы статических типов не выявляют ошибки других видов. Например, обычно они не обнаруживают бесконечные функции, нарушение границ массивов или деление на ноль. Вдобавок эти системы не смогут определить несоответствие вашей программы ее спецификации (при наличии таковой!). Поэтому некоторые отказываются от них, считая не слишком полезными. Аргументация такова: если эти системы могут выявлять только простые ошибки, а модульные тесты обеспечивают более широкий охват, то зачем вообще связываться со статическими типами? Мы считаем, что в этих аргументах упущено главное. Система статических типов, конечно же, не может *заменить* собой модульное тестирование, однако может сократить количество необходимых модульных тестов, выявляя некие свойства, которые в противном случае нужно было бы протестировать. А модульное тестирование не способно заменить статическую типизацию. Ведь Эдсгер Дейкстра (Edsger Dijkstra) сказал, что тестирование позволяет убедиться лишь в наличии ошибок, но не в их отсутствии[1]. Гарантии, которые обеспечиваются статической типизацией, могут быть простыми, но это реальные гарантии, не способные обеспечить никакие объемы тестирования.

Безопасный рефакторинг. Системы статических типов дают гарантии, позволяющие вам вносить изменения в основной код, будучи совершенно уверенными в благополучном исходе этого действия. Рассмотрим, к примеру, рефакторинг, при котором к методу нужно добавить еще один параметр. В статически типизированном языке вы можете внести изменения, перекомпилировать систему и просто исправить те строки, которые вызовут ошибку типа. Сделав это, вы будете пребывать в уверенности, что были найдены все места, требовавшие изменений. То же самое справедливо для другого простого рефакторинга, например изменения имени метода или перемещения метода из одного класса в другой. Во всех случаях проверка статического типа позволяет быть вполне уверенными в том, что работоспособность новой системы осталась на уровне работоспособности старой.

Документирование. Статические типы — документация программы, проверяемой компилятором на корректность. В отличие от обычного комментария, аннотация типа никогда не станет устаревшей (по крайней мере, если содержащий ее исходный файл недавно успешно прошел компиляцию). Более того, компиляторы и интегрированные среды разработки (integrated development environments, IDE) могут использовать аннотации для выдачи более качественной контекстной справки. Например, IDE может вывести на экран все элементы, доступные для выбора, путем определения статического типа выражения, которое выбрано, и дать возможность просмотреть все элементы этого типа.

Хотя статические типы в целом полезны для документирования программы, иногда они могут вызывать раздражение тем, что засоряют ее. Обычно полезным

---

[1] *Dijkstra E. W.* Notes on Structured Programming. — Апрель 1970 [Электронный ресурс]. — Режим доступа: www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF.

считается документирование тех сведений, которые читателям программы самостоятельно извлечь довольно трудно. Полезно знать, что в методе, определенном так:

```
def f(x: String) = ...
```

аргументы метода `f` должны принадлежать типу `String`. В то же время может вызвать раздражение по крайней мере одна из двух аннотаций в следующем примере:

```
val x: HashMap[Int, String] = new HashMap[Int, String]()
```

Понятно, что было бы достаточно показать отношение `x` к типу `HashMap` с `Int`-типами в качестве ключей и `String`-типами в качестве значений только один раз, дважды повторять одно и то же нет смысла.

В Scala имеется весьма сложная система логического вывода типов, позволяющая опускать почти всю информацию о типах, которая обычно вызывает раздражение. В предыдущем примере вполне работоспособны и две менее раздражающие альтернативы:

```
val x = new HashMap[Int, String]()
val x: Map[Int, String] = new HashMap()
```

Вывод типа в Scala может заходить довольно далеко. Фактически пользовательский код нередко вообще обходится без явного задания типов. Поэтому программы на Scala часто выглядят похожими на программы, написанные на динамически типизированных языках скриптов. Это, в частности, справедливо для прикладного клиентского кода, который склеивается из заранее написанных библиотечных компонентов. Но для них самих это менее характерно, поскольку в них зачастую применяются довольно сложные типы, не допускающие гибкого использования таких схем. И это вполне естественно. Ведь сигнатуры типов элементов, составляющих интерфейс повторно используемых компонентов, должны задаваться в явном виде, поскольку составляют существенную часть соглашения между компонентом и его клиентами.

1.4. Истоки Scala

На идею создания Scala повлияли многие языки программирования и идеи, выработанные на основе исследований таких языков. Фактически обновления в Scala незначительны — большинство характерных особенностей языка уже применялось в том или ином виде в других языках программирования. Инновации в Scala появляются в основном из того, как его конструкции сводятся воедино. В этом разделе будут перечислены основные факторы, оказавшие влияние на структуру языка Scala. Перечень не может быть исчерпывающим, поскольку в дизайне языков программирования так много толковых идей, что перечислить здесь их все просто невозможно.

На внешнем уровне Scala позаимствовал существенную часть синтаксиса у Java и C#, которые, в свою очередь, взяли большинство своих синтаксических соглаше-

ний у C и C++. Выражения, инструкции и блоки — в основном из Java, как, собственно, и синтаксис классов, создание пакетов и импорт[1]. Кроме синтаксиса, Scala позаимствовал и другие элементы Java, такие как его основные типы, библиотеки классов и модель выполнения.

Scala многим обязан и другим языкам. Его однородная модель объектов впервые появилась в Smalltalk и впоследствии была принята языком Ruby. Его идея универсальной вложенности (почти каждую конструкцию в Scala можно вложить в любую другую) реализована также в Algol, Simula, а в последнее время в Beta и gbeta. Его принцип единообразного доступа к вызову методов и выбору полей пришел из Eiffel. Его подход к функциональному программированию очень близок по духу к применяемому в семействе языков ML, видными представителями которого являются SML, OCaml и F#. Многие функции высшего порядка в стандартной библиотеке Scala присутствуют также в ML или Haskell. Толчком для появления в Scala неявных параметров стали классы типов языка Haskell — в более классическом объектно-ориентированном окружении они дают аналогичные результаты. Используемая в Scala основная библиотека многопоточного вычисления на основе акторов — Akka — создавалась под сильным влиянием особенностей языка Erlang.

Scala — не первый язык, в котором акцент сделан на масштабируемости и расширяемости. Исторические корни расширяемых языков для различных областей применения обнаруживаются в статье Петера Ландина (Peter Landin) 1966 года The Next 700 Programming Languages («Следующие 700 языков программирования»)[2]. (Описываемый в ней язык Iswim, наряду с Lisp, — один из первых в своем роде функциональных языков.) Происхождение конкретной идеи трактовки инфиксного оператора в качестве функции можно проследить до Iswim и Smalltalk. Еще одна важная идея заключается в разрешении использования функционального литерала (или блока) в качестве параметра, позволяющего библиотекам определять управляющие конструкции. Она также проистекает из Iswim и Smalltalk. И у Smalltalk, и у Lisp довольно гибкий синтаксис, широко применявшийся для создания закрытых предметно-ориентированных языков. Еще один масштабируемый язык, C++, который может быть адаптирован и расширен благодаря применению перегрузки

---

[1] Главное отличие от Java касается синтаксиса для объявления типов: вместо «Тип переменная», как в Java, задействуется форма «переменная: Тип». Используемый в Scala постфиксный синтаксис типа похож на синтаксис, применяемый в Pascal, Modula-2 или Eiffel. Основная причина такого отклонения имеет отношение к логическому выводу типов, зачастую позволяющему опускать тип переменной или тип возвращаемого методом значения. Легче использовать синтаксис «переменная: Тип», поскольку двоеточие и тип можно просто не указывать. Но в стиле языка C, применяющем форму «Тип переменная», просто так не указывать тип нельзя, поскольку при этом исчезнет сам признак начала определения. Неуказанный тип в качестве заполнителя требует какое-нибудь ключевое слово (C# 3.0, в котором имеется логический вывод типов, для этой цели задействует ключевое слово var). Такое альтернативное ключевое слово представляется несколько более надуманным и менее привычным, чем подход, который используется в Scala.

[2] *Landin P. J.* The Next 700 Programming Languages // Communications of the ACM, 1966. — № 9 (3). — P. 157–166.

операторов и своей системе шаблонов, построен, по сравнению со Scala, на низкоуровневой и более системно-ориентированной основе. Кроме того, Scala — не первый язык, объединяющий в себе функциональное и объектно-ориентированное программирование, хотя, вероятно, в этом направлении продвинулся гораздо дальше прочих. К числу других языков, объединивших некоторые элементы функционального программирования с объектно-ориентированным, относятся Ruby, Smalltalk и Python. Расширения Java-подобного ядра некоторыми функциональными идеями были предприняты на Java-платформе в Pizza, Nice, Multi-Java и самом Java 8. Существуют также изначально функциональные языки, которые приобрели систему объектов. В качестве примера можно привести OCaml, F# и PLT-Scheme.

В Scala применяются также некоторые нововведения в области языков программирования. Например, его абстрактные типы — более объектно-ориентированная альтернатива обобщенным типам, его трейты позволяют выполнять гибкую сборку компонентов, а экстракторы обеспечивают независимый от представления способ сопоставления с образцом. Эти нововведения были представлены в статьях на конференциях по языкам программирования в последние годы[1].

Резюме

Ознакомившись с текущей главой, вы получили некоторое представление о том, что представляет собой Scala и как он может помочь программисту в работе. Разумеется, этот язык не решит все ваши проблемы и не увеличит волшебным образом вашу личную продуктивность. Следует заранее предупредить, что Scala нужно применять искусно, а для этого потребуется получить некоторые знания и практические навыки. Если вы перешли к Scala от языка Java, то одними из наиболее сложных аспектов его изучения для вас могут стать система типов Scala, которая существенно богаче, чем у Java, и его поддержка функционального стиля программирования. Цель данной книги — послужить руководством при поэтапном, от простого к сложному, изучении особенностей Scala. Полагаем, что вы приобретете весьма полезный интеллектуальный опыт, расширяющий ваш кругозор и изменяющий взгляд на проектирование программных средств. Надеемся, что вдобавок вы получите от программирования на Scala истинное удовольствие и познаете творческое вдохновение.

В следующей главе вы приступите к написанию кода Scala.

---

[1] Для получения большей информации см.: *Odersky. M., Cremet V., Röckl C., Zenger M.* A Nominal Theory of Objects with Dependent Types // In Proc. ECOOP'03, Springer LNCS. 2003. July. — P. 201–225; *Odersky M., Zenger M.* Scalable Component Abstractions // Proceedings of OOPSLA. 2005. October. — P. 41–58; *Emir B., Odersky M., Williams J.* Matching Objects With Patterns // Proc. ECOOP, Springer LNCS. 2007. July. — P. 273–295.

2 Первые шаги в Scala

Пришло время написать какой-нибудь код на Scala. Прежде чем углубиться в руководство по этому языку, мы приведем две обзорные главы по нему и, что наиболее важно, заставим вас приступить к написанию кода. Рекомендуем по мере освоения материала на практике проверить работу всех примеров кода, представленных в этой и последующей главах. Лучше всего приступить к изучению Scala, программируя на данном языке.

Запуск представленных далее примеров возможен с помощью стандартной установки Scala. Получить ее можно по адресу www.scala-lang.org/downloads, следуя инструкциям для вашей платформы. Кроме того, вы можете воспользоваться дополнительным модулем Scala для Eclipse, IntelliJ или NetBeans. Применительно к шагам, рассматриваемым в данной главе, предполагаем, что вы обратились к дистрибутиву Scala из scala-lang.org.[1]

Если вы опытный программист, но новичок в Scala, то внимательно прочитайте следующие две главы: в них приводится достаточный объем информации, позволяющий приступить к написанию полезных программ на этом языке. Если же опыт программирования у вас невелик, то часть материалов может показаться чем-то загадочным. Однако не стоит переживать. Чтобы ускорить процесс изучения, нам пришлось обойтись без некоторых подробностей. Более обстоятельные пояснения мы предоставим в последующих главах. Кроме того, в следующих двух главах дадим ряд сносок с указанием разделов книги, в которых можно найти более подробные объяснения.

2.1. Шаг 1. Осваиваем интерпретатор Scala

Проще всего приступить к изучению Scala, задействуя Scala-интерпретатор — интерактивную оболочку для написания выражений и программ на этом языке. Интерпретатор, который называется scala, будет вычислять набираемые вами

[1] Приводимые в книге примеры тестировались с применением Scala версии 2.13.1.

выражения и выводить на экран значение результата. Чтобы воспользоваться интерпретатором, нужно в приглашении командной строки набрать команду `scala`[1]:

```
$ scala
Welcome to Scala version 2.13.1
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

После того как вы наберете выражение, например, `1 + 2` и нажмете клавишу Enter:

```
scala> 1 + 2
```

интерпретатор выведет на экран:

```
res0: Int = 3
```

Эта строка включает:

- автоматически сгенерированное или определенное пользователем имя для ссылки на вычисленное значение (`res0`, означающее результат 0);
- двоеточие (`:`), за которым следует тип выражения (`Int`);
- знак равенства (`=`);
- значение, полученное в результате вычисления выражения (`3`).

Тип `Int` означает класс `Int` в пакете `scala`. Пакеты в Scala аналогичны пакетам в Java — они разбивают глобальное пространство имен на части и предоставляют механизм для сокрытия данных[2]. Значения класса `Int` соответствуют `int`-значениям в Java. Если говорить в общем, то все примитивные типы Java имеют соответствующие классы в пакете `scala`. Например, `scala.Boolean` соответствует Java-типу `boolean`. А `scala.Float` соответствует Java-типу `float`. И при компиляции вашего кода Scala в байт-код Java компилятор Scala будет по возможности использовать примитивные типы Java, чтобы обеспечить вам преимущество в производительности при работе с примитивными типами.

Идентификатор `resX` может использоваться в последующих строках. Например, поскольку ранее для `res0` было установлено значение `3`, то результат выражения `res0 * 3` будет равен `9`:

```
scala> res0 * 3
res1: Int = 9
```

Чтобы вывести на экран необходимое, но недостаточно информативное приветствие `Hello, world!`, наберите следующую команду:

```
scala> println("Hello, world!")
Hello, world!
```

---

[1] Если вы используете Windows, то команду scala нужно набирать в окне командной строки.

[2] Если вы не знакомы с пакетами Java, то их можно рассматривать как средство предоставления классам полных имен. Поскольку Int входит в пакет scala, Int — простое имя класса, а scala.Int — полное. Более подробно пакеты рассматриваются в главе 13.

Функция `println` выводит на стандартное устройство вывода переданную ей строку, подобно тому как это делает `System.out.println` в Java.

2.2. Шаг 2. Объявляем переменные

В Scala имеется две разновидности переменных: `val`-переменные и `var`-переменные. Первые аналогичны финальным переменным в Java. После инициализации `val`-переменная уже никогда не может быть присвоена повторно. В отличие от нее `var`-переменная аналогична нефинальной переменной в Java и может быть присвоена повторно в течение своего жизненного цикла. Определение `val`-переменной выглядит так:

```
scala> val msg = "Hello, world!"
msg: String = Hello, world!
```

Эта инструкция вводит в употребление переменную `msg` в качестве имени для строки `"Hello, world!"`. Типом `msg` является `java.lang.String`, поскольку строки в Scala реализуются Java-классом `String`.

Если вы привыкли объявлять переменные в Java, то в этом примере кода можете заметить одно существенное отличие: в `val`-определении нигде не фигурируют ни `java.lang.String`, ни `String`. Пример демонстрирует *логический вывод типов*, то есть возможность Scala определять неуказанные типы. В данном случае, поскольку вы инициализировали `msg` строковым литералом, Scala придет к выводу, что типом `msg` должен быть `String`. Когда интерпретатор (или компилятор) Scala хочет выполнить вывод типов, зачастую лучше всего будет позволить ему сделать это, не засоряя код ненужными явными аннотациями типов. Но при желании можете указать тип явно, и, вероятно, иногда это придется делать. Явная аннотация типа может не только гарантировать, что компилятор Scala выведет желаемый тип, но и послужить полезной документаций для тех, кто впоследствии станет читать ваш код. В отличие от Java, где тип переменной указывается перед ее именем, в Scala вы указываете тип переменной после ее имени, отделяя его двоеточием, например:

```
scala> val msg2: java.lang.String = "Hello again, world!"
msg2: String = Hello again, world!
```

Или же, поскольку типы `java.lang` вполне опознаваемы в программах на Scala по их простым именам[1], запись можно упростить:

```
scala> val msg3: String = "Hello yet again, world!"
msg3: String = Hello yet again, world!
```

Возвратимся к исходной переменной `msg`. Поскольку она определена, то ею можно воспользоваться в соответствии с вашими ожиданиями, например:

```
scala> println(msg)
Hello, world!
```

[1] Простым именем java.lang.String является String.

Учитывая, что `msg` является `val`-, а не `var`-переменной, вы не сможете повторно присвоить ей другое значение[1]. Посмотрите, к примеру, как интерпретатор выражает свое недовольство при попытке сделать следующее:

```
scala> msg = "Goodbye cruel world!"
       ^
       error: reassignment to val
```

При необходимости выполнить повторное присваивание следует воспользоваться `var`-переменной:

```
scala> var greeting = "Hello, world!"
greeting: String = Hello, world!
```

Как только приветствие станет `var`-, а не `val`-переменной, ему можно будет присвоить другое значение. Если, к примеру, чуть позже вы станете более раздражительным, то можете поменять приветствие на просьбу оставить вас в покое:

```
scala> greeting = "Leave me alone, world!"
mutated greeting
```

Чтобы ввести в интерпретатор код, который не помещается в одну строку, просто продолжайте набирать код после заполнения первой строки. Если набор кода еще не завершен, то интерпретатор отреагирует установкой на следующей строке вертикальной черты:

```
scala> val multiLine =
     |    "This is the next line."
multiLine: String = This is the next line.
```

Если поймете, что набрали не то, а интерпретатор все еще ждет ввода дополнительного текста, то можете сделать отмену, дважды нажав клавишу Enter:

```
scala> val oops =
     |
     |
You typed two blank lines. Starting a new command.

scala>
```

Далее в книге символы вертикальной черты отображаться не будут, чтобы код читался легче и его было проще скопировать и вставить в интерпретатор из электронной версии в формате PDF.

2.3. Шаг 3. Определяем функции

После работы с переменными в Scala вам, вероятно, захотелось написать какие-нибудь функции. В данном языке это делается так:

---

[1] Но в интерпретаторе новую val-переменную можно определить с именем, которое до этого уже использовалось. Этот механизм рассматривается в разделе 7.7.

```
scala> def max(x: Int, y: Int): Int = {
         if (x > y) x
         else y
       }
max: (x: Int, y: Int)Int
```

Определение функции начинается с ключевого слова `def`. После имени функции, в данном случае `max`, стоит заключенный в круглые скобки перечень параметров, разделенных запятыми. За каждым параметром функции должна следовать аннотация типа, перед которой ставится двоеточие, поскольку компилятор Scala (и интерпретатор, но с этого момента будет упоминаться только компилятор) не выводит типы параметров функции. В данном примере функция по имени `max` получает два параметра, `x` и `y`, и оба они относятся к типу `Int`. После закрывающей круглой скобки перечня параметров функции `max` обнаруживается аннотация типа `: Int`. Она определяет *результирующий тип* самой функции `max`[1]. За ним стоят знак равенства и пара фигурных скобок, внутри которых содержится тело функции. В данном случае в теле содержится одно выражение `if`, с помощью которого в качестве результата выполнения функции `max` выбирается либо `x`, либо `y` в зависимости от того, значение какой из переменных больше. Как здесь показано, выражение `if` в Scala может вычисляться в значение, подобное тому, что вычисляется тернарным оператором Java. Например, Scala-выражение `if (x > y) x else y` вычисляется точно так же, как выражение `(x > y) ? x : y` в Java. Знак равенства, предшествующий телу функции, дает понять, что с точки зрения функционального мира функция определяет выражение, результатом вычисления которого становится значение. Основная структура функции показана на рис. 2.1.

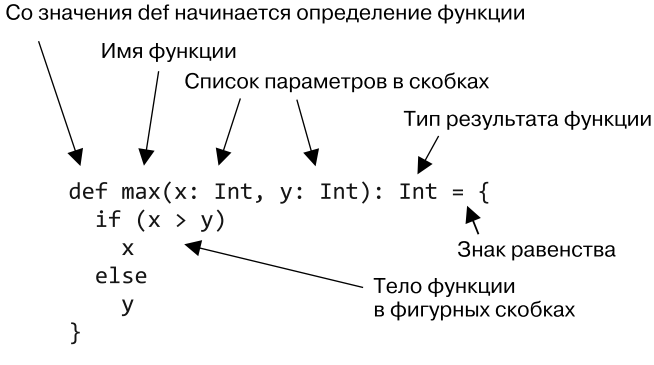


Рис. 2.1. Основная форма определения функции в Scala

Иногда компилятор Scala может потребовать от вас указать результирующий тип функции. Если, к примеру, функция является *рекурсивной*[2], то вы должны

[1] В Java тип возвращаемого из метода значения является возвращаемым типом. В Scala то же самое понятие называется результирующим типом.

[2] Функция называется рекурсивной, если вызывает саму себя.

указать ее результирующий тип явно. Но в случае с функцией `max` вы можете не указывать результирующий тип функции — компилятор выведет его самостоятельно[1]. Кроме того, если функция состоит только из одной инструкции, то при желании фигурные скобки можно не ставить. Следовательно, альтернативный вариант функции `max` может быть таким:

```
scala> def max(x: Int, y: Int) = if (x > y) x else y
max: (x: Int, y: Int)Int
```

После определения функции ее можно вызвать по имени:

```
scala> max(3, 5)
res4: Int = 5
```

А вот определение функции, которая не получает параметры и не возвращает полезный результат:

```
scala> def greet() = println("Hello, world!")
greet: ()Unit
```

Когда определяется функция приветствия `greet()`, интерпретатор откликается следующим приветствием: `()Unit`. Разумеется, слово `greet` — это имя функции. Пустота в скобках показывает, что функция не получает параметров. А `Unit` — результирующий тип функции `greet`. Он показывает, что функция не возвращает никакого интересного значения. Тип `Unit` в Scala подобен типу `void` в Java. Фактически каждый метод, возвращающий `void` в Java, отображается на метод, возвращающий `Unit` в Scala. Таким образом, методы с результирующим типом `Unit` выполняются только для того, чтобы проявились их побочные эффекты. В случае с `greet()` побочным эффектом будет дружеское приветствие, выведенное на стандартное устройство вывода.

При выполнении следующего шага код Scala будет помещен в файл и запущен в качестве скрипта. Если нужно выйти из интерпретатора, то это можно сделать с помощью команды `:quit` или `:q`:

```
scala> :quit
$
```

2.4. Шаг 4. Пишем Scala-скрипты

Несмотря на то что язык Scala разработан, чтобы помочь программистам создавать очень большие масштабируемые системы, он вполне может подойти и для решения менее масштабных задач наподобие написания скриптов. Скрипт представляет собой простую последовательность инструкций, размещенных в файле и выполняемых друг за другом. Поместите в файл по имени `hello.scala` следующий код:

[1] Тем не менее зачастую есть смысл указывать результирующий тип явно, даже когда компилятор этого не требует. Такая аннотация типа может упростить чтение кода, поскольку читателю не придется изучать тело функции, чтобы определить, каким будет вывод результирующего типа.

```
println("Hello, world, from a script!")
```

а затем запустите файл на выполнение[1]:

```
$ scala hello.scala
```

И вы получите еще одно приветствие:

```
Hello, world, from a script!
```

Аргументы командной строки, указанные для скрипта Scala, можно получить из Scala-массива по имени `args`. В Scala индексация элементов массива начинается с нуля и обращение к элементу выполняется указанием индекса в круглых скобках. Следовательно, первым элементом в Scala-массиве по имени `steps` будет `steps(0)`, а не `steps[0]`, как в Java. Убедиться в этом на практике можно, создав новый файл по имени `helloarg.scala`:

```
// Поприветствуйте содержимое первого аргумента
println("Hello, " + args(0) + "!")
```

а затем запустив его на выполнение:

```
$ scala helloarg.scala planet
```

В данной команде `planet` передается в качестве аргумента командной строки, доступного в скрипте при использовании выражения `args(0)`. Поэтому вы должны увидеть на экране следующий текст:

```
Hello, planet!
```

Обратите внимание на наличие комментария в скрипте. Компилятор Scala проигнорирует символы между парой символов `//` и концом строки, а также все символы между сочетаниями символов `/*` и `*/`. Вдобавок в этом примере показана конкатенация `String`-значений, выполненная с помощью оператора `+`. Весь код работает вполне предсказуемо. Выражение `"Hello, " + "world!"` будет вычислено в строку `"Hello, world!"`.

2.5. Шаг 5. Организуем цикл с while и принимаем решение с if

Чтобы попробовать в работе конструкцию `while`, наберите следующий код и сохраните его в файле `printargs.scala`:

```
var i = 0
while (i < args.length) {
  println(args(i))
  i += 1
}
```

[1] В Unix и Windows скрипты можно запускать, не набирая слово scala, а используя синтаксис «решетка — восклицательный знак», показанный в приложении.

ПРИМЕЧАНИЕ

Хотя примеры в данном разделе помогают объяснить суть циклов while, они не демонстрируют наилучший стиль программирования на Scala. В следующем разделе будут показаны более рациональные подходы, которые позволяют избежать перебора элементов массива с помощью индексов.

Этот скрипт начинается с определения переменой, `var i = 0`. Вывод типов относит переменную `i` к типу `scala.Int`, поскольку это тип ее начального значения `0`. Конструкция `while` на следующей строке заставляет *блок* (код между фигурными скобками) повторно выполняться, пока булево выражение `i < args.length` будет вычисляться в `false`. Метод `args.length` вычисляет длину массива `args`. Блок содержит две инструкции, каждая из которых набрана с отступом в два пробела, что является рекомендуемым стилем отступов для кода на Scala. Первая инструкция, `println(args(i))`, выводит на экран `i`-й аргумент командной строки. Вторая, `i += 1`, увеличивает значение переменной `i` на единицу. Обратите внимание: Java-код `++i` и `i++` в Scala не работает. Чтобы в Scala увеличить значение переменной на единицу, нужно использовать одно из двух выражений: либо `i = i + 1`, либо `i += 1`. Запустите этот скрипт с помощью команды, показанной ниже:

```
$ scala printargs.scala Scala is fun
```

И вы увидите:

```
Scala
is
fun
```

Чтобы продолжить развлекаться, наберите в новом файле по имени `echoargs.scala` следующий код:

```
var i = 0
while (i < args.length) {
  if (i != 0)
    print(" ")
  print(args(i))
  i += 1
}
println()
```

В целях вывода всех аргументов в одной и той же строке в этой версии вместо вызова `println` используется вызов `print`. Чтобы эту строку можно было прочитать, перед каждым аргументом, за исключением первого, благодаря использованию конструкции `if (i != 0)` вставляется пробел. При первом проходе цикла `while` выражение `i != 0` станет вычисляться в `false`, поэтому перед начальным элементом пробел выводиться не будет. В самом конце добавлена еще одна инструкция `println`, чтобы после вывода аргументов произошел переход на новую строку. Тогда у вас получится очень красивая картинка. Если запустить этот скрипт с помощью команды:

```
$ scala echoargs.scala Scala is even more fun
```

то вы увидите на экране такой текст:

```
Scala is even more fun
```

Обратите внимание: в Scala, как и в Java, булевы выражения для `while` или `if` нужно помещать в круглые скобки. (Иными словами, вы не можете в Scala прибегнуть к такому выражению, как `if i < 10`, вполне допустимому в языках, подобных Ruby. В Scala нужно воспользоваться записью `if (i < 10)`.) Еще один прием программирования, аналогичный применяемому в Java, заключается в том, что если в блоке `if` используется только одна инструкция, то при желании фигурные скобки можно не ставить, как показано в конструкции `if` в файле `echoargs.scala`. Но хотя их там нет, в Scala, как и в Java, для отделения инструкций друг от друга используются точки с запятыми (стоит уточнить: в Scala точки с запятыми зачастую необязательны, что снижает нагрузку на правый мизинец при наборе текста). Но если вы склонны к многословию, то можете записать содержимое файла `echoargs.scala` в следующем виде:

```
var i = 0;
while (i < args.length) {
  if (i != 0) {
    print(" ");
  }
  print(args(i));
  i += 1;
}
println();
```

2.6. Шаг 6. Перебираем элементы с foreach и for

Возможно, при написании циклов `while` на предыдущем шаге вы даже не осознавали того, что программирование велось в *императивном* стиле. Обычно он применяется с такими языками, как Java, C++ и C. При работе в этом стиле императивные команды в случае последовательного перебора элементов в цикле выдаются поочередно и зачастую изменяемое состояние совместно используется различными функциями. Scala позволяет программировать в императивном стиле, но, узнав этот язык получше, вы, скорее всего, перейдете преимущественно на *функциональный* стиль. По сути, одна из основных целей этой книги — помочь освоить работу в функциональном стиле, чтобы она стала такой же комфортной, как и работа в императивном.

Одна из основных характеристик функционального языка — то, что его функции относятся к конструкциям первого класса, и это абсолютно справедливо для языка Scala. Например, еще один, гораздо более лаконичный способ вывода каждого аргумента командной строки выглядит так:

```
args.foreach(arg => println(arg))
```

В этом коде в отношении массива `args` вызывается метод `foreach`, в который передается функция. В данном случае передается *функциональный литерал* с одним параметром `arg`. Тело функции — вызов `println(arg)`. Если набрать показанный

ранее код в новом файле по имени `pa.scala` и запустить этот файл на выполнение с помощью команды:

```
$ scala pa.scala Concise is nice
```

то на экране появятся строки:

```
Concise
is
nice
```

В предыдущем примере интерпретатор Scala вывел тип `arg`, причислив эту переменную к `String`, поскольку `String` — тип элемента массива, в отношении которого вызван метод `foreach`. Если вы предпочитаете конкретизировать, то можете упомянуть название типа. Но, пойдя по этому пути, придется часть кода, в которой указывается переменная аргумента, заключать в круглые скобки (это и есть обычный синтаксис):

```
args.foreach((arg: String) => println(arg))
```

При запуске этот скрипт ведет себя точно так же, как и предыдущий.

Если же вы склонны не к конкретизации, а к более лаконичному изложению кода, то можете воспользоваться специальными сокращениями, принятыми в Scala. Если функциональный литерал функции состоит из одной инструкции, принимающей один аргумент, то обозначать данный аргумент явным образом по имени не нужно[1]. Поэтому работать будет и следующий код:

```
args.foreach(println)
```

Резюмируем усвоенное: синтаксис для функционального литерала представляет собой список поименованных параметров, заключенный в круглые скобки, а также правую стрелку, за которой следует тело функции. Этот синтаксис показан на рис. 2.2.

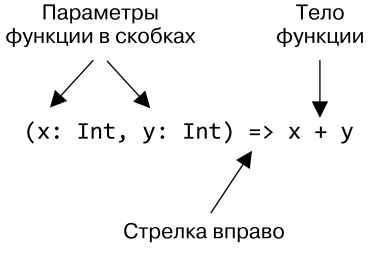


Рис. 2.2. Синтаксис функционального литерала в Scala

Теперь вы можете поинтересоваться: что же случилось с теми проверенными циклами `for`, которые вы привыкли использовать в таких императивных языках,

[1] Это сокращение, называемое частично примененной функцией, рассматривается в разделе 8.6.

как Java или C? Придерживаться функционального направления в Scala возможно с помощью только одного функционального родственника императивной конструкции `for`, который называется *выражением `for`*. Так как вы не сможете понять всю его эффективность и выразительность, пока не доберетесь до раздела 7.3 (или не заглянете в него), здесь о нем будет дано лишь общее представление. Наберите в новом файле по имени `forargs.scala` следующий код:

```
for (arg <- args)
  println(arg)
```

В круглых скобках после `for` содержится выражение `arg <- args`[1]. Справа от обозначения `<-` указан уже знакомый вам массив `args`. Слева от `<-` указана переменная `arg`, относящаяся к `val`-, а не к `var`-переменным. (Так как она всегда относится к `val`-переменным, записывается только `arg`, а не `val arg`.) Может показаться, что `arg` относится к `var`-переменной, поскольку она будет получать новое значение при каждой итерации, однако в действительности она относится к `val`-переменной: `arg` не может получить новое значение внутри тела выражения. Вместо этого для каждого элемента массива `args` будет создана новая `val`-переменная по имени `arg`, которая будет инициализирована значением элемента, и тело `for` будет выполнено.

Если скрипт `forargs.scala` запустить с помощью команды:

```
$ scala forargs.scala for arg in args
```

то вы увидите:

```
for
arg
in
args
```

Диапазон применения используемого в Scala выражения `for` значительно шире, чем показано здесь, но для начала достаточно и этого примера. Дополнительные возможности `for` будут показаны в разделе 7.3 и в главе 23.

Резюме

В данной главе мы привели основную информацию о Scala. Надеемся, вы воспользовались возможностью создать код на этом языке. В следующей главе мы продолжим вводный обзор и рассмотрим более сложные темы.

[1] Обозначение <– можно трактовать как «в». То есть код for (arg <– args) может быть прочитан как «для arg в args».

3 Дальнейшие шаги в Scala

В этой главе продолжается введение в Scala, начатое в предыдущей главе. Здесь мы рассмотрим более сложные функциональные возможности. Когда вы усвоите материал главы, у вас будет достаточно знаний, чтобы начать создавать полезные скрипты на Scala. Мы вновь рекомендуем по мере чтения текста получать практические навыки с помощью приводимых примеров. Лучше всего осваивать Scala, начиная создавать код на данном языке.

3.1. Шаг 7. Параметризуем массивы типами

В Scala создавать объекты или экземпляры класса можно с помощью ключевого слова `new`. При создании объекта в Scala вы можете *параметризовать* его значениями и типами. Параметризация означает конфигурирование экземпляра при его создании. Параметризация экземпляра значениями производится путем передачи конструктору объектов в круглых скобках. Например, код Scala, показанный ниже, создает новый объект `java.math.BigInteger`, выполняя его параметризацию значением `"12345"`:

```
val big = new java.math.BigInteger("12345")
```

Параметризация экземпляра типами выполняется с помощью указания одного или нескольких типов в квадратных скобках. Пример показан в листинге 3.1. Здесь `greetStrings` — значение типа `Array[String]` («массив строк»), инициализируемое длиной `3` путем его параметризации значением `3` в первой строке кода. Если запустить код в листинге 3.1 в качестве скрипта, то вы увидите еще одно приветствие `Hello, world!`. Учтите, что при параметризации экземпляра как типом, так и значением тип стоит первым и указывается в квадратных скобках, а за ним следует значение в круглых скобках.

Листинг 3.1. Параметризация массива типом

```
val greetStrings = new Array[String](3)

greetStrings(0) = "Hello"
```

```
greetStrings(1) = ", "
greetStrings(2) = "world!\n"

for (i <- 0 to 2)
  print(greetStrings(i))
```

ПРИМЕЧАНИЕ

Хотя код в листинге 3.1 содержит важные понятия, он не показывает рекомендуемый способ создания и инициализации массива в Scala. Более рациональный способ будет показан в листинге 3.2.

Если вы склонны делать более явные указания, то тип `greetStrings` можно обозначить так:

```
val greetStrings: Array[String] = new Array[String](3)
```

С учетом имеющегося в Scala вывода типов эта строка кода семантически эквивалентна первой строке листинга 3.1. Но в данной форме показано следующее: часть параметризации, которая относится к типу (название типа в квадратных скобках), формирует часть типа экземпляра, однако часть параметризации, относящаяся к значению (значения в круглых скобках), в формировании не участвует. Типом `greetStrings` является `Array[String]`, а не `Array[String](3)`.

В следующих трех строках кода в листинге 3.1 инициализируется каждый элемент массива `greetStrings`:

```
greetStrings(0) = "Hello"
greetStrings(1) = ", "
greetStrings(2) = "world!\n"
```

Как уже упоминалось, доступ к массивам в Scala осуществляется за счет помещения индекса элемента в круглые, а не в квадратные скобки, как в Java. Следовательно, нулевым элементом массива будет `greetStrings(0)`, а не `greetStrings[0]`.

Эти три строки кода иллюстрируют важное понятие, помогающее осмыслить значение для Scala `val`-переменных. Когда переменная определяется с помощью `val`, повторно присвоить значение данной переменной нельзя, но объект, на который она ссылается, потенциально может быть изменен. Следовательно, в данном случае присвоить `greetStrings` значение другого массива невозможно — переменная `greetStrings` всегда будет указывать на один и тот же экземпляр типа `Array[String]`, которым она была инициализирована. Но впоследствии в элементы типа `Array[String]` можно вносить изменения, следовательно, сам массив является изменяемым.

Последние две строки листинга 3.1 содержат выражение `for`, которое поочередно выводит каждый элемент массива `greetStrings`:

```
for (i <- 0 to 2)
  print(greetStrings(i))
```

В первой строке кода для этого выражения `for` показано еще одно общее правило Scala: если метод получает лишь один параметр, то его можно вызвать

без точки или круглых скобок. В данном примере `to` на самом деле является методом, получающим один `Int`-аргумент. Код `0 to 2` преобразуется в вызов метода `(0).to(2)`[1]. Следует заметить, что этот синтаксис работает только при явном указании получателя вызова метода. Код `println 10` использовать нельзя, а код `Console println 10` — можно.

С технической точки зрения в Scala нет перегрузки операторов, поскольку в нем фактически отсутствуют операторы в традиционном понимании. Вместо этого такие символы, как `+`, `-`, `*`, `/`, могут использоваться в качестве имен методов. Следовательно, когда при выполнении шага 1 вы набираете в интерпретаторе Scala код `1 + 2`, в действительности вы вызываете метод по имени `+` в отношении `Int`-объекта `1`, передавая ему в качестве параметра значение `2`. Как показано на рис. 3.1, вместо этого `1 + 2` можно записать с помощью традиционного синтаксиса вызова метода: `(1).+(2)`.

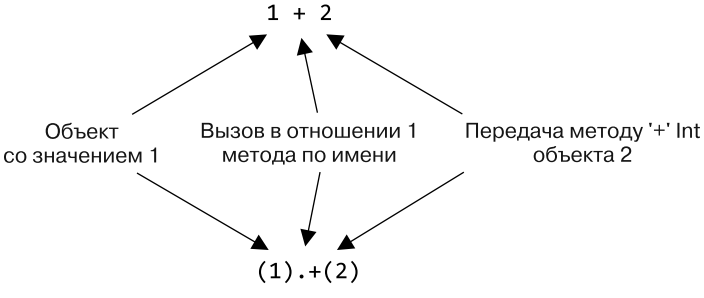


Рис. 3.1. Все операции в Scala являются вызовами методов

Еще одна весьма важная идея, проиллюстрированная в этом примере, поможет понять, почему доступ к элементам массивов Scala осуществляется с помощью круглых скобок. В Scala меньше особых случаев по сравнению с Java. Массивы в Scala, как и в случае с любыми другими классами, — просто экземпляры классов. При использовании круглых скобок, окружающих одно или несколько значений переменной, Scala преобразует код в вызов метода по имени `apply` применительно к данной переменной. Следовательно, код `greetStrings(i)` преобразуется в код `greetStrings.apply(i)`. Получается, что элемент массива в Scala является просто вызовом обычного метода, ничем не отличающегося от любого своего собрата. Этот принцип не ограничивается массивами: любое использование объекта в отношении каких-либо аргументов в круглых скобках будет преобразовано в вызов метода `apply`. Разумеется, данный код будет скомпилирован, только если в этом типе объекта определен метод `apply`. То есть это не особый случай, а общее правило.

[1] Этот метод to фактически возвращает не массив, а иную разновидность последовательности, содержащую значения 0, 1 и 2, последовательный перебор которых выполняется выражением for. Последовательности и другие коллекции будут рассматриваться в главе 17.

По аналогии с этим, когда присваивание выполняется в отношении переменной, к которой применены круглые скобки с одним или несколькими аргументами внутри, компилятор выполнит преобразование в вызов метода `update`, получающего не только аргументы в круглых скобках, но и объект, расположенный справа от знака равенства. Например, код:

```
greetStrings(0) = "Hello"
```

будет преобразован в код:

```
greetStrings.update(0, "Hello")
```

Таким образом, следующий код семантически эквивалентен коду листинга 3.1:

```
val greetStrings = new Array[String](3)

greetStrings.update(0, "Hello")
greetStrings.update(1, ", ")
greetStrings.update(2, "world!\n")

for (i <- 0.to(2))
  print(greetStrings.apply(i))
```

Концептуальная простота в Scala достигается за счет того, что все — от массивов до выражений — рассматривается как объекты с методами. Вам не нужно запоминать особые случаи, например такие, как существующее в Java различие между примитивными типами и соответствующими им типами-оболочками или между массивами и обычными объектами. Более того, подобное единообразие не вызывает больших потерь производительности. Компилятор Scala везде, где только возможно, использует в скомпилированном коде массивы Java, элементарные типы и чистые арифметические операции.

Рассмотренные до сих пор в этом шаге примеры компилируются и выполняются весьма неплохо, однако в Scala имеется более лаконичный способ создания и инициализации массивов, который, как правило, вы и будете использовать (см. листинг 3.2). Данный код создает новый массив длиной три элемента, инициализируемый переданными строками `"zero"`, `"one"` и `"two"`. Компилятор выводит тип массива как `Array[String]`, поскольку ему передаются строки.

Листинг 3.2. Создание и инициализация массива

```
val numNames = Array("zero", "one", "two")
```

Фактически в листинге 3.2 вызывается фабричный метод по имени `apply`, создающий и возвращающий новый массив. Метод `apply` получает переменное количество аргументов[1] и определяется в *объекте-компаньоне* `Array`. Подробнее объекты-компаньоны будут рассматриваться в разделе 4.3. Если вам приходилось программировать на Java, то можете воспринимать это как вызов статического

[1] Списки аргументов переменной длины или повторяемые параметры рассматриваются в разделе 8.8.

метода по имени `apply` в отношении класса `Array`. Менее лаконичный способ вызова того же метода `apply` выглядит следующим образом:

```
val numNames2 = Array.apply("zero", "one", "two")
```

3.2. Шаг 8. Используем списки

Одна из превосходных отличительных черт функционального стиля программирования — полное отсутствие у методов побочных эффектов. Единственным действием метода должно быть вычисление и возвращение значения. Получаемые в результате применения такого подхода преимущества заключаются в том, что методы становятся менее запутанными, и это упрощает их чтение и повторное использование. Есть и еще одно преимущество (в статически типизированных языках): все попадающее в метод и выходящее за его пределы проходит проверку на принадлежность к определенному типу, поэтому логические ошибки, скорее всего, проявятся сами по себе в виде ошибок типов. Применять данную функциональную философию к миру объектов означает превратить эти объекты в неизменяемые.

Как вы уже видели, массив Scala — неизменяемая последовательность объектов с общим типом. Тип `Array[String]`, к примеру, содержит только строки. Изменить длину массива после создания его экземпляра невозможно, но вы можете изменять значения его элементов. Таким образом, массивы относятся к изменяемым объектам.

Для неизменяемой последовательности объектов с общим типом можно воспользоваться списком, определяемым Scala-классом `List`. Как и в случае применения массивов, в типе `List[String]` содержатся только строки. Список Scala, `scala.List`, отличается от Java-типа `java.util.List` тем, что списки Scala всегда неизменямые, а списки Java могут изменяться. В более общем смысле список Scala разработан с прицелом на использование функционального стиля программирования. Список создается очень просто, и листинг 3.3 как раз показывает это.

Листинг 3.3. Создание и инициализация списка

```
val oneTwoThree = List(1, 2, 3)
```

Код в листинге 3.3 создает новую `val`-переменную по имени `oneTwoThree`, инициализируемую новым списком `List[Int]` с целочисленными элементами `1`, `2` и `3`[1]. Из-за своей неизменяемости списки ведут себя подобно строкам в Java: при вызове метода в отношении списка из-за имени данного метода может создаваться впечатление, что обрабатываемый список будет изменен, но вместо этого создается и возвращается новый список с новым значением. Например, в `List` для объединения списков имеется метод, обозначаемый как `:::`. Используется он следующим образом:

---

[1] Использовать запись new List не нужно, поскольку List.apply() определен в объекте-компаньоне scala.List как фабричный метод. Более подробно объекты-компаньоны рассматриваются в разделе 4.3.

```
val oneTwo = List(1, 2)
val threeFour = List(3, 4)
val oneTwoThreeFour = oneTwo ::: threeFour
println(oneTwo + " и " + threeFour + " не изменились.")
println("Следовательно, " + oneTwoThreeFour + " является новым списком.")
```

Запустив этот скрипт на выполнение, вы увидите следующую картину:

```
List(1, 2) и List(3, 4) не изменились.
Следовательно, List(1, 2, 3, 4) является новым списком.
```

Возможно, со списками чаще всего будет использоваться оператор `::`, который произносится как «конс». Он добавляет в начало существующего списка новый элемент и возвращает список, который получился в результате. Например, если запустить на выполнение следующий скрипт:

```
val twoThree = List(2, 3)
val oneTwoThree = 1 :: twoThree
println(oneTwoThree)
```

то вы увидите:

```
List(1, 2, 3)
```

ПРИМЕЧАНИЕ

В выражении 1 :: twoThree метод :: является методом его правого операнда — списка twoThree. Можно заподозрить, будто с ассоциативностью метода :: что-то не то, но есть простое мнемоническое правило: если метод используется в виде оператора, например a * b, то вызывается в отношении левого операнда, как в выражении a.*(b), если только имя метода не заканчивается двоеточием. А если оно заканчивается двоеточием, то метод вызывается в отношении правого операнда. Поэтому в выражении 1 :: twoThree метод :: вызывается в отношении twoThree с передачей ему 1, то есть: twoThree.::(1). Ассоциативность операторов более подробно будет рассматриваться в разделе 5.9.

Исходя из того, что короче всего указать пустой список с помощью `Nil`, один из способов инициализировать новые списки — связать элементы с помощью `cons`-оператора с `Nil` в качестве последнего элемента[1]. Например, следующий скрипт создаст ту же картину на выходе, что и предыдущий `List(1, 2, 3)`:

```
val oneTwoThree = 1 :: 2 :: 3 :: Nil
println(oneTwoThree)
```

Имеющийся в Scala класс `List` укомплектован весьма полезными методами, многие из которых показаны в табл. 3.1. Вся эффективность списков будет раскрыта в главе 16.

[1] Причина, по которой в конце списка нужен Nil, заключается в том, что метод :: определен в классе List. Если попытаться просто воспользоваться кодом 1 :: 2 :: 3, то он не пройдет компиляцию, поскольку 3 относится к типу Int, у которого нет метода ::.

Почему со списками не следует использовать операцию добавления в конец (append)?

В классе `List` есть операция добавления, которая записывается как `:+` (о ней говорится в главе 24), но используется очень редко. Так происходит потому, что время, которое она тратит на добавление к списку, растет линейно с размером списка, а на добавление в начало списка с помощью метода `::` всегда затрачивается одно и то же время. Если нужно добиться эффективности при создании списка с помощью добавления элементов, то можно добавлять их в начало, а завершив добавление, вызвать метод реверсирования `reverse`. Или же можно воспользоваться изменяемым списком `ListBuffer`, предлагающим операцию добавления, а затем, завершив добавление, вызвать метод `toList` и преобразовать его в обычный список. Список типа `ListBuffer` будет рассмотрен в разделе 22.2.

Таблица 3.1. Некоторые методы класса List и их использование

Что используется	Что этот метод делает
List() или Nil	Создает пустой список List
List("Cool", "tools", "rule")	Создает новый список типа List[String] с тремя значениями: "Cool", "tools" и "rule"
val thrill = "Will" :: "fill" :: "until" :: Nil	Создает новый список типа List[String] с тремя значениями: "Will", "fill" и "until"
List("a", "b") ::: List("c", "d")	Объединяет два списка (возвращает новый список типа List[String] со значениями "a", "b", "c" и "d")
thrill(2)	Возвращает элемент с индексом 2 (при начале отсчета с нуля) списка thrill (возвращает "until")
thrill.count(s => s.length == 4)	Подсчитывает количество строковых элементов в thrill, имеющих длину 4 (возвращает 2)
thrill.drop(2)	Возвращает список thrill без его первых двух элементов (возвращает List("until"))
thrill.dropRight(2)	Возвращает список thrill без крайних справа двух элементов (возвращает List("Will"))
thrill.exists(s => s == "until")	Определяет наличие в списке thrill строкового элемента, имеющего значение "until" (возвращает true)
thrill.filter(s => s.length == 4)	Возвращает список всех элементов списка thrill, имеющих длину 4, соблюдая порядок их следования в списке (возвращает List("Will", "fill"))
thrill.forall(s => s.endsWith("l"))	Показывает, заканчиваются ли все элементы в списке thrill буквой "l" (возвращает true)
thrill.foreach(s => print(s))	Выполняет инструкцию print в отношении каждой строки в списке thrill (выводит "Willfilluntil")

Что используется	Что этот метод делает
thrill.foreach(print)	Делает то же самое, что и предыдущий код, но с использованием более лаконичной формы записи (также выводит "Willfilluntil")
thrill.head	Возвращает первый элемент в списке thrill (возвращает "Will")
thrill.init	Возвращает список всех элементов списка thrill, кроме последнего (возвращает List("Will", "fill"))
thrill.isEmpty	Показывает, не пуст ли список thrill (возвращает false)
thrill.last	Возвращает последний элемент в списке thrill (возвращает "until")
thrill.length	Возвращает количество элементов в списке thrill (возвращает 3)
thrill.map(s => s + "y")	Возвращает список, который получается в результате добавления "y" к каждому строковому элементу в списке thrill (возвращает List("Willy", "filly", "untily"))
thrill.mkString(", ")	Создает строку с элементами списка (возвращает "Will, fill, until")
thrill.filterNot(s => s.length == 4)	Возвращает список всех элементов в порядке их следования в списке thrill, за исключением имеющих длину 4 (возвращает List("until"))
thrill.reverse	Возвращает список, содержащий все элементы списка thrill, следующие в обратном порядке (возвращает List("until", "fill", "Will"))
thrill.sortWith((s, t) => s.charAt(0).toLower < t.charAt(0).toLower)	Возвращает список, содержащий все элементы списка thrill в алфавитном порядке с первым символом, преобразованным в символ нижнего регистра (возвращает List("fill", "until", "will"))
thrill.tail	Возвращает список thrill за исключением его первого элемента (возвращает List("fill", "until"))

3.3. Шаг 9. Используем кортежи

Еще один полезный объект-контейнер — *кортеж*. Как и списки, кортежи не могут быть изменены, но, в отличие от списков, могут содержать различные типы элементов. Список может быть типа `List[Int]` или `List[String]`, а кортеж может содержать одновременно как целые числа, так и строки. Кортежи находят широкое применение, например, при возвращении из метода сразу нескольких объектов. Там, где на Java для хранения нескольких возвращаемых значений зачастую приходится создавать JavaBean-подобный класс, в Scala можно просто вернуть кортеж. Все делается просто: чтобы создать экземпляр нового кортежа, содержащего объекты, нужно лишь заключить объекты в круглые скобки, отделив их друг от друга запятыми. После создания экземпляра кортежа доступ к его элементам можно получить, используя точку, знак подчеркивания и индекс элемента, причем подсчет элементов начинается с единицы. Пример показан в листинге 3.4.

Листинг 3.4. Создание и использование кортежа

```
val pair = (99, "Luftballons")
println(pair._1)
println(pair._2)
```

В первой строке листинга 3.4 создается новый кортеж, содержащий в качестве первого элемента целочисленное значение `99`, а в качестве второго — строку `"Luftballons"`. Scala выводит тип кортежа в виде `Tuple2[Int, String]`, а также присваивает этот тип паре переменных. Во второй строке выполняется доступ к полю `_1`, в результате чего получается первый элемент `99`. Символ точки (`.`) во второй строке аналогичен той точке, которая используется для доступа к полю или вызова метода. В данном случае выполняется доступ к полю по имени `_1`. Если запустим этот скрипт на выполнение, то получим следующий результат:

```
99
Luftballons
```

Реальный тип кортежа зависит от количества содержащихся в нем элементов и от типов этих элементов. Следовательно, типом кортежа `(99, "Luftballons")` является `Tuple2[Int, String]`. А типом кортежа `('u', 'r', "the", 1, 4, "me")` является `Tuple6[Char, Char, String, Int, Int, String]`[1].

Обращение к элементам кортежа

Возникает вопрос: а почему к элементам кортежа нельзя обратиться точно так же, как к элементам списка, например `pair(0)`? Дело в том, что используемый в списках метод `apply` всегда возвращает один и тот же тип, а в кортеже все элементы могут быть разных типов: у `_1` может быть один результирующий тип, у `_2` — другой и т. д. Эти числа вида `_N` начинаются с единицы, а не с нуля, поскольку начало отсчета с единицы традиционно используется в языках со статически типизированными кортежами, например `Haskell` и `ML`.

3.4. Шаг 10. Используем множества и отображения

Scala призван помочь вам использовать преимущества как функционального, так и объектно-ориентированного стиля, поэтому в библиотеках его коллекций особое внимание обращают на разницу между изменяемыми и неизменяемыми коллекциями. Например, массивы всегда изменяемы, а списки всегда неизменяемы. Scala также предоставляет изменяемые и неизменяемые альтернативы для множеств и отображений, но использует для обеих версий одни и те же простые имена. Для множеств и отображений Scala моделирует изменяемость в иерархии классов.

Например, в API Scala содержится основной *трейт* для множеств, где этот трейт аналогичен Java-интерфейсу. (Более подробно трейты рассматриваются в главе 12.)

---

[1] Концептуально можно создавать кортежи любой длины, однако на данный момент библиотека Scala определяет их только до Tuple22.

Затем Scala предоставляет два трейта-наследника: один для изменяемых, а второй для неизменяемых множеств.

На рис. 3.2 показано, что для всех трех трейтов используется одно и то же простое имя `Set`. Но их полные имена отличаются друг от друга, поскольку все трейты размещаются в разных пакетах. Классы для конкретных множеств в Scala API, например `HashSet` (см. рис. 3.2), являются расширениями либо изменяемого, либо неизменяемого трейта `Set`. (В то время как в Java вы реализуете интерфейсы, в Scala расширяете (иначе говоря, подмешиваете) трейты.) Следовательно, если нужно воспользоваться `HashSet`, то в зависимости от потребностей можно выбирать между его изменяемой и неизменяемой разновидностями. Способ создания множества по умолчанию показан в листинге 3.5.

Листинг 3.5. Создание, инициализация и использование неизменяемого множества

```
var jetSet = Set("Boeing", "Airbus")
jetSet += "Lear"
println(jetSet.contains("Cessna"))
```

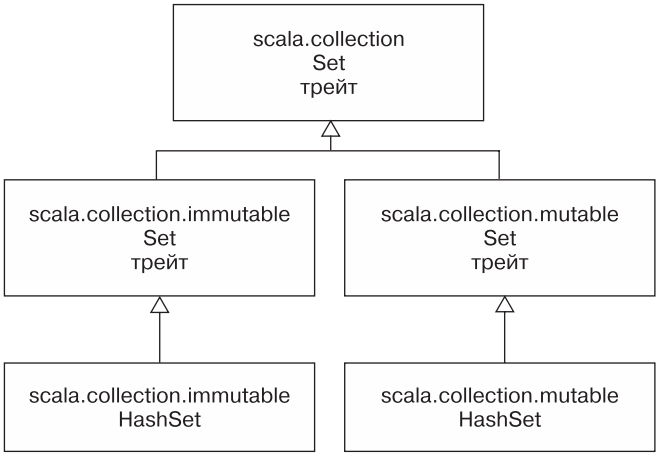


Рис. 3.2. Иерархия классов для множеств Scala

В первой строке кода листинга 3.5 определяется новая `var`-переменная по имени `jetSet`, которая инициализируется неизменяемым множеством, содержащим две строки: `"Boeing"` и `"Airbus"`. В этом примере показано, что в Scala множества можно создавать точно так же, как списки и массивы: путем вызова фабричного метода по имени `apply` в отношении объекта-компаньона `Set`. В листинге 3.5 метод `apply` вызывается в отношении объекта-компаньона для `scala.collection.immutable.Set`, возвращающего экземпляр исходного, неизменяемого класса `Set`. Компилятор Scala выводит тип переменной `jetSet`, определяя его как неизменяемый `Set[String]`.

Чтобы добавить новый элемент в неизменяемое множество, в отношении последнего вызывается метод `+`, которому и передается этот элемент. Метод `+` создает и возвращает новое неизменяемое множество с добавленным элементом. Конкретный метод `+=` предоставляется исключительно для изменяемых множеств.

В данном случае вторая строка кода, `jetSet += "Lear"`, фактически является сокращенной формой записи следующего кода:

```
jetSet = jetSet + "Lear"
```

Следовательно, во второй строке кода листинга 3.5 `var`-переменной `jetSet` присваивается новое множество, содержащее `"Boeing"`, `"Airbus"` и `"Lear"`. И наконец, в последней строке выводятся данные о том, содержится ли во множестве строка `"Cessna"`. (Как и ожидалось, выводится `false`.)

Если нужно изменяемое множество, то следует, как показано в листинге 3.6, воспользоваться инструкцией *`import`*.

Листинг 3.6. Создание, инициализация и использование изменяемого множества

```
import scala.collection.mutable

val movieSet = mutable.Set("Hitch", "Poltergeist")
movieSet += "Shrek"
println(movieSet)
```

В первой строке данного листинга выполняется импорт `scala.collection.mutable`. Инструкция `import` позволяет использовать простое имя, например `Set`, вместо длинного полного имени. В результате при указании `mutable.Set` во второй строке компилятор знает, что имеется в виду `scala.collection.mutable.Set`. В этой строке `movieSet` инициализируется новым изменяемым множеством, содержащим строки `"Hitch"` и `"Poltergeist"`. В следующей строке к изменяемому множеству добавляется `"Shrek"`, для чего в отношении множества вызывается метод `+=` с передачей ему строки `"Shrek"`. Как уже упоминалось, `+=` — метод, определенный для изменяемых множеств. При желании можете вместо кода `movieSet += "Shrek"` воспользоваться кодом `movieSet.+=("Shrek")`[1].

Рассмотренной до сих пор исходной реализации множеств, которые выполняются изменяемыми и неизменяемыми фабричными методами `Set`, скорее всего, будет достаточно для большинства ситуаций. Однако временами может потребоваться специальный вид множества. К счастью, при этом используется аналогичный синтаксис. Следует просто импортировать нужный класс и применить фабричный метод в отношении его объекта-компаньона. Например, если требуется неизменяемый `HashSet`, то можно сделать следующее:

```
import scala.collection.immutable.HashSet

val hashSet = HashSet("Tomatoes", "Chilies")
println(hashSet + "Coriander")
```

Еще одним полезным трейтом коллекций в Scala является отображение — `Map`. Как и для множеств, Scala предоставляет изменяемые и неизменяемые версии `Map` с применением иерархии классов. Как показано на рис. 3.3, иерархия классов для ото-

[1] Множество в листинге 3.6 изменяемое, поэтому повторно присваивать значение movieSet не нужно и данная переменная может относиться к val-переменным. В отличие от этого использование метода += с неизменяемым множеством в листинге 3.5 требует повторного присваивания значения переменной jetSet, поэтому она должна быть var-переменной.

бражений во многом похожа на иерархию для множеств. В пакете `scala.collection` есть основной трейт `Map` и два трейта-наследника отображения `Map`: изменяемый вариант в `scala.collection.mutable` и неизменяемый в `scala.collection.immutable`.

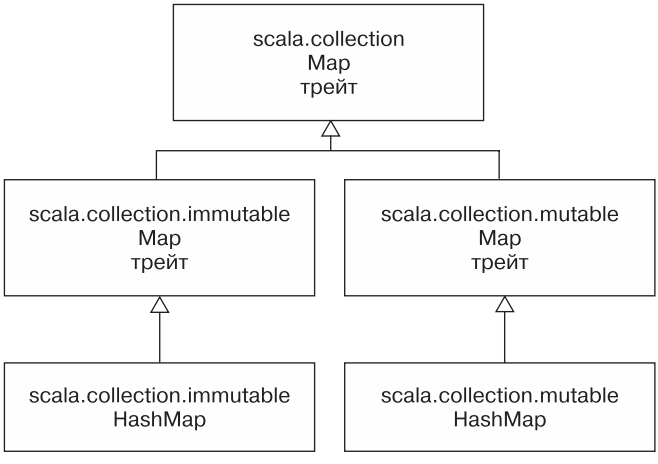


Рис. 3.3. Иерархия классов для Scala-отображений

Реализации `Map`, например `HashMap`-реализации в иерархии классов, показанной на рис. 3.3, расширяются либо в изменяемый, либо в неизменяемый трейт. Отображения можно создавать и инициализировать, используя фабричные методы, подобные тем, что применялись для массивов, списков и множеств.

Например, в листинге 3.7 показана работа с изменяемым отображением.

Листинг 3.7. Создание, инициализация и использование изменяемого отображения

```
import scala.collection.mutable

val treasureMap = mutable.Map[Int, String]()
treasureMap += (1 -> "Go to island.")
treasureMap += (2 -> "Find big X on ground.")
treasureMap += (3 -> "Dig.")
println(treasureMap(2))
```

В первой строке импортируется изменяемое отображение. Затем определяется `val`-переменная `treasureMap`, которая инициализируется пустым изменяемым отображением, имеющим целочисленные ключи и строковые значения. Отображение задается пустым, поскольку фабричному методу ничего не передается (в круглых скобках в `Map[Int, String]()` ничего не указано)[1]. В следующих трех строках

[1] Явная параметризация типа, "[Int, String]", требуется в листинге 3.7 из-за того, что без какого-либо значения, переданного фабричному методу, компилятор не в состоянии выполнить логический вывод типов параметров отображения. В отличие от этого компилятор может выполнить вывод типов параметров из значений, переданных фабричному методу map, показанному в листинге 3.8, поэтому явного указания типов параметров там не требуется.

к отображению добавляются пары «ключ — значение», для чего используются методы -> и +=. Как уже было показано, компилятор Scala преобразует выражения бинарных операций вида 1 -> "Go to island." в код (1).->("Go to island."). Следовательно, когда указывается 1 -> "Go to island.", фактически в отношении объекта 1 вызывается метод по имени ->, которому передается строка со значением "Go to island.". Метод ->, который можно вызвать в отношении любого объекта в программе Scala, возвращает двухэлементный кортеж, содержащий ключ и значение[1]. Затем этот кортеж передается методу += объекта отображения, на который ссылается treasureMap. И наконец, в последней строке выводится значение, соответствующее в treasureMap ключу 2.

Если запустить этот код, то он выведет следующие данные:

```
Find big X on ground.
```

Если отдать предпочтение неизменяемому отображению, то ничего импортировать не нужно, поскольку это отображение используется по умолчанию. Пример показан в листинге 3.8.

Листинг 3.8. Создание, инициализация и использование неизменяемого отображения

```
val romanNumeral = Map(
1 -> "I", 2 -> "II", 3 -> "III", 4 -> "IV", 5 -> "V"
)
println(romanNumeral(4))
```

Учитывая отсутствие импортирования, при указании Map в первой строке данного листинга вы получаете используемый по умолчанию экземпляр класса scala.collection.immutable.Map. Фабричному методу отображения передаются пять кортежей «ключ — значение», а он возвращает неизменяемое Map-отображение, содержащее эти переданные пары. Если запустить код, показанный в листинге 3.8, то он выведет IV.

3.5. Шаг 11. Учимся распознавать функциональный стиль

Как упоминалось в главе 1, Scala позволяет программировать в императивном стиле, но побуждает вас переходить преимущественно к функциональному. Если к Scala вы пришли, имея опыт работы в императивном стиле, к примеру, вам приходилось программировать на Java, то одной из основных возможных сложностей станет программирование в функциональном стиле. Мы понимаем, что поначалу этот стиль может быть неизвестен, и в данной книге стараемся перевести вас из одного состояния в другое. От вас также потребуются некоторые усилия, которые

[1] Используемый в Scala механизм, позволяющий вызывать метод –> в отношении любого объекта, — неявное преобразование — будет рассмотрен в главе 21.

мы настоятельно рекомендуем приложить. Мы уверены, что при наличии опыта работы в императивном стиле изучение программирования в функциональном позволит вам не только стать более квалифицированным программистом на Scala, но и расширит ваш кругозор, сделав вас более ценным программистом в общем смысле.

Сначала нужно усвоить разницу между двумя стилями, отражающуюся в коде. Один верный признак заключается в том, что если код содержит `var`-переменные, то он, вероятнее всего, написан в императивном стиле. Если он вообще не содержит `var`-переменных, то есть включает *только* `val`-переменные, то, вероятнее всего, он написан в функциональном стиле. Следовательно, один из способов приблизиться к последнему — попытаться обойтись в программах без `var`-переменных.

Обладая багажом императивности, то есть опытом работы с такими языками, как Java, C++ или C#, `var`-переменные можно рассматривать в качестве обычных, а `val`-переменные — в качестве переменных особого вида. В то же время, если у вас имеется опыт работы в функциональном стиле на таких языках, как Haskell, OCaml или Erlang, `val`-переменные можно представлять как обычные, а `var`-переменные — как некое кощунственное обращение с кодом. Но с точки зрения Scala `val`- и `var`-переменные — всего лишь два разных инструмента в вашем арсенале средств и оба одинаково полезны и не отвергаемы. Scala побуждает вас к использованию `val`-переменных, но, по сути, дает возможность применять тот инструмент, который лучше подходит для решаемой задачи. И тем не менее, даже будучи согласными с подобной философией, вы поначалу можете испытывать трудности, связанные с избавлением от `var`-переменных в коде.

Рассмотрим позаимствованный из главы 2 пример цикла `while`, в котором используется `var`-переменная, означающая, что он выполнен в императивном стиле:

```
def printArgs(args: Array[String]): Unit = {
  var i = 0
  while (i < args.length) {
    println(args(i))
    i += 1
  }
}
```

Вы можете преобразовать этот код — придать ему более функциональный стиль, отказавшись от использования `var`-переменной, например, так:

```
def printArgs(args: Array[String]): Unit = {
  for (arg <- args)
    println(arg)
}
```

или вот так:

```
def printArgs(args: Array[String]): Unit = {
  args.foreach(println)
}
```

В этом примере демонстрируется одно из преимуществ программирования с меньшим количеством `var`-переменных. Код после рефакторинга (более функциональный) выглядит понятнее, он более лаконичен, и в нем труднее допустить какие-либо ошибки, чем в исходном (более императивном) коде. Причина навязывания в Scala функционального стиля заключается в том, что он помогает создавать более понятный код, при написании которого труднее ошибиться.

Но вы можете пойти еще дальше. Метод после рефакторинга `printArgs` нельзя отнести к *чисто* функциональным, поскольку у него имеются побочные эффекты. В данном случае такой эффект — вывод в поток стандартного устройства вывода. Признаком функции, имеющей побочные эффекты, выступает то, что результирующим типом у нее является `Unit`. Если функция не возвращает никакое интересное значение, о чем, собственно, и свидетельствует результирующий тип `Unit`, то единственный способ внести этой функцией какое-либо изменение в окружающий мир — проявить некий побочный эффект. Более функциональным подходом будет определение метода, который форматирует передаваемые аргументы в целях их последующего вывода и, как показано в листинге 3.9, просто возвращает отформатированную строку.

Листинг 3.9. Функция без побочных эффектов или var-переменных

```
def formatArgs(args: Array[String]) = args.mkString("\n")
```

Теперь вы действительно перешли на функциональный стиль: нет ни побочных эффектов, ни `var`-переменных. Метод `mkString`, который можно вызвать в отношении любой коллекции, допускающей последовательный перебор элементов (включая массивы, списки, множества и отображения), возвращает строку, состоящую из результата вызова метода `toString` в отношении каждого элемента, с разделителями из переданной строки. Таким образом, если `args` содержит три элемента, `"zero"`, `"one"` и `"two"`, то метод `formatArgs` возвращает `"zero\none\ntwo"`. Разумеется, эта функция, в отличие от методов `printArgs`, ничего не выводит, но в целях выполнения данной работы ее результаты можно легко передать функции `println`:

```
println(formatArgs(args))
```

Каждая полезная программа, вероятнее всего, будет иметь какие-либо побочные эффекты. Отдавая предпочтение методам без побочных эффектов, вы будете стремиться к разработке программ, в которых такие эффекты сведены к минимуму. Одним из преимуществ такого подхода станет упрощение тестирования ваших программ.

Например, чтобы протестировать любой из трех показанных ранее в этом разделе методов `printArgs`, вам придется переопределить метод `println`, перехватить передаваемый ему вывод и убедиться в том, что он соответствует вашим ожиданиям. В отличие от этого функцию `formatArgs` можно протестировать, просто проверяя ее результат:

```
val res = formatArgs(Array("zero", "one", "two"))
assert(res == "zero\none\ntwo")
```

Имеющийся в Scala метод `assert` проверяет переданное ему булево выражение и, если последнее вычисляется в `false`, выдает ошибку `AssertionError`. Если же переданное булево выражение вычисляется в `true`, то метод просто молча возвращает управление вызвавшему его коду. Более подробно о тестах, проводимых с помощью `assert`, и тестировании речь пойдет в главе 14.

И все-таки нужно иметь в виду: ни `var`-переменные, ни побочные эффекты не следует рассматривать как нечто абсолютно неприемлемое. Scala не является чисто функциональным языком, заставляющим вас программировать в функциональном стиле. Scala — гибрид императивного и функционального языков. Может оказаться, что в некоторых ситуациях для решения текущей задачи больше подойдет императивный стиль, и тогда вы должны прибегнуть к нему без всяких колебаний. Но чтобы помочь вам разобраться в программировании без использования `var`-переменных, в главе 7 мы покажем множество конкретных примеров кода с использованием `var`-переменных и рассмотрим способы их преобразования в `val`-переменные.

Сбалансированная позиция для программистов, работающих на Scala

Отдавайте предпочтение `val`-переменным, неизменяемым объектам и методам без побочных эффектов. Стремитесь применять их в первую очередь. Используйте `var`-переменные, изменяемые объекты и методы с побочными эффектами в случае необходимости и при наличии четкой обоснованности их применения.

3.6. Шаг 12. Читаем строки из файла

Скрипты, выполняющие небольшие повседневные задачи, часто нуждаются в обработке строк, взятых из файлов. В этом разделе будет создан скрипт, который считывает строки из файла и выводит их на стандартное устройство, предваряя каждую строку количеством содержащихся в ней символов. Первая версия скрипта показана в листинге 3.10.

Листинг 3.10. Считывание строк из файла

```
import scala.io.Source

if (args.length > 0) {

  for (line <- Source.fromFile(args(0)).getLines())
    println(line.length.toString + " " + line)
}
else
  Console.err.println("Please enter filename")
```

Скрипт начинается с импорта класса `Source` из пакета `scala.io`. Затем он проверяет, указан ли в командной строке хотя бы один аргумент. Если указан, то первый аргумент рассматривается как имя открываемого и обрабатываемого файла. Выражение `Source.fromFile(args(0))` пробует открыть указанный файл и возвращает объект типа `Source`, в отношении которого вызывается метод `getLines`. Этот метод возвращает значение типа `Iterator[String]`, в котором при каждой итерации предоставляется по одной строке за исключением символа конца строки. Выражение `for` выполняет последовательный перебор этих строк и выводит длину каждой строки, затем пробел, а потом саму строку. Если аргументы в командной строке не указаны, то финальное условие `else` выведет сообщение в поток стандартного устройства вывода. Поместив данный код в файл `countchars1.scala` и запустив его с указанием самого этого файла:

```
$ scala countchars1.scala countchars1.scala
```

вы увидите следующий текст:

```
22 import scala.io.Source
0
22 if (args.length > 0) {
0
51   for (line <- Source.fromFile(args(0)).getLines())
37     println(line.length + " " + line)
1 }
4 else
46   Console.err.println("Please enter filename")
```

Скрипт в его текущем виде выводит необходимую информацию, однако, возможно, вам захочется выстроить числа, выровняв их по правому краю, и добавить символ вертикальной черты, чтобы выводимая информация приобрела следующий вид:

```
22 | import scala.io.Source
 0 |
22 | if (args.length > 0) {
 0 |
51 |   for (line <- Source.fromFile(args(0)).getLines())
37 |     println(line.length + " " + line)
 1 | }
 4 | else
46 |   Console.err.println("Please enter filename")
```

Чтобы реализовать задуманное, можно выполнить последовательный перебор строк дважды. При первом переборе нужно определить максимальную ширину, требующуюся для вывода количества символов в любой строке. А при втором — вывести данные, используя вычисленную ранее максимальную ширину. Поскольку перебор строк будет выполняться дважды, то вы также можете присвоить эти строки переменной:

```
val lines = Source.fromFile(args(0)).getLines().toList
```

Завершающий выражение вызов метода `toList` нужен потому, что метод `getLines` возвращает итератор. По мере проведения итерации через итератор он истощается. Преобразование в список с помощью вызова `toList` дает возможность выполнять итерацию любое необходимое количество раз, не прибегая к многократному выделению памяти для хранения всех строк из файла. Получается, переменная `lines` ссылается на список строк с содержимым файла, указанного в командной строке. Вам нужно вычислять ширину позиции для количества символов в каждой строке дважды — по одному разу за каждую итерацию. Поэтому из данного выражения можно вывести небольшую функцию, которая подсчитывает ширину, необходимую для символов, отображающих длину переданной строки:

```
def widthOfLength(s: String) = s.length.toString.length
```

Применяя эту функцию, максимальную ширину можно вычислить следующим образом:

```
var maxWidth = 0
for (line <- lines)
  maxWidth = maxWidth.max(widthOfLength(line))
```

Здесь с помощью выражения `for` выполняется последовательный перебор всех строк, вычисляется ширина, необходимая для символов, показывающих длину строки. И если она больше текущего максимума, то ее значение присваивается `var`-переменной `maxWidth`, которой было присвоено начальное значение `0`. (Метод `max`, который можно вызвать в отношении любого объекта типа `Int`, возвращает большее число, сравнивая значение, в отношении которого он был вызван, и переданное ему значение.) В качестве альтернативного варианта, если предпочтительнее искать максимум, не прибегая к использованию `var`-переменных, можно сначала определить самую длинную строку:

```
val longestLine = lines.reduceLeft(
  (a, b) => if (a.length > b.length) a else b
)
```

Метод `reduceLeft` применяет переданную ему функцию к первым двум элементам в списке `lines`, затем — к результату первого применения и к следующему элементу в `lines` и далее до конца списка. Результатом каждого применения будет самая длинная строка из встреченных до сих пор, поскольку переданная функция `(a, b) => if (a.length > b.length) a else b` возвращает самую длинную из двух переданных строк. Метод `reduceLeft` возвратит результат последнего применения функции, который в данном случае будет самым длинным строковым элементом списка `lines`.

Получив результат, можно вычислить максимальную ширину, передав самую длинную строку функции `widthOfLength`:

```
val maxWidth = widthOfLength(longestLine)
```

Теперь останется только вывести строки с надлежащим форматированием. Это можно сделать следующим образом:

```
for (line <- lines) {
  val numSpaces = maxWidth - widthOfLength(line)
  val padding = " " * numSpaces
  println(padding + line.length + " | " + line)
}
```

В этом выражении `for` еще раз выполняет последовательный перебор элементов списка `lines`. Для каждой строки сначала вычисляется количество пробелов, устанавливаемых перед указанием длины строки, и присваивается переменной `numSpaces` с помощью выражения `" " * numSpaces`. И наконец, выводится информация с надлежащим форматированием. Весь скрипт показан в листинге 3.11.

Листинг 3.11. Вывод отформатированного количества символов каждой строки файла

```
import scala.io.Source

def widthOfLength(s: String) = s.length.toString.length

if (args.length > 0) {

  val lines = Source.fromFile(args(0)).getLines().toList

  val longestLine = lines.reduceLeft(
    (a, b) => if (a.length > b.length) a else b
  )
  val maxWidth = widthOfLength(longestLine)

  for (line <- lines) {
    val numSpaces = maxWidth - widthOfLength(line)
    val padding = " " * numSpaces
    println(padding + line.length + " | " + line)
  }
}
else
  Console.err.println("Пожалуйста, введите имя файла ")
```

Резюме

Знания, полученные в этой главе, позволят вам начать применять Scala для решения небольших задач, в особенности тех, для которых используются скрипты. В последующих главах мы глубже изучим рассмотренные темы, а также представим другие, не затронутые здесь.

4 Классы и объекты

В предыдущих двух главах вы разобрались в основах классов и объектов. В этой главе вам предстоит углубленно проработать данную тему. Здесь мы дадим дополнительные сведения о классах, полях и методах, а также общее представление о том, когда подразумевается использование точки с запятой. Кроме того, рассмотрим объекты-одиночки (singleton) и то, как с их помощью писать и запускать приложения на Scala. Если вам уже знаком язык Java, то вы увидите, что в Scala фигурируют похожие, но все же немного отличающиеся концепции. Поэтому чтение данной главы пойдет на пользу даже великим знатокам языка Java.

4.1. Классы, поля и методы

Классы — «чертежи» объектов. После определения класса из него, как по чертежу, можно создавать объекты, воспользовавшись для этого ключевым словом `new`. Например, при наличии следующего определения класса:

```
class ChecksumAccumulator {
  // Сюда помещается определение класса
}
```

с помощью кода:

```
new ChecksumAccumulator
```

можно создавать объекты `ChecksumAccumulator`.

Внутри определения класса помещаются поля и методы, которые в общем называются *членами* класса. Поля, которые определяются либо как `val`-, либо как `var`-переменные, являются переменными, относящимися к объектам. Методы, определяемые с помощью ключевого слова `def`, содержат исполняемый код. В полях хранятся состояние или данные объекта, а методы используют эти данные для выполнения в отношении объекта вычислений. При создании экземпляра класса среда выполнения приложения резервирует часть памяти для хранения образа состояния получающегося при этом объекта (то есть содержимого его переменных).

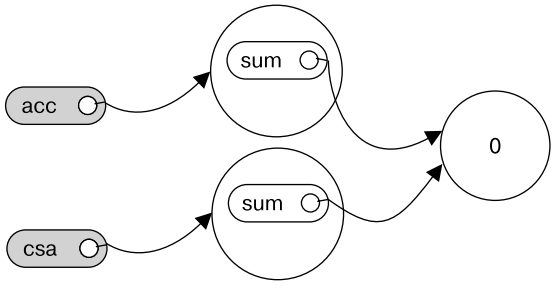
Например, если вы определите класс `ChecksumAccumulator` и дадите ему `var`-поле по имени `sum`:

```
class ChecksumAccumulator {
  var sum = 0
}
```

а потом дважды создадите его экземпляры с помощью следующего кода:

```
val acc = new ChecksumAccumulator
val csa = new ChecksumAccumulator
```

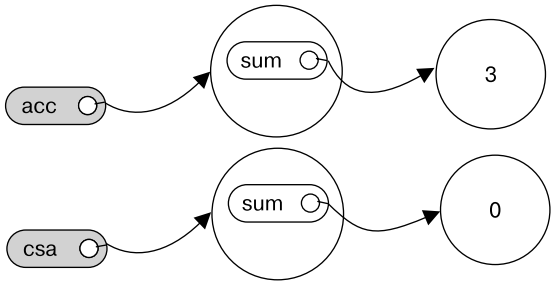
то образ объектов в памяти может выглядеть так:



Поскольку `sum` — поле, определенное внутри класса `ChecksumAccumulator`, и относится к `var`-, а не к `val`-переменным, то впоследствии ему (полю) можно заново присвоить другое `Int`-значение:

```
acc.sum = 3
```

Теперь картинка может выглядеть так:



По поводу этой картинки нужно отметить следующее: на ней показаны две переменные `sum`. Одна из них находится в объекте, на который ссылается `acc`, а другая — в объекте, на который ссылается `csa`. Поля также называют *переменными экземпляра*, поскольку каждый экземпляр получает собственный набор переменных. Все переменные экземпляра объекта составляют образ объекта в памяти. То, что здесь показано, свидетельствует не только о наличии двух переменных `sum`, но и о том, что изменение одной из них никак не отражается на другой.

В этом примере следует также отметить: у вас есть возможность изменить объект, на который ссылается `acc`, даже несмотря на то, что `acc` относится к `val`-переменным. Но с учетом того, что `acc` (или `csa`) являются `val`-, а не `var`-переменными, вы не можете присвоить им какой-нибудь другой объект. Например, попытка, показанная ниже, не будет успешной:

```
// Не пройдет компиляцию, поскольку acc является val-переменной
acc = new ChecksumAccumulator
```

Теперь вы можете рассчитывать на то, что переменая `acc` всегда будет ссылаться на тот же объект `ChecksumAccumulator`, с помощью которого вы ее инициализировали; поля же, содержащиеся внутри этого объекта, могут со временем измениться.

Один из важных способов обеспечения надежности объекта — гарантия того, что состояние этого объекта, то есть значения его переменных экземпляра, остается корректным в течение всего его жизненного цикла. Первый шаг к предотвращению непосредственного стороннего доступа к полям — создание *приватных* (private) полей. Доступ к приватным полям можно получить только методами, определенными в том же самом классе, поэтому весь код, который может обновить состояние, будет локализован в классе. Чтобы объявить поле приватным, перед ним нужно поставить модификатор доступа `private`:

```
class ChecksumAccumulator {
  private var sum = 0
}
```

С таким определением `ChecksumAccumulator` любая попытка доступа к `sum` за пределами класса будет неудачной:

```
val acc = new ChecksumAccumulator
acc.sum = 5 // Не пройдет компиляцию, поскольку поле sum является приватным
```

ПРИМЕЧАНИЕ

> В Scala элементы класса делают публичными, если нет явного указания какого-либо модификатора доступа. Иначе говоря, там, где в Java ставится модификатор public, в Scala вы обходитесь простым замалчиванием. Публичный (public) доступ в Scala — уровень доступа по умолчанию.

Теперь, когда поле `sum` стало приватным, доступ к нему можно получить только из кода, определенного внутри тела самого класса. Следовательно, класс `ChecksumAccumulator` не будет особо полезен, пока внутри него не будут определены некоторые методы:

```
class ChecksumAccumulator {

  private var sum = 0

  def add(b: Byte): Unit = {
    sum += b
  }
```

```
  def checksum(): Int = {
    return ~(sum & 0xFF) + 1
  }
}
```

Теперь у `ChecksumAccumulator` есть два метода, `add` и `checksum`, и оба они демонстрируют основную форму определения функции, показанную на рис. 2.1 (см. выше).

Внутри этого метода могут использоваться любые параметры метода. Одной из важных характеристик параметров метода в Scala является то, что они относятся к `val`-, а не к `var`-переменным[1]. При попытке повторного присваивания значения параметру внутри метода в Scala произойдет сбой компиляции:

```
def add(b: Byte): Unit = {
  b = 1    // Не пройдет компиляцию, поскольку b относится к val-переменным
  sum += b
}
```

Хотя методы `add` и `checksum` в данной версии `ChecksumAccumulator` реализуют желаемые функциональные свойства вполне корректно, их можно определить в более лаконичном стиле. Во-первых, в конце метода `checksum` можно избавиться от лишнего слова `return`. В отсутствие явно указанной инструкции `return` метод в Scala возвращает последнее вычисленное им значение.

При написании методов рекомендуется применять стиль, исключающий явное и особенно многократное использование инструкции `return`. Каждый метод нужно рассматривать в качестве выражения, выдающего одно значение, которое и является возвращаемым. Эта философия будет побуждать вас создавать небольшие методы и разбивать слишком крупные методы на несколько мелких. В то же время выбор конструктивного решения зависит от контекста решаемых задач, и, если того требуют условия, Scala упрощает написание методов, которые имеют несколько явно указанных возвращаемых значений.

Метод `checksum` всего лишь вычисляет значение, поэтому ему не нужна явно указанная инструкция `return`. Еще один способ сократить метод — отказаться от использования фигурных скобок, если в методе вычисляется только одно выражение, дающее результат. Если результирующее выражение достаточно краткое, то его можно даже поместить на той же строке, в которой указано ключевое слово `def`. В целях максимальной лаконичности можно не указывать результирующий тип, и тогда Scala выведет его самостоятельно. После внесения всех этих изменений класс `ChecksumAccumulator` приобретет следующий вид:

```
class ChecksumAccumulator {
  private var sum = 0
```

[1] Причина, по которой val-переменные используются в качестве параметров, заключается в более легком составлении представления об этих переменных. Такую переменную больше не нужно отслеживать на предмет присваивания ей нового значения, как это было бы в случае использования var-переменной.

```
  def add(b: Byte) = sum += b
  def checksum() = ~(sum & 0xFF) + 1
}
```

Несмотря на то что компилятор Scala вполне корректно выполнит вывод результирующих типов методов `add` и `checksum`, показанных в предыдущем примере, читатели кода будут вынуждены вывести результирующие типы путем *логических умозаключений* на основе изучения тел методов. Поэтому лучше все-таки будет всегда явно указывать результирующие типы для публичных методов, объявленных в классе, даже когда компилятор может вывести их для вас самостоятельно. Применение этого стиля показано в листинге 4.1.

Листинг 4.1. Окончательная версия класса ChecksumAccumulator

```
// Этот код находится в файле ChecksumAccumulator.scala
class ChecksumAccumulator {
  private var sum = 0
  def add(b: Byte): Unit = { sum += b }
  def checksum(): Int = ~(sum & 0xFF) + 1
}
```

Методы с результирующим типом `Unit`, к которым относится и метод `add` класса `ChecksumAccumulator`, выполняются для получения побочного эффекта. Последний обычно определяется в виде изменения внешнего по отношению к методу состояния или в виде выполнения какой-либо операции ввода-вывода. Что касается метода `add`, то побочный эффект заключается в присваивании `sum` нового значения. Метод, который выполняется только для получения его побочного эффекта, называется *процедурой*.

4.2. Когда подразумевается использование точки с запятой

В программе на Scala точку с запятой в конце инструкции обычно можно не ставить. Если вся инструкция помещается на одной строке, то при желании можете поставить в конце данной строки точку с запятой, но это не обязательно. В то же время точка с запятой нужна, если на одной строке размещаются сразу несколько инструкций:

```
val s = "hello"; println(s)
```

Если требуется набрать инструкцию, занимающую несколько строк, то в большинстве случаев вы можете просто ее ввести, а Scala разделит инструкции в нужном месте. Например, следующий код рассматривается как одна инструкция, расположенная на четырех строках:

```
if (x < 2)
  println("too small")
else
  println("ok")
```

И все же временами Scala разбивает инструкции на две части, вопреки вашим желаниям:

```
x
+ y
```

Этот код рассматривается как две инструкции, `x` и `+y`. Если подразумевается, что он должен рассматриваться как одна инструкция `x + y`, то его нужно заключать в круглые скобки:

```
(x
+ y)
```

В качестве альтернативного варианта можно поместить `+` в конец строки. Именно поэтому при составлении цепочки из инфиксных операций, таких как `+`, в Scala обычно применяется стиль, который предусматривает помещение операторов в конец строки, а не в начало:

```
x +
y +
z
```

Правило, при соблюдении которого подразумевается постановка точки с запятой

Принцип работы точных правил разделения инструкций удивительно прост. Если вкратце, то конец строки рассматривается как точка с запятой, пока не возникнет одно из следующих условий.

1. Рассматриваемая строка заканчивается элементом, который согласно правилам не может находиться в конце строки, например точкой или инфиксным оператором.
2. Следующая строка начинается с элемента, с которого не может начинаться инструкция.
3. Строка заканчивается внутри круглых `(...)` или квадратных `[...]` скобок, поскольку там в любом случае не могут содержаться сразу несколько инструкций.

4.3. Объекты-одиночки

Как упоминалось в главе 1, один из аспектов, позволяющих Scala быть более объектно-ориентированным языком, чем Java, заключается в том, что в классах Scala не могут содержаться статические элементы. Вместо этого в Scala есть *объекты-одиночки*, или *синглтоны*. Определение объекта-одиночки выглядит так же, как определение класса, за исключением того, что вместо ключевого слова `class` используется ключевое слово `object`. Пример показан в листинге 4.2.

Объект-одиночка в этом листинге называется `ChecksumAccumulator`, то есть носит имя, совпадающее с именем класса в предыдущем примере. Когда объ-

ект-одиночка использует общее с классом имя, то для класса он называется *объектом-компаньоном*. И класс, и его объект-компаньон нужно определять в одном и том же исходном файле. Класс по отношению к объекту-одиночке называется *классом-компаньоном*. Класс и его объект-компаньон могут обращаться к приватным элементам друг друга.

Листинг 4.2. Объект-компаньон для класса ChecksumAccumulator

```
// Этот код находится в файле ChecksumAccumulator.scala
import scala.collection.mutable

object ChecksumAccumulator {

  private val cache = mutable.Map.empty[String, Int]

  def calculate(s: String): Int =
    if (cache.contains(s))
      cache(s)
    else {
      val acc = new ChecksumAccumulator
      for (c <- s)
        acc.add(c.toByte)
      val cs = acc.checksum()
      cache += (s -> cs)
      cs
    }
}
```

Объект-одиночка `ChecksumAccumulator` располагает одним методом по имени `calculate`, который получает строку `String` и вычисляет контрольную сумму символов этой строки. Вдобавок он имеет одно приватное поле `cache`, представленное изменяемым отображением, в котором кэшируются ранее вычисленные контрольные суммы[1]. В первой строке метода, `if (cache.contains(s))`, определяется, не содержится ли в отображении `cache` переданная строка в качестве ключа. Если да, то просто возвращается отображенное на этот ключ значение `cache(s)`. В противном случае выполняется условие `else`, вычисляющее контрольную сумму. В первой строке условия `else` определяется `val`-переменная по имени `acc`, которая инициализируется новым экземпляром `ChecksumAccumulator`[2]. В следующей строке

[1] Здесь cache используется, чтобы показать объект-одиночку с полем. Кэширование с помощью поля cache помогает оптимизировать производительность, сокращая за счет расхода памяти время вычисления и разменивая расход памяти на время вычисления. Как правило, использовать кэш-память таким образом целесообразно только в том случае, если с ее помощью можно решить проблемы производительности и воспользоваться отображением со слабыми ссылками, например WeakHashMap в scala.collection.mutable, чтобы записи в кэш-памяти могли попадать в сборщик мусора при наличии дефицита памяти.

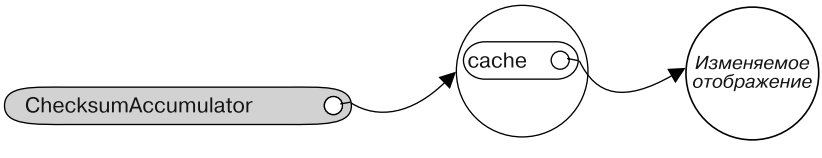
[2] Поскольку ключевое слово new используется только для создания экземпляров классов, новый объект, созданный здесь в качестве экземпляра класса ChecksumAccumulator, не является одноименным объектом-одиночкой.

находится выражение `for`. Оно выполняет последовательный перебор каждого символа в переданной строке, преобразует символ в значение типа `Byte`, вызывая в отношении этого символа метод `toByte`, и передает результат в метод `add` того экземпляра `ChecksumAccumulator`, на который ссылается `acc`. Когда завершится вычисление выражения `for`, в следующей строке метода в отношении `acc` будет вызван метод `checksum`, который берет контрольную сумму для переданного значения типа `String` и сохраняет ее в `val`-переменной по имени `cs`. В следующей строке, `cache += (s -> cs)`, переданный строковый ключ отображается на целочисленное значение контрольной суммы, и эта пара «ключ — значение» добавляется в отображение `cache`. В последнем выражении метода, `cs`, обеспечивается использование контрольной суммы в качестве результата выполнения метода.

Если у вас есть опыт программирования на Java, то объекты-одиночки можно представить в качестве хранилища для любых статических методов, которые вполне могли быть написаны на Java. Методы в объектах-одиночках можно вызывать с помощью такого синтаксиса: имя объекта, точка, имя метода. Например, метод `calculate` объекта-одиночки `ChecksumAccumulator` можно вызвать следующим образом:

```
ChecksumAccumulator.calculate("Каждое значениие является объектом.")
```

Но объект-одиночка не только хранилище статических методов. Он объект первого класса. Поэтому имя объекта-одиночки можно рассматривать в качестве «этикетки», прикрепленной к объекту.



Определение объекта-одиночки не является определением типа на том уровне абстракции, который используется в Scala. Имея лишь определение объекта `ChecksumAccumulator`, невозможно создать одноименную переменную типа. Точнее, тип с именем `ChecksumAccumulator` определяется классом-компаньоном объекта-одиночки. Тем не менее объекты-одиночки расширяют суперкласс и могут подмешивать трейты. Учитывая то, что каждый объект-одиночка — экземпляр своего суперкласса и подмешанных в него трейтов, его методы можно вызывать через эти типы, ссылаясь на него из переменных этих типов и передавая ему методы, ожидающие использования этих типов. Примеры объектов-одиночек, являющихся наследниками классов и трейтов, показаны в главе 13.

Одно из отличий классов от объектов-одиночек состоит в том, что объекты-одиночки не могут принимать параметры, а классы — могут. Создать экземпляр объекта-одиночки с помощью ключевого слова `new` нельзя, поэтому передать ему параметры не представляется возможным. Каждый объект-одиночка реализуется

как экземпляр *синтетического класса*, ссылка на который находится в статической переменной, поэтому у них и у статических классов Java одинаковая семантика инициализации[1]. В частности, объект-одиночка инициализируется при первом обращении к нему какого-либо кода.

Объект-одиночка, который не имеет общего имени с классом-компаньоном, называется *самостоятельным*. Такие объекты можно применять для решения многих задач, включая сбор в одно целое родственных вспомогательных методов или определение точки входа в приложение Scala. Именно этот случай мы и рассмотрим в следующем разделе.

4.4. Приложение на языке Scala

Чтобы запустить программу на Scala, нужно предоставить имя автономного объекта-одиночки с методом `main`, который получает один параметр с типом `Array[String]` и имеет результирующий тип `Unit`. Точкой входа в приложение может стать любой самостоятельный объект с методом `main`, имеющим надлежащую сигнатуру. Пример показан в листинге 4.3.

Листинг 4.3. Приложение Summer

```
// Код находится в файле Summer.scala
import ChecksumAccumulator.calculate

object Summer {
  def main(args: Array[String]) = {
    for (arg <- args)
      println(arg + ": " + calculate(arg))
  }
}
```

Объект-одиночка, показанный в данном листинге, называется `Summer`. Его метод `main` имеет надлежащую сигнатуру, поэтому его можно задействовать в качестве приложения. Первая инструкция в файле импортирует метод `calculate`, который определен в объекте `ChecksumAccumulator` из предыдущего примера. Инструкция `import` позволяет далее использовать в файле простое имя метода[2]. Тело метода `main` всего лишь выводит на стандартное устройство каждый аргумент и контрольную сумму для аргумента, разделяя их двоеточием.

---

[1] В качестве имени синтетического класса используется имя объекта со знаком доллара. Следовательно, синтетический класс, применяемый для объекта-одиночки ChecksumAccumulator, называется ChecksumAccumulator$.

[2] Наличие опыта программирования на Java позволяет сопоставить такой импорт с объявлением статического импорта, введенным в Java 5. Единственное отличие — в Scala импортировать элементы можно из любого объекта, а не только из объектов-одиночек.

ПРИМЕЧАНИЕ

Подразумевается, что в каждый свой исходный файл Scala импортирует элементы пакетов java.lang и scala, а также элементы объекта-одиночки по имени Predef. В Predef, который находится в пакете scala, содержится множество полезных методов. Например, когда в исходном файле Scala встречается println, фактически вызывается println из Predef. (А метод Predef.println, в свою очередь, вызывает метод Console.println, который фактически и выполняет всю работу.) Когда же встречается assert, вызывается метод Predef.assert.

Чтобы запустить приложение Summer, поместите код из листинга 4.3 в файл `Summer.scala`. В Summer используется `ChecksumAccumulator`, поэтому поместите код для `ChecksumAccumulator` как для класса, показанного в листинге 4.1, так и для его объекта-компаньона, показанного в листинге 4.2, в файл `ChecksumAccumulator.scala`.

Одним из отличий Scala от Java является то, что в Java от вас требуется поместить публичный класс в файл, названный по имени класса, например, класс `SpeedRacer` — в файл `SpeedRacer.java`. А в Scala файл с расширением `.scala` можно называть как угодно, независимо от того, какие классы Scala или код в них помещаются. Но обычно, когда речь идет не о скриптах, рекомендуется придерживаться стиля, при котором файлы называются по именам включенных в них классов, как это делается в Java, чтобы программистам было легче искать классы по именам их файлов. Именно этим подходом мы и воспользовались в отношении двух файлов в данном примере. Имеются в виду файлы `Summer.scala` и `ChecksumAccumulator.scala`.

Ни `ChecksumAccumulator.scala`, ни `Summer.scala` не являются скриптами, поскольку заканчиваются определением. В отличие от этого скрипт должен заканчиваться выражением, выдающим результат. Поэтому при попытке запустить `Summer.scala` в качестве скрипта интерпретатор Scala пожалуется на то, что `Summer.scala` не заканчивается выражением, выдающим результат. (Конечно, если предположить, что вы самостоятельно не добавили какое-либо выражение после определения объекта `Summer`.) Вместо этого нужно будет скомпилировать данные файлы с помощью компилятора Scala, а затем запустить получившиеся в результате файлы классов. Для этого можно воспользоваться основным компилятором Scala по имени `scalac`:

```
$ scalac ChecksumAccumulator.scala Summer.scala
```

Эта команда скомпилирует ваши исходные файлы, но завершиться компиляция может после весьма ощутимой задержки. Дело в том, что при каждом своем запуске компилятор тратит время на сканирование содержимого `jar`-файлов и выполнение другой предварительной работы до того, как обратит внимание на новые исходные файлы. Поэтому в дистрибутив Scala включается *работающий в фоновом режиме* Scala-компилятор под названием `fsc` (от fast Scala compiler — быстрый компилятор Scala). Он используется следующим образом:

```
$ fsc ChecksumAccumulator.scala Summer.scala
```

При первом запуске `fsc` будет создан локальный фоновый сервер (демон), прикрепленный к порту вашего компьютера. Затем через этот порт будут отправляться списки компилируемых файлов, а фоновая программа станет заниматься их компиляцией. При следующем запуске `fsc` она уже будет запущена, поэтому `fsc` просто отправит ей список файлов, а та без промедления скомпилирует их. Использование `fsc` чревато ожиданием запуска среды выполнения Java лишь в первый раз. Если понадобится остановить фоновую программу `fsc`, то можно будет прибегнуть к команде `fsc -shutdown`.

Запуск любой из команд `scalac` или `fsc` приведет к созданию файлов классов Java, которые затем можно будет запускать через команду `scala` — ту же самую, с помощью которой вы вызывали интерпретатор в предыдущих примерах. Во всех предыдущих примерах интерпретатору передавалось имя файла с расширением `.scala`, который содержит предназначенный для интерпретации код Scala[1]. В данном же случае ему нужно будет передать имя самостоятельного объекта, содержащего метод `main`, имеющий надлежащую сигнатуру. Следовательно, приложение Summer можно запустить, набрав команду:

```
$ scala Summer of love
```

Вы сможете увидеть контрольные суммы, выведенные для двух аргументов командной строки:

```
of: -213
love: -182
```

4.5. Трейт App

В Scala есть трейт `scala.App`, который призван экономить время, которое вы тратите на набор текста. До сих пор мы не рассматривали все позволившее бы досконально разобраться в работе трейтов, однако все же решили, что об этом трейте вам захочется узнать прямо сейчас. Пример его использования показан в листинге 4.4.

Листинг 4.4. Использование трейта App

```
import ChecksumAccumulator.calculate

object FallWinterSpringSummer extends App {

  for (season <- List("fall", "winter", "spring"))
    println(season + ": " + calculate(season))
}
```

[1] Механизм, который программа scala использует для интерпретации исходного файла Scala, заключается в том, что она компилирует исходный код Scala в байт-коды Java, тут же загружает их с помощью загрузчика класса и приступает к их выполнению.

Чтобы воспользоваться трейтом, сначала нужно указать после имени вашего объекта-одиночки инструкцию расширения `extends App`. Затем, вместо того чтобы писать метод `main`, между фигурными скобками объекта-одиночки нужно набрать код, который вы поместили бы непосредственно в метод `main`. Обратиться к аргументам командной строки можно через массив строковых элементов по имени `args`. Вот и все. Это приложение можно скомпилировать и запустить точно так же, как и любое другое.

Резюме

В этой главе мы рассмотрели основы классов и объектов в Scala и показали приемы компиляции и запуска приложений. В следующей главе рассмотрим основные типы данных и их использование.

5 Основные типы и операции

После того как были рассмотрены в действии классы и объекты, самое время поглубже изучить имеющиеся в Scala основные типы и операции. Если вы хорошо знакомы с Java, то вас может обрадовать тот факт, что в Scala и в Java основные типы и операторы имеют тот же смысл. И все же есть интересные различия, ради которых с этой главой стоит ознакомиться даже тем, кто считает себя опытным разработчиком Java-приложений. Некоторые аспекты Scala, рассматриваемые в данной главе, в основном такие же, как и в Java, поэтому мы указываем, какие разделы Java-разработчики могут пропустить.

В текущей главе мы представим обзор основных типов Scala, включая строки типа `String` и типы значений `Int`, `Long`, `Short`, `Byte`, `Float`, `Double`, `Char` и `Boolean`. Кроме того, рассмотрим операции, которые могут выполняться с этими типами, и вопросы соблюдения приоритета операторов в выражениях Scala. Поговорим мы и о том, как неявные преобразования могут обогатить варианты основных типов, позволяя выполнять дополнительные операции вдобавок к тем, что поддерживаются в Java.

5.1. Некоторые основные типы

В табл. 5.1 показан ряд основных типов, используемых в Scala, а также диапазоны значений, которые могут принимать их экземпляры. В совокупности типы `Byte`, `Short`, `Int`, `Long` и `Char` называются *целочисленными*. Целочисленные типы плюс `Float` и `Double` называются *числовыми*.

Таблица 5.1. Некоторые основные типы

Основной тип	Диапазон
Byte	8-битовое знаковое целое число в дополнительном коде (от -2^7 до $2^7 - 1$ включительно)
Short	16-битовое знаковое целое число в дополнительном коде (от -2^{15} до $2^{15} - 1$ включительно)

Продолжение

Таблица 5.1 *(продолжение)*

Основной тип	Диапазон
Int	32-битовое знаковое целое число в дополнительном коде (от -2^{31} до $2^{31} - 1$ включительно)
Long	64-битовое знаковое целое число в дополнительном коде (от -2^{63} до $2^{63} - 1$ включительно)
Char	16-битовый беззнаковый Unicode-сивол (от 0 до $2^{16} - 1$ включительно)
String	Последовательность из Char
Float	32-битовое число с плавающей точкой одинарной точности, которое соответствует стандарту IEEE 754
Double	64-битовое число с плавающей точкой двойной точности, которое соответствует стандарту IEEE 754
Boolean	true или false

За исключением типа `String`, который находится в пакете `java.lang`, все типы, показанные в данной таблице, входят в пакет `scala`[1]. Например, полное имя типа `Int` обозначается `scala.Int`. Но, учитывая, что все элементы пакета `scala` и `java.lang` автоматически импортируются в каждый исходный файл Scala, можно повсеместно использовать только простые имена, то есть имена вида `Boolean`, `Char` или `String`.

Опытные Java-разработчики заметят, что основные типы Scala имеют в точности такие же диапазоны, как и соответствующие им типы в Java. Это позволяет компилятору Scala в создаваемом им байт-коде преобразовывать экземпляры *типов значений* Scala, например `Int` или `Double`, в примитивные типы Java.

5.2. Литералы

Все основные типы, перечисленные в табл. 5.1 (см. выше), можно записать с помощью *литералов*. Литерал представляет собой способ записи постоянного значения непосредственно в коде.

УСКОРЕННЫЙ РЕЖИМ ЧТЕНИЯ ДЛЯ JAVA-ПРОГРАММИСТОВ

Синтаксис большинства литералов, показанных в данном разделе, совпадает с синтаксисом, применяемым в Java, поэтому знатоки Java могут спокойно пропустить практически весь раздел. Отдельные различия, о которых стоит прочитать, касаются используемых в Scala неформатированных строк и символов (рассматриваются в подразделе «Строковые литералы»), а также интерполяции строк. Кроме того, в Scala не поддерживаются восьмеричные литералы, а целочисленные, начинающиеся с нуля, например 031, не проходят компиляцию.

[1] Пакеты, кратко рассмотренные в шаге 1 главы 2, подробнее рассматриваются в главе 13.

Целочисленные литералы

Целочисленные литералы для типов `Int`, `Long`, `Short` и `Byte` используются в двух видах: десятичном и шестнадцатеричном. Способ, применяемый для начала записи целочисленного литерала, показывает основание числа. Если число начинается с `0x` или `0X`, то оно шестнадцатеричное (по основанию 16) и может содержать цифры от 0 до 9, а также буквы от A до F в верхнем или нижнем регистре. Примеры использования выглядят следующим образом:

```
scala> val hex = 0x5
hex: Int = 5

scala> val hex2 = 0x00FF
hex2: Int = 255

scala> val magic = 0xcafebabe
magic: Int = -889275714
```

Обратите внимание на то, что оболочка Scala всегда выводит целочисленные значения в десятичном виде, независимо от формы литерала, которую вы могли задействовать для инициализации этих значений. Таким образом, интерпретатор показывает значение переменной `hex2`, которая была инициализирована с помощью литерала `0x00FF`, как десятичное число `255`. (Разумеется, не нужно все принимать на веру. Хорошим способом начать осваивать язык станет практическая работа с этими инструкциями в интерпретаторе по мере чтения данной главы.) Если цифра, с которой начинается число, не нуль и не имеет никаких других знаков отличия, значит, число десятичное (по основанию 10), например:

```
scala> val dec1 = 31
dec1: Int = 31

scala> val dec2 = 255
dec2: Int = 255

scala> val dec3 = 20
dec3: Int = 20
```

Если целочисленный литерал заканчивается на `L` или `l`, значит, показывает число типа `Long`, в противном случае это число относится к типу `Int`. Посмотрите на примеры целочисленных литералов `Long`:

```
scala> val prog = 0XCAFEBABEL
prog: Long = 3405691582

scala> val tower = 35L
tower: Long = 35

scala> val of = 31l
of: Long = 31
```

Если `Int`-литерал присваивается переменной типа `Short` или `Byte`, то рассматривается как принадлежащий к типу `Short` или `Byte`, если, конечно, его значение находится внутри диапазона, допустимого для данного типа, например:

```
scala> val little: Short = 367
little: Short = 367

scala> val littler: Byte = 38
littler: Byte = 38
```

Литералы чисел с плавающей точкой

Литералы чисел с плавающей точкой состоят из десятичных цифр, которые также могут содержать необязательный символ десятичной точки, и после них может стоять необязательный символ `E` или `e` и экспонента. Посмотрите на примеры литералов чисел с плавающей точкой:

```
scala> val big = 1.2345
big: Double = 1.2345

scala> val bigger = 1.2345e1
bigger: Double = 12.345

scala> val biggerStill = 123E45
biggerStill: Double = 1.23E47
```

Обратите внимание: экспонента означает степень числа 10, на которую умножается остальная часть числа. Следовательно, `1.2345e1` равняется числу 1,2345, *умноженному* на 10^1, то есть получается число 12,345. Если литерал числа с плавающей точкой заканчивается на `F` или `f`, значит, число относится к типу `Float`, в противном случае оно относится к типу `Double`. Дополнительно литералы чисел с плавающей точкой могут заканчиваться на `D` или `d`. Посмотрите на примеры литералов чисел с плавающей точкой:

```
scala> val little = 1.2345F
little: Float = 1.2345

scala> val littleBigger = 3e5f
littleBigger: Float = 300000.0
```

Последнее значение, выраженное как тип `Double`, может также принимать иную форму:

```
scala> val anotherDouble = 3e5
anotherDouble: Double = 300000.0

scala> val yetAnother = 3e5D
yetAnother: Double = 300000.0
```

Символьные литералы

Символьные литералы состоят из любого Unicode-символа, заключенного в одинарные кавычки:

```
scala> val a = 'A'
a: Char = A
```

Помимо того что символ представляется в одинарных кавычках в явном виде, его можно указывать с помощью кода из таблицы символов Unicode. Для этого нужно записать `\u`, после чего указать четыре шестнадцатеричные цифры кода:

```
scala> val d = '\u0041'
d: Char = A

scala> val f = '\u0044'
f: Char = D
```

Такие символы в кодировке Unicode могут появляться в любом месте программы на языке Scala. Например, вы можете набрать следующий идентификатор:

```
scala> val B\u0041\u0044 = 1
BAD: Int = 1
```

Он рассматривается точно так же, как идентификатор `BAD`, являющийся результатом раскрытия символов в кодировке Unicode в показанном ранее коде. По сути, в именовании идентификаторов подобным образом нет ничего хорошего, поскольку их трудно прочесть. Иногда с помощью этого синтаксиса исходные файлы Scala, которые содержат отсутствующие в таблице ASCII символы из таблицы Unicode, можно представить в кодировке ASCII.

И наконец, нужно упомянуть о нескольких символьных литералах, представленных специальными управляющими последовательностями (escape sequences), показанными в табл. 5.2, например:

```
scala> val backslash = '\\'
backslash: Char = \
```

Таблица 5.2. Управляющие последовательности специальных символьных литералов

Литерал	Предназначение
\n	Перевод строки (\u000A)
\b	Возврат на одну позицию (\u0008)
\t	Табуляция (\u0009)
\f	Перевод страницы (\u000C)
\r	Возврат каретки (\u000D)
\”	Двойная кавычка (\u0022)
\’	Одинарная кавычка (\u0027)
\\	Обратный слеш (\u005C)

Строковые литералы

Строковый литерал состоит из символов, заключенных в двойные кавычки:

```
scala> val hello = "hello"
hello: String = hello
```

Синтаксис символов внутри кавычек такой же, как и в символьных литералах, например:

```
scala> val escapes = "\\\"\'"
escapes: String = \"'
```

Данный синтаксис неудобен для строк, в которых содержится множество управляющих последовательностей, или для строк, не умещающихся в одну строку текста, поэтому для *неформатированных строк* в Scala включен специальный синтаксис. Неформатированная строка начинается и заканчивается тремя идущими подряд двойными кавычками (`"""`). Внутри нее могут содержаться любые символы, включая символы новой строки, кавычки и специальные символы, за исключением, разумеется, трех кавычек подряд. Например, следующая программа выводит сообщение, используя неформатированную строку:

```
println("""Welcome to Ultamix 3000.
           Type "HELP" for help.""")
```

Но при запуске этого кода получается не совсем то, что хотелось:

```
Welcome to Ultamix 3000.
           Type "HELP" for help.
```

Проблема во включении в строку пробелов перед второй строкой текста! Чтобы справиться с этой весьма часто возникающей ситуацией, вы можете вызывать в отношении строк метод `stripMargin`. Чтобы им воспользоваться, поставьте символ вертикальной черты (`|`) перед каждой строкой текста, а затем в отношении всей строки вызовите метод `stripMargin`:

```
println("""|Welcome to Ultamix 3000.
           |Type "HELP" for help.""".stripMargin)
```

Вот теперь код ведет себя подобающим образом:

```
Welcome to Ultamix 3000.
Type "HELP" for help.
```

Литералы обозначений

Литерал обозначения записывается как `'`*`ident`*, где *`ident`* может быть любым буквенно-цифровым идентификатором. Такие литералы отображаются на экземпляры предопределенного класса `scala.Symbol`. Например, литерал `'cymbal` будет расширен компилятором в вызов фабричного метода `Symbol("cymbal")`. К литералам обозначений обычно прибегают в ситуациях, при которых в динамически типи-

зированных языках вы бы воспользовались просто идентификатором. Например, может потребоваться определить метод, обновляющий запись в базе данных:

```
scala> def updateRecordByName(r: Symbol, value: Any) = {
         // Сюда помещается код
       }
updateRecordByName: (Symbol,Any)Unit
```

Метод получает в качестве параметров обозначение, указывающее на имя поля, в которое ведется запись, и значение, которым это поле будет обновлено в записи. В динамически типизированных языках данную операцию можно вызвать, передавая методу необъявленный идентификатор поля, но в Scala такой код не пройдет компиляцию:

```
scala> updateRecordByName(favoriteAlbum, "OK Computer")
<console>:6: error: not found: value favoriteAlbum
       updateRecordByName(favoriteAlbum, "OK Computer")
                          ^
```

Вместо этого практически с такой же лаконичностью можно передать литерал обозначения:

```
scala> updateRecordByName('favoriteAlbum, "OK Computer")
```

С обозначением, кроме как узнать его имя, сделать ничего нельзя:

```
scala> val s = 'aSymbol
s: Symbol = 'aSymbol

scala> val nm = s.name
nm: String = aSymbol
```

Следует также заметить, что обозначения характеризуются *изолированностью*. Если набрать один и тот же литерал обозначения дважды, то оба выражения будут ссылаться на один и тот же `Symbol`-объект.

Булевы литералы

У типа `Boolean` имеется два литерала, `true` и `false`:

```
scala> val bool = true
bool: Boolean = true

scala> val fool = false
fool: Boolean = false
```

Вот, собственно, и все. Теперь вы буквально (или литерально)[1] стали большим специалистом по Scala.

[1] Образно говоря.

5.3. Интерполяция строк

В Scala включен довольно гибкий механизм для интерполяции строк, позволяющий вставлять выражения в строковые литералы. В самом распространенном случае использования этот механизм предоставляет лаконичную и удобочитаемую альтернативу конкатенации строк. Рассмотрим пример:

```
val name = "reader"
println(s"Hello, $name!")
```

Выражение `s"Hello, $name!"` — *обрабатываемый* строковый литерал. Поскольку за буквой `s` стоит открывающая кавычка, то Scala для обработки литерала воспользуется *интерполятором строк* `s`. Он станет вычислять каждое встроенное выражение, вызывая в отношении каждого результата метод `toString` и заменяя встроенные выражения в литерале этими результатами. Таким образом, из `s"Hello, $name!"` получится `"Hello, reader!"`, то есть точно такой же результат, как при использовании кода `"Hello, " + name + "!"`.

После знака доллара (`$`) в обрабатываемом строковом литерале можно указать любое выражение. Для выражений с одной переменной зачастую можно просто поместить после знака доллара имя этой переменной. Все символы, вплоть до первого символа, не относящегося к идентификатору, Scala будет интерпретировать как выражение. Если в него включены символы, не являющиеся идентификаторами, то это выражение следует заключить в фигурные скобки, а открывающая фигурная скобка должна ставиться сразу же после знака доллара, например:

```
scala> s"The answer is ${6 * 7}."
res0: String = The answer is 42.
```

Scala содержит еще два интерполятора строк: `raw` и `f`. Интерполятор строк `raw` ведет себя практически так же, как и `s`, за исключением того, что не распознает управляющие последовательности символьных литералов (те самые, которые показаны в табл. 5.2). Например, следующая инструкция выводит четыре, а не два обратных слеша:

```
println(raw"No\\\\escape!") // Выводит: No\\\\escape!
```

Интерполятор строк `f` позволяет прикреплять к встроенным выражениям инструкции форматирования в стиле функции `printf`. Инструкции ставятся после выражения и начинаются со знака процента (`%`), при этом используется синтаксис, заданный классом `java.util.Formatter`. Например, вот как можно было бы отформатировать число π:

```
scala> f"${math.Pi}%.5f"
res1: String = 3.14159
```

Если для встроенного выражения не указать никаких инструкций форматирования, то интерполятор строк `f` по умолчанию превратится в `%s`, что означает под-

становку значения, полученного в результате выполнения метода `toString`, точно так же, как это делает интерполятор строк `s`, например:

```
scala> val pi = "Pi"
pi: String = Pi

scala> f"$pi is approximately ${math.Pi}%.8f."
res2: String = Pi is approximately 3.14159265.
```

В Scala интерполяция строк реализуется перезаписью кода в ходе компиляции. Компилятор в качестве выражения интерполятора строк будет рассматривать любое выражение, состоящее из идентификатора, за которым сразу же стоит открывающая двойная кавычка строкового литерала. Интерполяторы строк `s`, `f` и `raw` реализуются с помощью этого общего механизма. Библиотеки и пользователи могут определять другие интерполяторы строк, применяемые в иных целях.

5.4. Все операторы являются методами

Для основных типов Scala предоставляет весьма богатый набор операторов. Как упоминалось в предыдущих главах, эти операторы — всего лишь приятный синтаксис для обычных вызовов методов. Например, `1 + 2` означает то же самое, что и `1.+(2)`. Иными словами, в классе `Int` имеется метод по имени `+`, который получает `Int`-значение и возвращает `Int`-результат. Он вызывается при сложении двух `Int`-значений:

```
scala> val sum = 1 + 2 // Scala вызывает 1.+(2)
sum: Int = 3
```

Чтобы убедиться в этом, можете набрать выражение, в точности соответствующее вызову метода:

```
scala> val sumMore = 1.+(2)
sumMore: Int = 3
```

Фактически в классе `Int` содержится несколько *перегруженных* методов `+`, получающих различные типы параметров[1]. Например, у `Int` есть еще один метод, тоже по имени `+`, который получает и возвращает значения типа `Long`. При сложении `Long` и `Int` будет вызван именно этот альтернативный метод:

```
scala> val longSum = 1 + 2L // Scala вызывает 1.+(2L)
longSum: Long = 3
```

Символ `+` — оператор, точнее, инфиксный оператор. Форма записи операторов не ограничивается методами, подобными `+`, которые в других языках выглядят как

[1] Перегруженные методы имеют точно такие же имена, но используют другие типы аргументов. Более подробно перегрузка методов рассматривается в разделе 6.11.

операторы. *Любой* метод может использоваться в нотации операторов. Например, в классе `String` есть метод `indexOf`, получающий один параметр типа `Char`. Метод `indexOf` ведет поиск первого появления в строке указанного символа и возвращает его индекс или `-1`, если символ найден не будет. Метод `indexOf` можно использовать как оператор:

```
scala> val s = "Hello, world!"
s: String = Hello, world!

scala> s indexOf 'o' // Scala вызывает s.indexOf('o')
res0: Int = 4
```

Кроме того, в классе `String` предлагается перегруженный метод `indexOf`, получающий два параметра: символ, поиск которого будет вестись, и индекс, с которого нужно начинать поиск. (Другой метод, показанный ранее `indexOf`, начинает поиск с нулевого индекса, то есть с начала строки `String`.) Несмотря на то что данный метод `indexOf` получает два аргумента, его также можно применять, используя форму записи операторов. Но когда с использованием формы записи операторов вызывается метод, получающий несколько аргументов, последние нужно заключать в круглые скобки. Например, вот каким образом эта другая форма метода `indexOf` используется в качестве оператора (в продолжение предыдущего примера):

```
scala> s indexOf ('o', 5) // Scala вызывает s.indexOf('o', 5)
res1: Int = 8
```

Оператором может быть любой метод

Операторы в Scala не являются специальным синтаксисом языка, оператором может быть любой метод. Метод превращается в оператор в зависимости от способа его применения. Когда записывается код `s.indexOf('o')`, метод `indexOf` не является оператором. Но, когда запись принимает вид `s indexOf 'o'`, этот метод становится оператором, поскольку при его использовании применяется форма записи операторов.

До сих пор рассматривались только примеры *инфиксной* формы записи операторов, означающей, что вызываемый метод находится между объектом и параметром или параметрами, которые нужно передать методу, как в выражении `7 + 2`. В Scala также имеются две другие формы записи операторов: префиксная и постфиксная. В префиксной форме записи имя метода ставится перед объектом, в отношении которого вызывается этот метод (например, `-` в выражении `-7`). В постфиксной форме имя метода ставится после объекта (например, `toLong` в выражении `7 toLong`).

В отличие от инфиксной формы записи, в которой операторы получают два операнда (один слева, другой справа), префиксные и постфиксные операторы являются *унарными* — получают только один операнд. В префиксной форме записи операнд размещается справа от оператора. В качестве примеров можно привести выражения `-2.0`, `!found` и `~0xFF`. Как и в случае использования инфиксных опера-

торов, эти префиксные операторы являются сокращенной формой вызова методов. Но в данном случае перед символом оператора в имени метода ставится приставка `unary_`. Например, Scala превратит выражение `-2.0` в вызов метода `(2.0).unary_-`. Вы можете убедиться в этом, набрав вызов метода как с использованием формы записи операторов, так и в явном виде:

```
scala> -2.0 // Scala вызывает (2.0).unary_-
res2: Double = -2.0

scala> (2.0).unary_-
res3: Double = -2.0
```

Идентификаторами, которые могут служить в качестве префиксных операторов, являются только `+`, `-`, `!` и `~`. Следовательно, если вы определите метод по имени `unary_!`, то сможете вызвать его в отношении значения или переменной подходящего типа, прибегнув к префиксной форме записи операторов, например, `!p`. Но определив метод по имени `unary_*`, вы не сможете использовать префиксную форму записи операторов, поскольку `*` не входит в число четырех идентификаторов, которые могут использоваться в качестве префиксных операторов. Метод можно вызвать обычным способом как `p.unary_*`, но при попытке вызвать его в виде `*p` Scala воспримет код так, словно он записан в виде `*.p`, что, вероятно, совершенно не совпадает с задуманным![1]

Постфиксные операторы, будучи вызванными без точки или круглых скобок, являются методами, не получающими аргументов. В Scala при вызове метода пустые круглые скобки можно не ставить. Соглашение гласит, что круглые скобки ставятся в том случае, если метод имеет побочные эффекты, как в случае с методом `println()`. Но их можно не ставить, если метод не имеет побочных эффектов, как в случае с методом `toLowerCase`, вызываемым в отношении значения типа `String`:

```
scala> val s = "Hello, world!"
s: String = Hello, world!

scala> s.toLowerCase
res4: String = hello, world!
```

В последнем случае, где методу не требуются аргументы, можно при желании не ставить точку и воспользоваться постфиксной формой записи операторов:

```
scala> s toLowerCase
res5: String = hello, world!
```

Здесь метод `toLowerCase` используется в качестве постфиксного оператора в отношении операнда `s`.

Чтобы понять, какие операторы можно использовать с основными типами Scala, нужно посмотреть на методы, объявленные в классах типов, в документации

[1] Однако не обязательно все будет потеряно. Есть весьма незначительная вероятность того, что программа с кодом *p может скомпилироваться как код C++.

по Scala API. Но данная книга — учебник по языку Scala, поэтому в нескольких следующих разделах будет представлен краткий обзор большинства этих методов.

УСКОРЕННЫЙ РЕЖИМ ЧТЕНИЯ ДЛЯ JAVA-ПРОГРАММИСТОВ

Многие аспекты Scala, рассматриваемые в оставшейся части главы, совпадают с аналогичными Java-аспектами. Если вы хорошо разбираетесь в Java и у вас мало времени, то можете спокойно перейти к разделу 5.8, в котором рассматриваются отличия Scala от Java в области равенства объектов.

5.5. Арифметические операции

Арифметические методы при работе с любыми числовыми типами можно вызвать в инфиксной форме для сложения (`+`), вычитания (`-`), умножения (`*`), деления (`/`) и получения остатка от деления (`%`). Вот несколько примеров:

```
scala> 1.2 + 2.3
res6: Double = 3.5

scala> 3 - 1
res7: Int = 2

scala> 'b' - 'a'
res8: Int = 1

scala> 2L * 3L
res9: Long = 6

scala> 11 / 4
res10: Int = 2

scala> 11 % 4
res11: Int = 3

scala> 11.0f / 4.0f
res12: Float = 2.75

scala> 11.0 % 4.0
res13: Double = 3.0
```

Когда целочисленными типами являются как правый, так и левый операнды (`Int`, `Long`, `Byte`, `Short` или `Char`), оператор `/` выведет всю числовую часть результата деления, исключая остаток. Оператор `%` показывает остаток от предполагаемого целочисленного деления.

Остаток от деления числа с плавающей точкой, полученный с помощью метода `%`, не определен в стандарте IEEE 754. Что касается операции вычисления остатка, в этом стандарте используется деление с округлением, а не деление с отбрасыванием остатка. Поэтому данная операция сильно отличается от опе-

рации вычисления остатка от целочисленного деления. Если все-таки нужно получить остаток по стандарту IEEE 754, то можно вызвать метод `IEEEremainder` из `scala.math`:

```
scala> math.IEEEremainder(11.0, 4.0)
res14: Double = -1.0
```

Числовые типы также предлагают прибегнуть к унарным префиксным операторам `+` (метод `unary_+`) и `-` (метод `unary_-`), позволяющим показать положительное или отрицательное значение числового литерала, как в `-3` или `+4.0`. Если не указать унарный `+` или `-`, то литерал интерпретируется как положительный. Унарный `+` существует исключительно для симметрии с унарным `-`, однако не производит никаких действий. Унарный `-` может также использоваться для смены знака переменной. Вот несколько примеров:

```
scala> val neg = 1 + -3
neg: Int = -2

scala> val y = +3
y: Int = 3

scala> -neg
res15: Int = 2
```

5.6. Отношения и логические операции

Числовые типы можно сравнивать с помощью методов отношений «больше» (`>`), «меньше» (`<`), «больше или равно» (`>=`) и «меньше или равно» (`<=`), которые выдают в качестве результата булево значение. Дополнительно, чтобы инвертировать булево значение, можно использовать унарный оператор `!` (метод `unary_!`). Вот несколько примеров:

```
scala> 1 > 2
res16: Boolean = false

scala> 1 < 2
res17: Boolean = true

scala> 1.0 <= 1.0
res18: Boolean = true

scala> 3.5f >= 3.6f
res19: Boolean = false

scala> 'a' >= 'A'
res20: Boolean = true

scala> val untrue = !true
untrue: Boolean = false
```

Методы «логическое “И”» (`&&` и `&`) и «логическое “ИЛИ”» (`||` и `|`) получают операнды типа `Boolean` в инфиксной нотации и выдают результат в виде `Boolean`-значения, например:

```
scala> val toBe = true
toBe: Boolean = true

scala> val question = toBe || !toBe
question: Boolean = true

scala> val paradox = toBe && !toBe
paradox: Boolean = false
```

Операции `&&` и `||`, как и в Java, — *сокращенно вычисляемые*: выражения, построенные с помощью этих операторов, вычисляются, только когда это нужно для определения результата. Иными словами, правая часть выражений с использованием `&&` и `||` не будет вычисляться, если результат уже определился при вычислении левой части. Например, если левая часть выражения с методом `&&` вычисляется в `false`, то результатом выражения, несомненно, будет `false`, поэтому правая часть не вычисляется. Аналогично этому, если левая часть выражения с методом `||` вычисляется в `true`, то результатом выражения, конечно же, будет `true`, поэтому правая часть не вычисляется:

```
scala> def salt() = { println("salt"); false }
salt: ()Boolean

scala> def pepper() = { println("pepper"); true }
pepper: ()Boolean

scala> pepper() && salt()
pepper
salt
res21: Boolean = false

scala> salt() && pepper()
salt
res22: Boolean = false
```

В первом выражении вызываются `pepper` и `salt`, но во втором вызывается только `salt`. Поскольку `salt` возвращает `false`, то необходимость в вызове `pepper` отпадает.

Если правую часть нужно вычислить при любых условиях, то вместо показанных выше методов следует обратиться к методам `&` и `|`. Первый выполняет логическую операцию «И», а второй — операцию «ИЛИ», но при этом они не прибегают к сокращенному вычислению, как этот делают методы `&&` и `||`. Вот как выглядит пример их использования:

```
scala> salt() & pepper()
salt
pepper
res23: Boolean = false
```

ПРИМЕЧАНИЕ

Возникает вопрос: как сокращенное вычисление может работать, если операторы — всего лишь методы? Обычно все аргументы вычисляются перед входом в метод, тогда каким же образом метод может избежать вычисления своего второго аргумента? Дело в том, что у всех методов Scala есть средство для задержки вычисления его аргументов или даже полной его отмены. Оно называется «параметр, передаваемый по имени» и будет рассмотрено в разделе 9.5.

5.7. Поразрядные операции

Scala позволяет выполнять операции над отдельными разрядами целочисленных типов, используя несколько поразрядных методов. К таким методам относятся поразрядное «И» (`&`), поразрядное «ИЛИ» (`|`) и поразрядное исключающее «ИЛИ» (`^`)[1]. Унарный поразрядный оператор дополнения (`~`, метод `unary_~`) инвертирует каждый разряд в своем операнде, например:

```
scala> 1 & 2
res24: Int = 0

scala> 1 | 2
res25: Int = 3

scala> 1 ^ 3
res26: Int = 2

scala> ~1
res27: Int = -2
```

В первом выражении, `1 & 2`, выполняется поразрядное И над каждым разрядом чисел `1` (`0001`) и `2` (`0010`) и выдается результат `0` (`0000`). Во втором выражении, `1 | 2`, выполняется поразрядное «ИЛИ» над теми же операндами и выдается результат `3` (`0011`). В третьем выражении, `1 ^ 3`, выполняется поразрядное исключающее «ИЛИ» над каждым разрядом `1` (`0001`) и `3` (`0011`) и выдается результат `2` (`0010`). В последнем выражении, `~1`, инвертируется каждый разряд в `1` (`0001`) и выдается результат `-2`, который в двоичной форме выглядит как `11111111111111111111111111111110`.

Целочисленные типы Scala также предлагают три метода сдвига: влево (`<<`), вправо (`>>`) и беззнаковый сдвиг вправо (`>>>`). Методы сдвига, примененные в инфиксной форме записи операторов, сдвигают разряды целочисленного значения, указанные слева от оператора, на количество разрядов, указанное в целочисленном значении справа от оператора. При сдвиге влево и беззнаковом сдвиге вправо разряды по мере сдвига заполняются нулями. При сдвиге вправо разряды указанного

[1] Метод поразрядного исключающего «ИЛИ» выполняет соответствующую операцию в отношении своих операндов. Из одинаковых разрядов получается 0, а из разных — 1. Следовательно, выражение 0011 ^ 0101 вычисляется в 0110.

слева значения по мере сдвига заполняются значением самого старшего разряда (разряда знака). Вот несколько примеров:

```
scala> -1 >> 31
res28: Int = -1

scala> -1 >>> 31
res29: Int = 1

scala> 1 << 2
res30: Int = 4
```

Число –1 в двоичном виде выглядит как 11111111111111111111111111111111. В первом примере, `-1 >> 31`, в числе `-1` происходит сдвиг вправо на 31 разрядную позицию. В значении типа `Int` содержатся 32 разряда, поэтому данная операция по сути перемещает самый левый разряд до тех пор, пока тот не станет самым правым[1]. Поскольку метод `>>` выполняет заполнение по мере сдвига единицами ввиду того, что самый левый разряд числа –1 — это 1, результат получается идентичным исходному левому операнду и состоит из 32 единичных разрядов или равняется –1. Во втором примере, `-1 >>> 31`, самый левый разряд опять сдвигается вправо в самую правую позицию, однако на сей раз освобождающиеся разряды заполняются нулями. Поэтому результат в двоичном виде получается 00000000000000000000000000000001, или 1. В последнем примере, `1 << 2`, левый операнд, `1`, сдвигается влево на две позиции (освобождающиеся позиции заполняются нулями), в результате чего в двоичном виде получается число 00000000000000000000000000000100, или 4.

5.8. Равенство объектов

Если нужно сравнить два объекта на равенство, то можно воспользоваться либо методом `==`, либо его противоположностью — методом `!=`. Вот несколько простых примеров:

```
scala> 1 == 2
res31: Boolean = false

scala> 1 != 2
res32: Boolean = true

scala> 2 == 2
res33: Boolean = true
```

По сути, эти две операции применимы ко всем объектам, а не только к основным типам. Например, оператор `==` можно использовать для сравнения списков:

```
scala> List(1, 2, 3) == List(1, 2, 3)
res34: Boolean = true
```

[1] Самый левый разряд в целочисленном типе является знаковым. Если крайний слева разряд установлен в 1, значит, число отрицательное. Если он в 0, то положительное.

```
scala> List(1, 2, 3) == List(4, 5, 6)
res35: Boolean = false
```

Если пойти еще дальше, то можно сравнить два объекта, имеющих разные типы:

```
scala> 1 == 1.0
res36: Boolean = true

scala> List(1, 2, 3) == "hello"
res37: Boolean = false
```

Можно даже выполнить сравнение со значением `null` или с тем, что может иметь данное значение. Никакие исключения при этом выдаваться не будут:

```
scala> List(1, 2, 3) == null
res38: Boolean = false

scala> null == List(1, 2, 3)
res39: Boolean = false
```

Как видите, оператор `==` реализован весьма искусно, и вы в большинстве случаев получите то сравнение на равенство, которое вам нужно. Все делается по очень простому правилу: сначала левая часть проверяется на `null`. Если ее значение не `null`, то вызывается метод `equals`. Ввиду того что `equals` — метод, точность получаемого сравнения зависит от типа левого аргумента. Проверка на `null` выполняется автоматически, поэтому вам не нужно проводить ее[1].

Этот вид сравнения выдает `true` в отношении различных объектов, если их содержимое одинаково и их методы `equals` созданы на основе проверки содержимого. Например, вот как сравниваются две строки, в которых по пять одинаковых букв:

```
scala> ("he" + "llo") == "hello"
res40: Boolean = true
```

Чем оператор == в Scala отличается от аналогичного оператора в Java

В Java оператор `==` может использоваться для сравнения как примитивных, так и ссылочных типов. В отношении примитивных типов оператор `==` в Java проверяет равенство значений, как и в Scala. Но в отношении ссылочных типов оператор `==` в Java проверяет *равенство ссылок*. Это значит, две переменные указывают на один и тот же объект в куче, принадлежащей JVM. Scala также предоставляет средство `eq` для сравнения равенства ссылок. Но метод `eq` и его противоположность, метод `ne`, применяются только к объектам, которые непосредственно отображаются на объекты Java. Исчерпывающие подробности о `eq` и `ne` приводятся в разделах 11.1 и 11.2. Кроме того, в главе 30 показано, как создавать хорошие методы `equals`.

[1] Автоматическая проверка игнорирует правую сторону, но любой корректно реализованный метод equals должен возвращать false, если его аргумент имеет значение null.

5.9. Приоритет и ассоциативность операторов

Приоритет операторов определяет, какая часть выражения вычисляется самой первой. Например, выражение `2 + 2 * 7` вычисляется как `16`, а не как `28`, поскольку оператор `*` имеет более высокий приоритет, чем оператор `+`. Поэтому та часть выражения, в которой требуется перемножить числа, вычисляется до того, как будет выполнена часть, в которой числа складываются. Разумеется, чтобы уточнить в выражении порядок вычисления или переопределить приоритеты, можно воспользоваться круглыми скобками. Например, если вы действительно хотите, чтобы результат вычисления ранее показанного выражения был `28`, то можете набрать следующее выражение:

```
(2 + 2) * 7
```

Если учесть, что в Scala, по сути, нет операторов, а есть только способ применения методов в форме записи операторов, то возникает вопрос: а как тогда работает приоритет операторов? Scala принимает решение о приоритете на основе первого символа метода, использованного в форме записи операторов (из этого правила есть одно исключение, рассматриваемое ниже). Если имя метода начинается, к примеру, с `*`, то он получит более высокий приоритет, чем метод, чье имя начинается на `+`. Следовательно, выражение `2 + 2 * 7` будет вычислено как `2 + (2 * 7)`. Аналогично этому выражение `a +++ b *** c`, в котором `a`, `b` и `c` — переменные, а `+++` и `***` — методы, будет вычислено как `a +++ (b *** c)`, поскольку метод `***` обладает более высоким уровнем приоритета, чем метод `+++`.

Таблица 5.3. Приоритет операторов

(Все специальные символы)
* / %
+ –
:
= !
< >
&
^
|
(Все буквы)
(Все операторы присваивания)

В табл. 5.3 показан приоритет применительно к первому символу метода в убывающем порядке, где символы, показанные на одной строке, определяют оди-

наковый уровень приоритета. Чем выше символ в списке, тем выше приоритет начинающегося с него метода. Вот пример, показывающий влияние приоритета:

```
scala> 2 << 2 + 2
res41: Int = 32
```

Имя метода `<<` начинается с символа `<`, который появляется в приведенном списке ниже символа `+` — первого и единственного символа метода `+`. Следовательно, `<<` будет иметь более низкий уровень приоритета, чем `+`, и выражение будет вычислено путем вызова сначала метода `+`, а затем метода `<<`, как в выражении `2 << (2 + 2)`. При сложении `2 + 2` в результате математического действия получается `4`, а вычисление выражения `2 << 4` дает результат `32`. Если поменять операторы местами, то будет получен другой результат:

```
scala> 2 + 2 << 2
res42: Int = 16
```

Поскольку первые символы по сравнению с предыдущим примером не изменились, то методы будут вызваны в том же порядке: `+`, а затем `<<`. Следовательно, `2 + 2` опять вычислится как `4`, а `4 << 2` даст результат `16`.

Единственное исключение из правил, о существовании которого уже говорилось, относится к *операторам присваивания*, заканчивающихся знаком равенства. Если оператор заканчивается знаком равенства (`=`) и не относится к одному из операторов сравнения (`<=`, `>=`, `==` и `!=`), то приоритет оператора имеет такой же уровень, что и простое присваивание (`=`). То есть он ниже приоритета любого другого оператора. Например: `x *= y + 1` означает то же самое, что и `x *= (y + 1)`. поскольку оператор `*=` классифицируется как оператор присваивания, приоритет которого ниже, чем у `+`, даже притом, что первым символом оператора выступает знак `*`, который обозначил бы приоритет выше, чем у `+`.

Если в выражении рядом появляются операторы с одинаковым уровнем приоритета, то способ группировки операторов определяется их *ассоциативностью*. Ассоциативность оператора в Scala определяется по его последнему символу. Как уже упоминалось в главе 3, любой метод, имя которого заканчивается символом `:`, вызывается в отношении своего правого операнда с передачей ему левого. Методы, в окончании имени которых используются любые другие символы, действуют наоборот: они вызываются в отношении своего левого операнда с передачей себе правого. То есть из выражения `a * b` получается `a.*(b)`, но из `a ::: b` получается `b.:::(a)`.

Но независимо от того, какова ассоциативность оператора, его операнды всегда вычисляются слева направо. Следовательно, если `a` — выражение, не являющееся простой ссылкой на неизменяемое значение, то выражение `a ::: b` при более точном рассмотрении представляется следующим блоком:

```
{ val x = a; b.:::(x) }
```

В этом блоке `a` по-прежнему вычисляется раньше `b`, а затем результат данного вычисления передается в качестве операнда принадлежащему `b` методу `:::`.

Это правило ассоциативности играет роль также при появлении в одном выражении рядом сразу нескольких операторов с одинаковым уровнем приоритета.

Если имена методов заканчиваются на `:`, они группируются справа налево, в противном случае — слева направо. Например, `a ::: b ::: c` рассматривается как `a ::: (b ::: c)`. Но `a * b * c`, в отличие от этого, рассматривается как `(a * b) * c`.

Правила приоритета операторов — часть языка Scala, и вам не следует бояться применять ими. При этом, чтобы прояснить первоочередность использования операторов, в некоторых выражениях все же лучше прибегнуть к круглым скобкам. Пожалуй, единственное, на что можно реально рассчитывать в отношении знания порядка приоритета другими программистами, — то, что мультипликативные операторы `*`, `/` и `%` имеют более высокий уровень приоритета, чем аддитивные `+` и `-`. Таким образом, даже если выражение `a + b << c` выдает нужный результат и без круглых скобок, стоит внести дополнительную ясность с помощью записи `(a + b) << c`. Это снизит количество нелестных отзывов ваших коллег по поводу использованной вами формы записи операторов, которое выражается, к примеру, в недовольном восклицании вроде «Опять в его коде невозможно разобраться!» и отправке вам сообщения наподобие bills !*&^%~ code![1].

5.10. Обогащающие оболочки

В отношении основных типов Scala можно вызвать намного больше методов, чем рассмотрено в предыдущих разделах. Некоторые примеры показаны в табл. 5.4. Эти методы становятся доступными при использовании *неявного преобразования* — технологии, подробное описание которой будет дано в главе 21. А пока вам нужно знать лишь то, что для каждого основного типа, рассмотренного в текущей главе, существует «обогащающая оболочка», которая предоставляет ряд дополнительных методов. Поэтому увидеть все доступные методы, применяемые в отношении основных типов, можно, обратившись к документации по API, которая касается обогащающей оболочки для каждого основного типа. Эти классы перечислены в табл. 5.5.

Таблица 5.4. Некоторые обогащающие операции

Код	Результат
0 max 5	5
0 min 5	0
–2.7 abs	2.7
–2.7 round	–3L
1.5 isInfinity	False
(1.0 / 0) isInfinity	True

[1] Теперь вы уже знаете, что, получив такой код, компилятор Scala создаст вызов (bills.!*&^%~(code)).!().

Код	Результат
4 to 6	Range(4, 5, 6)
"bob" capitalize	"Bob"
"robert" drop 2	"bert"

Таблица 5.5. Классы обогащающих оболочек

Основной тип	Обогащающая оболочка
Byte	scala.runtime.RichByte
Short	scala.runtime.RichShort
Int	scala.runtime.RichInt
Long	scala.runtime.RichLong
Char	scala.runtime.RichChar
Float	scala.runtime.RichFloat
Double	scala.runtime.RichDouble
Boolean	scala.runtime.RichBoolean
String	scala.collection.immutable.StringOps

Резюме

Основное, что следует усвоить, прочитав данную главу, — операторы в Scala являются вызовами методов и для основных типов Scala существуют неявные преобразования в «обогащенные» варианты, которые добавляют дополнительные полезные методы. В главе 6 мы покажем, что означает конструирование объектов в функциональном стиле, обеспечивающее новые реализации некоторых операторов, рассмотренных в настоящей главе.

6 Функциональные объекты

Усвоив основы, рассмотренные в предыдущих главах, вы готовы разработать больше полнофункциональных классов Scala. В этой главе основное внимание мы уделим классам, определяющим функциональные объекты или объекты, не имеющие никакого изменяемого состояния. Запуская примеры, мы создадим несколько вариантов класса, моделирующего рациональные числа в виде неизменяемых объектов. Попутно будут показаны дополнительные аспекты объектно-ориентированного программирования на Scala: параметры класса и конструкторы, методы и операторы, приватные члены, переопределение, проверка соблюдения предварительных условий, перегрузка и рекурсивные ссылки.

6.1. Спецификация класса Rational

Рациональным называется число, которое может быть выражено соотношением `n / d`, где `n` и `d` представлены целыми числами, за исключением того, что `d` не может быть нулем. Здесь `n` называется *числителем*, а `d` — *знаменателем*. Примерами рациональных чисел могут послужить 1/2, 2/3, 112/239 и 2/1. В сравнении с числами с плавающей точкой рациональные числа имеют то преимущество, что дроби представлены точно, без округлений или приближений.

Разрабатываемый в этой главе класс должен моделировать поведение рациональных чисел, позволяя производить над ними действия по сложению, вычитанию умножению и делению. Для сложения двух рациональных чисел сначала нужно получить общий знаменатель, после чего сложить два числителя. Например, чтобы выполнить сложение 1/2 + 2/3, обе части левого операнда умножаются на 3, а обе части правого операнда — на 2, в результате чего получается 3/6 + 4/6. Сложение двух числителей дает результат 7/6. Для перемножения двух рациональных чисел можно просто перемножить их числители, а затем знаменатели. Таким образом, 1/2 · 2/5 дает число 2/10, которое можно представить более кратко в нормализованном виде как 1/5. Деление выполняется путем перестановки местами числителя и знаменателя правого операнда с последующим перемножением чисел. Например, 1/2 / 3/5 — то же самое, что и 1/2 · 5/3, в результате получается число 5/6.

Одно, возможно, очевидное наблюдение заключается в том, что в математике рациональные числа не имеют изменяемого состояния. Можно сложить два рациональных числа, и результатом будет новое рациональное число. Исходные числа не будут изменены. Неизменяемый класс `Rational`, разрабатываемый в данной главе, будет иметь такое же свойство. Каждое рациональное число будет представлено одним объектом `Rational`. При сложении двух объектов `Rational` для хранения суммы будет создаваться новый объект `Rational`.

В этой главе мы представим некоторые допустимые в Scala способы написания библиотек, которые создают впечатление, будто используется поддержка, присущая непосредственно самому языку программирования. Например, в конце этой главы вы сможете сделать с классом `Rational` следующее:

```
scala> val oneHalf = new Rational(1, 2)
oneHalf: Rational = 1/2

scala> val twoThirds = new Rational(2, 3)
twoThirds: Rational = 2/3

scala> (oneHalf / 7) + (1 - twoThirds)
res0: Rational = 17/42
```

6.2. Конструирование класса Rational

Конструирование класса `Rational` неплохо начать с рассмотрения того, как клиенты-программисты будут создавать новый объект `Rational`. Было решено создавать объекты `Rational` неизменяемыми, и потому мы потребуем, чтобы эти клиенты при создании экземпляра предоставляли все необходимые ему данные (в нашем случае числитель и знаменатель). Поэтому начнем конструирование со следующего кода:

```
class Rational(n: Int, d: Int)
```

По поводу этой строки кода в первую очередь следует заметить: если у класса нет тела, то вам не нужно ставить пустые фигурные скобки (конечно, при желании можете поставить). Идентификаторы `n` и `d`, указанные в круглых скобках после имени класса, `Rational`, называются *параметрами класса*. Компилятор Scala подберет эти два параметра и создаст *первичный конструктор*, получающий их же.

Плюсы и минусы неизменяемого объекта

Неизменяемые объекты имеют ряд преимуществ над изменяемыми и один потенциальный недостаток. Во-первых, о неизменяемых объектах проще говорить, чем об изменяемых, поскольку у них нет изменяемых со временем сложных областей состояния. Во-вторых, неизменяемые объекты можно совершенно свободно куда-нибудь передавать, а перед передачей изменяемых объектов в другой код порой приходится делать страховочные копии. В-третьих, если объект правильно сконструирован, то при одновременном обращении к неизменяемому объекту из двух потоков повредить

его состояние невозможно, поскольку никакой поток не может изменить состояние неизменяемого объекта. В-четвертых, неизменяемые объекты обеспечивают безопасность ключей хеш-таблиц. Если, к примеру, изменяемый объект изменился после помещения в `HashSet`, то в следующий раз при поиске там его можно не найти.

Главный недостаток неизменяемых объектов — им иногда требуется копирование больших графов объектов, тогда как вместо этого можно было бы сделать обновление. В некоторых случаях это может быть сложно выразить, а также могут выявиться узкие места в производительности. В результате в библиотеки нередко включают изменяемые альтернативы неизменяемым классам. Например, класс `StringBuilder` — изменяемая альтернатива неизменяемого класса `String`. Дополнительная информация о конструировании изменяемых объектов в Scala будет дана в главе 18.

ПРИМЕЧАНИЕ

Исходный пример с Rational подчеркивает разницу между Java и Scala. В Java классы имеют конструкторы, которые могут принимать параметры, а в Scala классы могут принимать параметры напрямую. Система записи в Scala куда более лаконична — параметры класса могут использоваться напрямую в теле, нет никакой необходимости определять поля и записывать присваивания, копирующие параметры конструктора в поля. Это может привести к дополнительной экономии на шаблонном коде, особенно когда дело касается небольших классов.

Компилятор Scala скомпилирует любой код, помещенный в тело класса и не являющийся частью поля или определения метода, в первичный конструктор. Например, можно вывести такое отладочное сообщение:

```
class Rational(n: Int, d: Int) {
  println("Created " + n + "/" + d)
}
```

Получив данный код, компилятор Scala поместит вызов `println` в первичный конструктор класса `Rational`. Поэтому при создании нового экземпляра `Rational` вызов `println` приведет к выводу отладочного сообщения:

```
scala> new Rational(1, 2)
Created 1/2
res0: Rational = Rational@6121a7dd
```

6.3. Переопределение метода toString

При создании экземпляра `Rational` в предыдущем примере интерпретатор вывел `Rational@2591e0c9`. Эта странная строка получилась ввиду вызова в отношении объекта `Rational` метода `toString`. По умолчанию класс `Rational` наследует реализацию `toString`, определенную в классе `java.lang.Object`, которая просто выводит имя класса, символ `@` и шестнадцатеричное число. Предполагалось, что результат

выполнения `toString` поможет программистам, предоставив информацию, которую можно использовать в отладочных инструкциях вывода информации, для ведения логов, в отчетах о сбоях тестов, а также для просмотра выходной информации интерпретатора и отладчика. Результат, выдаваемый на данный момент методом `toString`, не приносит особой пользы, поскольку не дает никакой информации относительно значения рационального числа. Более полезная реализация `toString` будет выводить значения числителя и знаменателя объекта `Rational`. *Переопределить* исходную реализацию можно, добавив метод `toString` к классу `Rational`:

```
class Rational(n: Int, d: Int) {
  override def toString = s"$n/$d
}
```

Модификатор `override` перед определением метода показывает, что предыдущее определение метода переопределяется (более подробно этот вопрос рассматривается в главе 10). Поскольку отныне числа типа `Rational` будут выводиться совершенно отчетливо, мы удаляем отладочную инструкцию `println`, помещенную в тело предыдущей версии класса `Rational`. Теперь новое поведение `Rational` можно протестировать в интерпретаторе:

```
scala> val x = new Rational(1, 3)
x: Rational = 1/3

scala> val y = new Rational(5, 7)
y: Rational = 5/7
```

6.4. Проверка соблюдения предварительных условий

В качестве следующего шага переключим внимание на проблему, связанную с текущим поведением первичного конструктора. Как упоминалось в начале главы, рациональные числа не должны содержать нуль в знаменателе. Но пока первичный конструктор может принимать нуль, передаваемый в качестве параметра `d`:

```
scala> new Rational(5, 0)
res1: Rational = 5/0
```

Одно из преимуществ объектно-ориентированного программирования — возможность инкапсуляции данных внутри объектов, чтобы можно было гарантировать, что данные корректны в течение всей жизни объекта. В данном случае для такого неизменяемого объекта, как `Rational`, это значит, что вы должны гарантировать корректность данных на этапе конструирования объекта при условии, что нулевой знаменатель — недопустимое состояние числа типа `Rational` и такое число не должно создаваться, если в качестве параметра `d` передается нуль.

Лучше всего решить эту проблему, определив для первичного конструктора *предусловие*, согласно которому `d` должен иметь ненулевое значение. Предусловие — ограничение, накладываемое на значения, передаваемые в метод или

конструктор, то есть требование, которое должно выполняться вызывающим кодом. Один из способов решить задачу — использовать метод `require`[1]:

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  override def toString = s"$n/$d
}
```

Метод `require` получает один булев параметр. Если переданное значение приведет к вычислению в `true`, то из метода `require` произойдет нормальный выход. В противном случае объект не создастся и будет выдано исключение `IllegalArgumentException`.

6.5. Добавление полей

Теперь, когда первичный конструктор выдвигает нужные предусловия, мы переключимся на поддержку сложения. Для этого определим в классе `Rational` публичный метод `add`, получающий в качестве параметра еще одно значение типа `Rational`. Чтобы сохранить неизменяемость класса `Rational`, метод `add` не должен прибавлять переданное рациональное число к объекту, в отношении которого он вызван. Ему нужно создать и вернуть новый объект `Rational`, содержащий сумму. Можно подумать, что метод `add` создается следующим образом:

```
class Rational(n: Int, d: Int) { // Этот код не будет скомпилирован
  require(d != 0)
  override def toString = s"$n/$d
  def add(that: Rational): Rational =
    new Rational(n * that.d + that.n * d, d * that.d)
}
```

Но, получив этот код, компилятор выдаст свои возражения:

```
<console>:11: error: value d is not a member of Rational
           new Rational(n * that.d + that.n * d, d * that.d)
                                ^
<console>:11: error: value d is not a member of Rational
           new Rational(n * that.d + that.n * d, d * that.d)
                                                       ^
```

Хотя параметры `n` и `d` класса находятся в области видимости кода вашего метода `add`, получить доступ к их значениям можно только в объекте, в отношении которого вызван данный метод. Следовательно, когда в реализации последнего указывается `n` или `d`, компилятор рад предоставить вам значения для этих параметров класса. Но он не может позволить указать `that.n` или `that.d`, поскольку они не ссылаются

[1] Метод require определен в самостоятельном объекте Predef. Как упоминалось в разделе 4.4, элементы класса Predef автоматически импортируются в каждый исходный файл Scala.

на объект `Rational`, в отношении которого был вызван метод `add`[1]. Чтобы подобным образом получить доступ к числителю или знаменателю, их нужно сделать полями. Как эти поля можно добавить к классу `Rational`, показано в листинге 6.1[2].

Листинг 6.1. Класс Rational с полями

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  val numer: Int = n
  val denom: Int = d
  override def toString = s"$numer/$denom"
  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
```

В версии `Rational`, показанной в листинге, добавлены два поля с именами `numer` и `denom`, которые были проинициализированы значениями параметров `n` и `d` данного класса[3]. Вдобавок были внесены изменения в реализацию методов `toString` и `add`, позволяющие им использовать поля, а не параметры класса. Эта версия класса `Rational` проходит компиляцию. Ее можно протестировать путем сложения рациональных чисел:

```
scala> val oneHalf = new Rational(1, 2)
oneHalf: Rational = 1/2

scala> val twoThirds = new Rational(2, 3)
twoThirds: Rational = 2/3

scala> oneHalf add twoThirds
res2: Rational = 7/6
```

Теперь вы уже можете сделать то, чего не могли сделать раньше, а именно получить доступ к значениям числителя и знаменателя из-за пределов объекта. Для этого нужно просто обратиться к публичным полям `numer` и `denom`:

```
scala> val r = new Rational(1, 2)
r: Rational = 1/2
```

---

[1] Фактически объект Rational можно сложить с самим собой, тогда ссылка будет на тот же объект, в отношении которого был вызван метод add. Но поскольку данному методу можно передать любой объект Rational, то компилятор все же не позволит вам воспользоваться кодом that.n.

[2] Параметрические поля, содержащие сокращения для написания аналогичного кода, будут рассматриваться в разделе 10.6.

[3] Несмотря на то, что n и d используются в теле класса, и учитывая, что они применяются только внутри конструкторов, компилятор Scala не станет выделять под них поля. Таким образом, получив этот код, компилятор Scala создаст класс с двумя полями типа Int: одним для numer, другим для denom.

```
scala> r.numer
res3: Int = 1

scala> r.denom
res4: Int = 2
```

6.6. Собственные ссылки

Ключевое слово `this` позволяет сослаться на экземпляр объекта, в отношении которого был вызван выполняемый в данный момент метод, или, если оно использовалось в конструкторе, — на создаваемый экземпляр объекта. Рассмотрим в качестве примера добавление метода `lessThan`. Он проверяет, не имеет ли объект `Rational`, в отношении которого он вызван, значение, меньшее чем значение параметра:

```
def lessThan(that: Rational) =
  this.numer * that.denom < that.numer * this.denom
```

Здесь выражение `this.numer` ссылается на числительное объекта, в отношении которого вызван метод `lessThan`. Можно также не указывать префикс `this` и написать просто `numer`, обе записи будут равнозначны.

В качестве примера случаев, когда вам не обойтись без `this`, рассмотрим добавление к классу `Rational` метода `max`, возвращающего наибольшее число из заданного рационального числа и переданного аргумента:

```
def max(that: Rational) =
  if (this.lessThan(that)) that else this
```

Здесь первое ключевое слово `this` избыточно. Можно его не указывать и написать `lessThan(that)`. Но второе ключевое слово `this` представляет результат метода в том случае, если тест вернет `false`, и если вы его не укажете, то возвращать будет просто нечего!

6.7. Вспомогательные конструкторы

Иногда нужно, чтобы в классе было несколько конструкторов. В Scala все конструкторы, кроме первичного, называются *вспомогательными*. Например, рациональное число со знаменателем 1 можно кратко записать просто в виде числителя. Вместо 5/1, например, можно просто указать 5. Поэтому было бы неплохо, чтобы вместо записи `new Rational(5, 1)` программисты могли написать просто `new Rational(5)`. Для этого потребуется добавить к классу `Rational` вспомогательный конструктор, который получает только один аргумент — числитель, а в качестве знаменателя имеет предопределенное значение 1. Как может выглядеть соответствующий код, показано в листинге 6.2.

Листинг 6.2. Класс Rational со вспомогательным конструктором

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  val numer: Int = n
  val denom: Int = d

  def this(n: Int) = this(n, 1) // Вспомогательный конструктор

  override def toString = s"$numer/$denom"

  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
```

Определения вспомогательных конструкторов в Scala начинаются с `def this(...)`. Тело вспомогательного конструктора класса `Rational` просто вызывает первичный конструктор, передавая дальше свой единственный аргумент `n` в качестве числителя и `1` — в качестве знаменателя. Увидеть вспомогательный конструктор в действии можно, набрав в интерпретаторе следующий код:

```
scala> val y = new Rational(3)
y: Rational = 3/1
```

В Scala каждый вспомогательный конструктор в качестве первого действия должен вызывать еще один конструктор того же класса. Иными словами, первой инструкции в каждом вспомогательном конструкторе каждого класса Scala следует иметь вид `this(...)`. Вызываемым должен быть либо первичный конструктор (как в примере с классом `Rational`), либо другой вспомогательный конструктор, который появляется в тексте программы перед вызывающим его конструктором. Конечный результат применения данного правила заключается в том, что каждый вызов конструктора в Scala должен в конце концов завершаться вызовом первичного конструктора класса. Первичный конструктор, таким образом, — единственная точка входа в класс.

ПРИМЕЧАНИЕ

Знатоков Java может удивить то, что в Scala правила в отношении конструкторов более строгие, чем в Java. Ведь в Java первым действием конструктора должен быть либо вызов другого конструктора того же класса, либо вызов конструктора суперкласса напрямую. В классе Scala конструктор суперкласса может быть вызван только первичным конструктором. Более сильные ограничения в Scala фактически являются компромиссом дизайна — платой за большую лаконичность и простоту конструкторов Scala по сравнению с конструкторами Java. Суперклассы и подробности вызова конструкторов и наследования будут рассмотрены в главе 10.

6.8. Приватные поля и методы

В предыдущей версии класса `Rational` мы просто инициализировали `numer` значением `n`, а `denom` — значением `d`. Из-за этого числитель и знаменатель `Rational` могут превышать необходимые значения. Например, дробь 66/42 можно сократить и привести к виду 11/7, но первичный конструктор класса `Rational` пока этого не делает:

```
scala> new Rational(66, 42)
res5: Rational = 66/42
```

Чтобы выполнить такое сокращение, нужно разделить числитель и знаменатель на их *наибольший общий делитель*. Например, таковым для 66 и 42 будет число 6. (Иными словами, 6 — наибольшее целое число, на которое без остатка делится как 66, так и 42.) Деление и числителя, и знаменателя числа 66/42 на 6 приводит к получению сокращенной формы 11/7. Один из способов решения данной задачи показан в листинге 6.3.

Листинг 6.3. Класс Rational с приватным полем и методом

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g

  def this(n: Int) = this(n, 1)

  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  override def toString = s"$numer/$denom"

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

В данной версии класса `Rational` было добавлено приватное поле `g` и изменены инициализаторы для полей `numer` и `denom`. (*Инициализатором* называется код, инициализирующий переменную, например, `n / g`, который инициализирует поле `numer`.) Поле `g` является приватным, поэтому доступ к нему может быть выполнен изнутри, но не снаружи тела класса. Кроме того, был добавлен приватный метод по имени `gcd`, вычисляющий наибольший общий делитель двух переданных ему значений `Int`. Например, вызов `gcd(12, 8)` дает результат 4. Как было показано в разделе 4.1, чтобы сделать поле или метод приватным, следует просто поставить перед его определением ключевое слово `private`. Назначение приватного «вспомо-

гательного метода» gcd — обособление кода, необходимого для остальных частей класса, в данном случае для первичного конструктора. Чтобы обеспечить постоянное положительное значение поля g, методу передаются абсолютные значения параметров n и d, которые вызов получает в отношении этих параметров метода abs. Последний может вызываться в отношении любого Int-объекта в целях получения его абсолютного значения.

Компилятор Scala поместит коды инициализаторов трех полей класса Rational в первичный конструктор в порядке их следования в исходном коде. Таким образом, инициализатор поля g, имеющий код gcd(n.abs, d.abs), будет выполнен до выполнения двух других инициализаторов, поскольку в исходном коде появляется первым. Поле g будет инициализировано результатом — наибольшим общим делителем абсолютных значений параметров n и d класса. Затем поле g будет использовано в инициализаторах полей numer и denom. Разделив n и d на их наибольший общий делитель g, каждый объект Rational можно сконструировать в нормализованной форме:

```
scala> new Rational(66, 42)
res6: Rational = 11/7
```

6.9. Определение операторов

Текущая реализация класса Rational нас вполне устраивает, но ее можно сделать гораздо более удобной в использовании. Вы можете спросить, почему допустима запись:

```
x + y
```

если x и y — целые числа или числа с плавающей точкой. Но когда это рациональные числа, приходится пользоваться записью:

```
x.add(y)
```

или в крайнем случае:

```
x add y
```

Такое положение дел ничем не оправдано. Рациональные числа совершенно неотличимы от остальных чисел. В математическом смысле они гораздо более естественны, скажем, числа с плавающей точкой. Так почему бы не воспользоваться во время работы с ними естественными математическими операторами? И в Scala есть такая возможность. Она будет показана в оставшейся части главы.

Сначала нужно заменить add обычным математическим символом. Сделать это нетрудно, поскольку знак + является в Scala вполне допустимым идентификатором. Можно просто определить метод с именем +. Если уж на то пошло, то можно определить и метод *, выполняющий умножение. Результат показан в листинге 6.4.

Листинг 6.4. Класс Rational с методами-операторами

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g

  def this(n: Int) = this(n, 1)

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def * (that: Rational): Rational =
    new Rational(numer * that.numer, denom * that.denom)

  override def toString = s"$numer/$denom"

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

После такого определения класса `Rational` можно будет воспользоваться следующим кодом:

```
scala> val x = new Rational(1, 2)
x: Rational = 1/2

scala> val y = new Rational(2, 3)
y: Rational = 2/3

scala> x + y
res7: Rational = 7/6
```

Как всегда, синтаксис оператора в последней строке ввода — эквивалент вызова метода. Можно также использовать следующий код:

```
scala> x.+(y)
res8: Rational = 7/6
```

но читать его будет намного труднее.

Следует также заметить, что из-за действующих в Scala правил приоритета операторов, рассмотренных в разделе 5.9, метод `*` будет привязан к объектам `Rational` сильнее метода `+`. Иными словами, выражения, в которых к объектам `Rational` применяются операции `+` и `*`, будут вести себя вполне ожидаемым образом. Например, `x + x * y` будет выполняться как `x + (x * y)`, а не как `(x + x) * y`:

```
scala> x + x * y
res9: Rational = 5/6

scala> (x + x) * y
res10: Rational = 2/3

scala> x + (x * y)
res11: Rational = 5/6
```

6.10. Идентификаторы в Scala

Вам уже встречались два наиболее важных способа составления идентификаторов в Scala — из буквенно-цифровых символов и из операторов. В Scala используются весьма гибкие правила формирования идентификаторов. Кроме двух уже встречавшихся форм, существуют еще две. Все четыре формы составления идентификаторов рассматриваются в этом разделе.

Буквенно-цифровые идентификаторы начинаются с буквы или знака подчеркивания, за которыми могут следовать другие буквы, цифры или знаки подчеркивания. Символ `$` также считается буквой, но зарезервирован для идентификаторов, создаваемых компилятором Scala. Идентификаторы в пользовательских программах не должны содержать символы `$`, несмотря на возможность успешно пройти компиляцию: если это произойдет, то могут возникнуть конфликты имен с теми идентификаторами, которые будут созданы компилятором Scala.

В Scala соблюдается соглашение, принятое в Java относительно применения идентификаторов в смешанном регистре[1], таких как `toString` и `HashSet`. Хотя использование знаков подчеркивания в идентификаторах вполне допустимо, в программах на Scala они встречаются довольно редко — отчасти в целях соблюдения совместимости с Java, а также из-за того, что знаки подчеркивания в коде Scala активно применяются не только для идентификаторов. Поэтому лучше избегать таких идентификаторов, как, например, `to_string`, `__init__` или `name_`. Имена полей, параметры методов, имена локальных переменных и имена функций в смешанном регистре должны начинаться с буквы в нижнем регистре, например: `length`, `flatMap` и `s`. Имена классов и трейтов в смешанном регистре должны начинаться с буквы в верхнем регистре, например: `BigInt`, `List` и `UnbalancedTreeMap`[2].

---

[1] Этот стиль именования идентификаторов называется верблюжьим, поскольку у идентификаторов ИмеютсяГорбы, состоящие из символов в верхнем регистре.

[2] В разделе 16.5 вы увидите, что иногда может возникнуть желание придать классу особый вид, как у case-класса, чье имя состоит только из символов оператора. Например, в API Scala имеется класс по имени ::, облегчающий сопоставление с образцом для объектов List.

ПРИМЕЧАНИЕ

Одним из последствий использования в идентификаторе замыкающего знака подчеркивания при попытке, к примеру, написания объявления val name_: Int = 1, может стать ошибка компиляции. Компилятор подумает, что вы пытаетесь объявить val-переменную по имени name_:. Чтобы такой идентификатор прошел компиляцию, перед двоеточием нужно поставить дополнительный пробел, как в коде val name_ : Int = 1.

Один из примеров отступления Scala от соглашений, принятых в Java, касается имен констант. В Scala слово *«константа»* означает не только `val`-переменную. Даже притом, что `val`-переменная остается неизменной после инициализации, она не перестает быть переменной. Например, параметры метода относятся к `val`-переменным, но при каждом вызове метода в этих `val`-переменных содержатся разные значения. Константа обладает более выраженным постоянством. Например, `scala.math.Pi` определяется как значение с двойной точностью, наиболее близкое к реальному значению числа π — отношению длины окружности к ее диаметру. Это значение вряд ли когда-либо изменится, поэтому со всей очевидностью можно сказать, что `Pi` — константа. Константы можно использовать также для присваивания имен значениям, которые иначе были бы в вашем коде *магическими числами* — буквальными значениями без объяснений, которые в худшем случае появлялись бы в коде в нескольких местах. Вдобавок может понадобиться определить константы для использования при сопоставлении с образцом (подобный случай будет рассматриваться в разделе 15.2). В соответствии с соглашением, принятым в Java, константам присваиваются имена, в которых используются символы в верхнем регистре, где знак подчеркивания является разделителем слов, например `MAX_VALUE` или `PI`. В Scala соглашение требует, чтобы в верхнем регистре была только первая буква. Таким образом, константы, названные в стиле Java, например `X_OFFSET`, будут работать в Scala в качестве констант, но согласно соглашению, принятому в Scala, для имен констант применяется смешанный регистр, например `XOffset`.

Идентификатор оператора состоит из одного или нескольких символов операторов. Таковыми являются выводимые на печать ASCII-символы, такие как `+`, `:`, `?`, `~` или `#`[1]. Ниже показаны некоторые примеры идентификаторов операторов:

```
+ ++ ::: <?> :->
```

Компилятор Scala на внутреннем уровне перерабатывает идентификаторы операторов, чтобы превратить их в допустимые Java-идентификаторы со встроенными символами `$`. Например, идентификатор `:->` будет представлен как

[1] Точнее, символ оператора принадлежит к математическим символам (Sm) или прочим символам (So) набора Unicode либо к семибитным ASCII-символам, не являющимся буквами, цифрами, круглыми скобками, квадратными скобками, фигурными скобками, одинарными или двойными кавычками или знаками подчеркивания, точки, точки с запятой, запятой или обратных кавычек.

`$colon$minus$greater`. Если вам когда-либо захочется получить доступ к этому идентификатору из кода Java, то потребуется использовать данное внутреннее представление.

Поскольку идентификаторы операторов в Scala могут принимать произвольную длину, то между Java и Scala есть небольшая разница в этом вопросе. В Java введенный код `x<-y` будет разобран на четыре лексических символа, в результате чего станет эквивалентен `x < - y`. В Scala оператор `<-` будет рассмотрен как единый идентификатор, в результате чего получится `x <- y`. Если нужно получить первую интерпретацию, то следует отделить символы `<` и `-` друг от друга пробелом. На практике это вряд ли станет проблемой, так как немногие станут писать на Java `x<-y`, не вставляя пробелы или круглые скобки между операторами.

Смешанный идентификатор состоит из буквенно-цифрового идентификатора, за которым стоят знак подчеркивания и идентификатор оператора. Например, `unary_+`, использованный как имя метода, определяет унарный оператор `+`. А `myvar_=`, использованный как имя метода, определяет оператор присваивания. Кроме того, смешанный идентификатор вида `myvar_=` генерируется компилятором Scala в целях поддержки *свойств* (более подробно этот вопрос рассматривается в главе 18).

Литеральный идентификатор представляет собой произвольную строку, заключенную в обратные кавычки (`` `...` ``). Примеры литеральных идентификаторов выглядят следующим образом:

```
`x` `<clinit>` `yield`
```

Замысел состоит в том, что между обратными кавычками можно поместить любую строку, которую среда выполнения станет воспринимать в качестве идентификатора. В результате всегда будет получаться идентификатор Scala. Это сработает даже в том случае, если имя, заключенное в обратные кавычки, является в Scala зарезервированным словом. Обычно такие идентификаторы используются при обращении к статическому методу `yield` в Java-классе `Thread`. Вы не можете прибегнуть к коду `Thread.yield()`, поскольку в Scala `yield` является зарезервированным словом. Но имя метода все же можно применить, если заключить его в обратные кавычки, например `Thread.` `` `yield` `` `()`.

6.11. Перегрузка методов

Вернемся к классу `Rational`. После внесения последних изменений появилась возможность применять операции сложения и умножения рациональных чисел в их естественном виде. Но мы все же упустили из виду смешанную арифметику. Например, вы не можете умножить рациональное число на целое, поскольку операнды у оператора `*` всегда должны быть объектами `Rational`. Следовательно, для рационального числа `r` вы не можете написать код `r * 2`. Вам нужно написать `r * new Rational(2)`, а это имеет неприглядный вид.

Чтобы сделать класс `Rational` еще более удобным в использовании, добавим к нему новые методы, выполняющие смешанное сложение и умножение рациональных и целых чисел. А заодно добавим методы вычитания и деления. Результат показан в листинге 6.5.

Листинг 6.5. Класс Rational с перегруженными методами

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g

  def this(n: Int) = this(n, 1)

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)

  def - (that: Rational): Rational =
    new Rational(
      numer * that.denom - that.numer * denom,
      denom * that.denom
    )

  def - (i: Int): Rational =
    new Rational(numer - i * denom, denom)

  def * (that: Rational): Rational =
    new Rational(numer * that.numer, denom * that.denom)

  def * (i: Int): Rational =
    new Rational(numer * i, denom)

  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom * that.numer)

  def / (i: Int): Rational =
    new Rational(numer, denom * i)

  override def toString = s"$numer/$denom"

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

Теперь здесь две версии каждого арифметического метода: одна в качестве аргумента получает рациональное число, вторая — целое. Иными словами, все эти методы называются *перегруженными*, поскольку каждое имя теперь используется несколькими методами. Например, имя `+` применяется и методом, получающим объект `Rational`, и методом, получающим объект `Int`. При вызове метода компилятор выбирает версию перегруженного метода, которая в точности соответствует типу аргументов. Например, если аргумент `y` в вызове `x.+(y)` является объектом `Rational`, то компилятор выберет метод `+`, получающий в качестве параметра объект `Rational`. Но если аргумент — целое число, то компилятор выберет метод `+`, получающий в качестве параметра объект `Int`. Если испытать код в действии:

```
scala> val x = new Rational(2, 3)
x: Rational = 2/3

scala> x * x
res12: Rational = 4/9

scala> x * 2
res13: Rational = 4/3
```

станет понятно, что вызываемый метод `*` определяется всякий раз по типу его правого операнда.

ПРИМЕЧАНИЕ

В Scala процесс анализа при выборе перегруженного метода очень похож на аналогичный процесс в Java. В любом случае выбирается перегруженная версия, которая лучше подходит к статическим типам аргументов. Иногда случается, что одной такой версии нет, и тогда компилятор выдаст ошибку, связанную с неоднозначной ссылкой, — ambiguous reference.

6.12. Неявные преобразования

Теперь, когда можно воспользоваться кодом `r * 2`, вы также можете захотеть поменять операнды местами, задействовав код `2 * r`. К сожалению, пока этот код работать не будет:

```
scala> 2 * r
         ^
       error: overloaded method value * with alternatives:
         (x: Double)Double <and>
         (x: Float)Float <and>
         (x: Long)Long <and>
         (x: Int)Int <and>
         (x: Char)Int <and>
         (x: Short)Int <and>
         (x: Byte)Int
        cannot be applied to (Rational)
```

Проблема в том, что эквивалент выражения `2 * r` — выражение `2.*(r)`, то есть вызов метода в отношении числа 2, которое является целым. Но в классе `Int` не содержится метода умножения, получающего в качестве аргумента объект `Rational`, его там и не может быть, поскольку он не входит в состав стандартных классов библиотеки Scala.

Но в Scala есть другой способ решения этой проблемы: можно создать неявное преобразование, которое при необходимости автоматически превратит целые числа в рациональные. Попробуйте добавить в интерпретатор следующую строку кода:

```
scala> implicit def intToRational(x: Int) = new Rational(x)
```

Она определяет метод преобразования из типа `Int` в тип `Rational`. Модификатор `implicit` перед определением метода сообщает компилятору о том, что в ряде ситуаций данный метод следует применять автоматически. Определив преобразование, можно еще раз попытаться выполнить код, который перед этим дал сбой:

```
scala> val r = new Rational(2,3)
r: Rational = 2/3

scala> 2 * r
res15: Rational = 4/3
```

Чтобы неявное преобразование заработало, оно должно находиться в области видимости. Если поместить определение метода с модификатором `implicit` внутрь класса `Rational`, то он не попадет в область видимости интерпретатора. Пока вам следует определить его непосредственно в интерпретаторе.

Из данного примера можно сделать вывод, что неявное преобразование — весьма эффективная технология для придания библиотекам гибкости и повышения удобства их использования. Такие преобразования весьма серьезно воздействуют на код программы, поэтому в способах их применения можно легко наделать ошибок. Дополнительные сведения о неявных преобразованиях, а также способах их помещения в случае необходимости в область видимости будут представлены в главе 21.

6.13. Предостережение

Создание методов с именами операторов и определение неявного преобразования, продемонстрированные в этой главе, призваны помочь в проектировании библиотек, для которых код клиента будет лаконичным и понятным. Scala предоставляет широкие возможности для разработки таких весьма доступных для использования библиотек. Но, пожалуйста, имейте в виду: реализуя возможности, не стоит забывать об ответственности.

При неумелом использовании и методы-операторы, и неявные преобразования могут сделать клиентский код таким, что его станет трудно читать и понимать. Выполнение компилятором неявного преобразования никак внешне не проявляет-

ся и не записывается в явном виде в исходный код. Поэтому программистам-клиентам может быть невдомек, что именно оно и применяется в вашем коде. И хотя методы-операторы обычно делают клиентский код более лаконичным и читабельным, таким он становится только для наиболее сведущих программистов-клиентов, способных запомнить и распознать значение каждого оператора.

При проектировании библиотек всегда нужно стремиться сделать клиентский код не просто лаконичным, но и легко читаемым и понятным. Читабельность в значительной степени может быть обусловлена лаконичностью, которая способна заходить очень далеко. Проектируя библиотеки, позволяющие добиваться изысканной лаконичности, и в то же время создавая понятный клиентский код, вы можете существенно повысить продуктивность работы программистов, которые используют эти библиотеки.

Резюме

В этой главе мы рассмотрели многие аспекты классов Scala. Вы увидели способы добавления к классу параметров, определили несколько конструкторов, операторы и методы и настроили классы таким образом, чтобы их применение приобрело более естественный вид. Важнее всего, вероятно, было показать вам, что определение и использование неизменяющихся объектов — вполне естественный способ программирования на Scala.

Хотя показанная здесь финальная версия класса `Rational` соответствует всем требованиям, обозначенным в начале главы, ее можно усовершенствовать. Позже мы вернемся к этому примеру. В частности, в главе 30 будет рассмотрено переопределение методов `equals` и `hashcode`, которое позволяет объектам `Rational` улучшить свое поведение в момент, когда их сравнивают с помощью оператора `==` или помещают в хеш-таблицы. В главе 21 поговорим о том, как помещать неявные определения методов в объекты-компаньоны класса `Rational`, которые упрощают для программистов-клиентов помещение в область видимости объектов типа `Rational`.

7 Встроенные управляющие конструкции

В Scala имеется весьма незначительное количество встроенных управляющих конструкций. К ним относятся `if`, `while`, `for`, `try`, `match` и вызовы функций. Их в Scala немного, поскольку с момента создания данного языка в него были включены функциональные литералы. Вместо накопления в базовом синтаксисе одной за другой высокоуровневых управляющих конструкций Scala собирает их в библиотеках. Как именно это делается, мы покажем в главе 9. А здесь рассмотрим имеющиеся в Scala немногочисленные встроенные управляющие конструкции.

Следует учесть, что почти все управляющие конструкции Scala приводят к какому-либо значению. Такой подход принят в функциональных языках, где программы рассматриваются в качестве вычислителей значений, стало быть, компоненты программы тоже должны вычислять значения. Можно рассматривать данное обстоятельство как логическое завершение тенденции, уже присутствующей в императивных языках. В них вызовы функций могут возвращать значение, даже когда наряду с этим будет происходить обновление вызываемой функцией выходной переменной, переданной в качестве аргумента. Кроме того, в императивных языках зачастую имеется тернарный оператор (такой как оператор `?:` в C, C++ и Java), который ведет себя полностью аналогично `if`, но при этом возвращает значение. Scala позаимствовал эту модель тернарного оператора, но назвал ее `if`. Иными словами, используемый в Scala оператор `if` может выдавать значение. Затем эта тенденция в Scala получила развитие: `for`, `try` и `match` тоже стали выдавать значения.

Программисты могут использовать полученное в результате значение, чтобы упростить свой код, применяя те же приемы, что и для значений, возвращаемых функциями. Не будь этой особенности, программистам пришлось бы создавать временные переменные, просто чтобы хранить результаты, вычисленные внутри управляющей конструкции. Отказ от таких переменных немного упрощает код, а также избавляет от многих ошибок, возникающих, когда в одном ответвлении переменная создается, а в другом о ее создании забывают.

В целом, основные управляющие конструкции Scala в минимальном составе обеспечивают все, что нужно было взять из императивных языков. При этом они

позволяют сделать код более лаконичным за счет неизменного наличия значений, получаемых в результате их применения. Чтобы показать все это в работе, далее более подробно рассмотрим основные управляющие конструкции Scala.

7.1. Выражения if

Выражение `if` в Scala работает практически так же, как во многих других языках. Оно проверяет условие, а затем выполняет одну из двух ветвей кода в зависимости от того, вычисляется ли условие в `true`. Простой пример, написанный в императивном стиле, выглядит следующим образом:

```
var filename = "default.txt"
if (!args.isEmpty)
  filename = args(0)
```

В этом коде объявляется переменная по имени `filename`, которая инициализируется значением по умолчанию. Затем в нем используется выражение `if` с целью проверить, предоставлены ли программе какие-либо аргументы. Если да, то в переменную вносят изменения, чтобы в ней содержалось значение, указанное в списке аргументов. Если нет, то выражение оставляет значение переменной, установленное по умолчанию.

Этот код можно сделать гораздо более выразительным, поскольку, как упоминалось в шаге 3 главы 2, выражение `if` в Scala возвращает значение. В листинге 7.1 показано, как можно выполнить те же самые действия, что и в предыдущем примере, не прибегая к использованию `var`-переменных.

Листинг 7.1. Особый стиль Scala, применяемый для условной инициализации

```
val filename =
  if (!args.isEmpty) args(0)
  else "default.txt"
```

На этот раз у `if` имеются два ответвления. Если массив `args` непустой, то выбирается его начальный элемент `args(0)`. В противном случае выбирается значение по умолчанию. Выражение `if` выдает результат в виде выбранного значения, которым инициализируется переменная `filename`. Данный код немного короче предыдущего. Но гораздо более существенно то, что в нем используется `val`-, а не `var`-переменная. Это соответствует функциональному стилю и помогает вам примерно так же, как применение финальной (`final`) переменной в Java. Она сообщает читателям кода, что переменная никогда не изменится, избавляя их от необходимости просматривать весь код в области видимости переменной, чтобы понять, изменяется ли она где-нибудь.

Второе преимущество использования `var`-переменной вместо `val`-переменной заключается в том, что она лучше поддерживает выводы, которые делаются с помощью *эквациональных рассуждений (equational reasoning)*. Введенная переменная *равна* вычисляющему выражению при условии, что у него нет побочных эффектов.

Таким образом, всякий раз, собираясь написать имя переменной, вы можете вместо него написать выражение. Вместо `println(filename)`, к примеру, можно просто написать следующий код:

```
println(if (!args.isEmpty) args(0) else "default.txt")
```

Выбор за вами. Вы можете прибегнуть к любому из вариантов. Использование `val`-переменных помогает совершенно безопасно проводить подобный рефакторинг кода по мере его развития.

> Всегда ищите возможности для применения `val`-переменных. Они смогут упростить не только чтение вашего кода, но и его рефакторинг.

7.2. Циклы while

Используемые в Scala циклы `while` ведут себя точно так же, как и в других языках. В них имеются условие и тело, которое выполняется снова и снова, пока условие вычисляется в `true`. Пример показан в листинге 7.2.

Листинг 7.2. Вычисление наибольшего общего делителя с применением цикла while

```
def gcdLoop(x: Long, y: Long): Long = {
  var a = x
  var b = y
  while (a != 0) {
    val temp = a
    a = b % a
    b = temp
  }
  b
}
```

В Scala имеется также цикл `do-while`. Он работает точно так же, как и цикл `while`, но условие в нем проверяется после тела цикла, а не до него. В листинге 7.3 показан скрипт на Scala, использующий цикл `do-while` для вывода на экран строк, считанных со стандартного устройства ввода, до тех пор, пока не будет введена пустая строка.

Листинг 7.3. Чтение со стандартного устройства ввода с применением цикла do-while

```
var line = ""
do {
  line = readLine()
  println("Read: " + line)
} while (line != "")
```

Конструкции `while` и `do-while` называются циклами, а не выражениями, поскольку в результате выполнения они не возвращают никакого содержательного значения. Типом результата является `Unit`. Получается так, что фактически суще-

ствует только одно значение, имеющее тип `Unit`. Оно называется *`unit`-значением* и записывается как `()`. Существование `()` отличает имеющийся в Scala класс `Unit` от используемого в Java типа `void`. Попробуйте выполнить в интерпретаторе следующий код:

```
scala> def greet() = { println("hi") }
greet: ()Unit

scala> val whatAmI = greet()
hi
whatAmI: Unit = ()
```

Поскольку тип выражения `println("hi")` определяется как `Unit`, то `greet` определяется как процедура с результирующим типом `Unit`. Поэтому `greet` возвращает `Unit` значение `()`. Это подтверждается в следующей строке, где переменная `whatAmI` имеет тип `Unit` и возвращает значение `()`.

Еще одна конструкция с типом результата в виде `Unit`-значения, о которой здесь стоит упомянуть, — присваивание нового значения `var`-переменным. Например, если попробовать считать строки в Scala, воспользовавшись следующей идиомой цикла `while` из Java (а также из C и C++), то возникнет проблема:

```
var line = ""
while ((line = readLine()) != "") // Этот код не будет работать!
  println("Read: " + line)
```

При компиляции этого кода Scala выдаст предупреждение о том, что сравнение значений типа `Unit` и `String` с использованием `!=` всегда дает результат `true`. В то время как в Java присваивание выдает в качестве результата присваиваемое значение (в данном случае строку из стандартного ввода), в Scala оно всегда выдает `Unit`-значение `()`. Таким образом, значением присваивания `line = readLine()` всегда будет `()`, но никак не `""`. В результате условие данного цикла `while` никогда не будет вычислено в `false`, и из-за этого цикл никогда не завершится.

В результате цикла `while` никакое значение не возвращается, поэтому зачастую его не включают в чисто функциональные языки. В таких языках имеются выражения, а не циклы. И тем не менее цикл `while` включен в Scala, поскольку иногда императивные решения бывает легче читать, особенно тем программистам, которые много работали с императивными языками. Например, если нужно закодировать алгоритм, повторяющий процесс до тех пор, пока не изменятся какие-либо условия, то цикл `while` позволяет выразить это напрямую, в то время как функциональную альтернативу наподобие использования рекурсии читателям кода распознавать оказывается сложнее.

Например, в листинге 7.4 показан альтернативный способ определения наибольшего общего знаменателя двух чисел[1]. При условии присваивания `x` и `y` в функции

[1] Здесь в функции gcd используется точно такой же подход, который применялся в одноименной функции в листинге 6.3 в целях вычисления наибольших общих делителей для класса Rational. Основное отличие заключается в том, что вместо Int-значений gcd код работает с Long-значениями.

gcd таких же значений, как и в функции gcdLoop, показанной в листинге 7.2, будет выдан точно такой же результат. Разница между этими двумя подходами состоит в том, что функция gcdLoop написана в императивном стиле с использованием var-переменных и цикла while, а функция gcd — в более функциональном стиле с применением рекурсии (gcd вызывает саму себя), для чего не нужны var-переменные.

Листинг 7.4. Вычисление наибольшего общего делителя с применением рекурсии

```
def gcd(x: Long, y: Long): Long =
  if (y == 0) x else gcd(y, x % y)
```

В целом мы рекомендуем относиться к циклам while в своем коде с оглядкой, как и к использованию в нем var-переменных. Фактически циклы while и var-переменные зачастую идут рука об руку. Поскольку циклы не дают результата в виде значения, то для внесения в программу каких-либо изменений цикл while обычно будет нуждаться либо в обновлении var-переменных, либо в выполнении ввода-вывода. В этом можно убедиться, посмотрев на работу показанного ранее примера gcdLoop. По мере выполнения своей задачи цикл while обновляет значения var-переменных a и b. Поэтому мы советуем проявлять особую осмотрительность при использовании в коде циклов while. Если нет достаточных оснований для применения цикла while или do-while, то попробуйте найти способ сделать то же самое без их участия.

7.3. Выражения for

Используемые в Scala выражения for являются для итераций чем-то наподобие швейцарского армейского ножа. Чтобы можно было реализовать широкий спектр итераций, эти выражения позволяют различными способами составлять комбинации из довольно простых ингредиентов. Простое применение дает возможность решать простые задачи вроде поэлементного обхода последовательности целых чисел. Более сложные выражения могут выполнять обход элементов нескольких коллекций различных видов, фильтровать элементы на основе произвольных условий и создавать новые коллекции.

Обход элементов коллекций

Самое простое, что можно сделать с выражением for, — это выполнить обход всех элементов коллекции. Например, в листинге 7.5 показан код, который выводит имена всех файлов, содержащихся в текущем каталоге. Ввод-вывод выполняется с помощью API Java. Сначала в текущем каталоге, ".", создается объект java.io.File и вызывается его метод listFiles. Последний возвращает массив объектов File, по одному на каталог или файл, содержащийся в текущем каталоге. Получившийся в результате массив сохраняется в переменной filesHere.

Листинг 7.5. Получение списка файлов в каталоге с применением выражения for

```
val filesHere = (new java.io.File(".")).listFiles

for (file <- filesHere)
  println(file)
```

С помощью синтаксиса `file <- filesHere`, называемого *генератором*, выполняется обход элементов массива `filesHere`. При каждой итерации значением элемента инициализируется новая `val`-переменная по имени `file`. Компилятор приходит к выводу, что типом `file` является `File`, поскольку `filesHere` имеет тип `Array[File]`. Для каждой итерации выполняется тело выражения `for`, имеющее код `println(file)`. Метод `toString`, определенный в классе `File`, выдает имя файла или каталога, поэтому будут выведены имена всех файлов и каталогов текущего каталога.

Синтаксис выражения `for` работает не только с массивами, но и с коллекциями любого типа[1]. Особый и весьма удобный случай — применение типа `Range`, который был упомянут в табл. 5.4 (см. выше). Можно создавать объекты `Range`, используя синтаксис вида `1 to 5`, и выполнять их обход с помощью `for`. Простой пример имеет следующий вид:

```
scala> for (i <- 1 to 4)
        println("Iteration " + i)
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

Если не нужно включать верхнюю границу диапазона в перечисляемые значения, то вместо `to` используется `until`:

```
scala> for (i <- 1 until 4)
        println("Iteration " + i)
Iteration 1
Iteration 2
Iteration 3
```

Перебор целых чисел наподобие этого встречается в Scala довольно часто, но намного реже, чем в других языках, где данным свойством можно воспользоваться для перебора элементов массива:

```
// В Scala такой код встречается довольно редко...
for (i <- 0 to filesHere.length - 1)
  println(filesHere(i))
```

[1] Точнее, выражение справа от символов <– в выражении for может быть любого типа, имеющего определенные методы (в данном случае foreach) с соответствующими сигнатурами. Подробности того, как компилятор Scala обрабатывает выражения for, рассматриваются в главе 23.

В это выражение `for` введена переменная `i`, которой по очереди присваивается каждое целое число от `0` до `filesHere.length - 1`, и для каждой установки `i` выполняется тело выражения. Для каждого значения `i` из массива `filesHere` извлекается и обрабатывается `i`-й элемент.

Такого вида итерации меньше распространены в Scala потому, что есть возможность выполнить непосредственный обход элементов коллекции. При этом код становится короче и исключаются многие ошибки смещения на единицу, которые могут возникнуть при обходе элементов массива. С чего нужно начинать, с 0 или 1? Что нужно прибавлять к завершающему индексу, –1, +1 или вообще ничего? На подобные вопросы ответить несложно, но также просто дать и неверный ответ. Безопаснее их вообще исключить.

Фильтрация

Иногда перебирать коллекцию целиком не нужно, а требуется отфильтровать ее в некое подмножество. В выражении `for` это можно сделать путем добавления *фильтра* в виде условия `if`, указанного в выражении `for` внутри круглых скобок. Например, код, показанный в листинге 7.6, выводит список только тех файлов текущего каталога, имена которых заканчиваются на `.scala`.

Листинг 7.6. Поиск файлов с расширением .scala с помощью for с фильтром

```
val filesHere = (new java.io.File(".")).listFiles

for (file <- filesHere if file.getName.endsWith(".scala"))
  println(file)
```

Для достижения той же цели можно применить альтернативный вариант:

```
for (file <- filesHere)
  if (file.getName.endsWith(".scala"))
    println(file)
```

Этот код выдает на выходе то же самое, что и предыдущий, и выглядит, вероятно, более привычно для программистов с опытом работы на императивных языках. Но императивная форма — только вариант, поскольку данное выражение `for` выполняется в целях получения побочных эффектов, выражающихся в выводе данных, и выдает результат в виде `Unit`-значения `()`. Чуть позже в этом разделе будет показано, что `for` называется выражением, так как по итогу его выполнения получается представляющий интерес результат, то есть коллекция, чей тип определяется компонентами `<-` выражения `for`.

Если потребуется, то в выражение можно включить еще больше фильтров. В него просто нужно добавлять условия `if`. Например, чтобы обеспечить безопасность, код в листинге 7.7 выводит только файлы, исключая каталоги. Для этого добавляется фильтр, который проверяет имеющийся у файла метод `isFile`.

Листинг 7.7. Использование в выражении for нескольких фильтров

```
for (
  file <- filesHere
  if file.isFile
  if file.getName.endsWith(".scala")
) println(file)
```

Вложенные итерации

Если добавить несколько операторов `<-`, то будут получены вложенные циклы. Например, выражение `for`, показанное в листинге 7.8, имеет два таких цикла. Внешний перебирает элементы массива `filesHere`, а внутренний — элементы `fileLines(file)` для каждого файла, имя которого заканчивается на `.scala`.

Листинг 7.8. Использование в выражении for нескольких генераторов

```
def fileLines(file: java.io.File) =
  scala.io.Source.fromFile(file).getLines().toArray

def grep(pattern: String) =
  for (
    file <- filesHere
    if file.getName.endsWith(".scala");
    line <- fileLines(file)
    if line.trim.matches(pattern)
  ) println(s"file: ${line.trim}")

grep(".*gcd.*")
```

При желании вокруг генераторов и фильтров вместо круглых скобок можно поставить фигурные. Одно из преимуществ использования фигурных скобок заключается в том, что они позволяют не ставить некоторые точки с запятой, которые нужны в момент применения круглых скобок. Это возможно потому, что (см. объяснение в разделе 4.2) компилятор Scala не будет определять использование точки с запятой внутри круглых скобок.

Привязки промежуточных переменных

Обратите внимание на повторение в предыдущем коде выражения `line.trim`. Данное вычисление довольно сложное, и потому выполнить его лучше один раз. Сделать это позволяет привязка результата к новой переменной с помощью знака равенства (`=`). Связанная переменная вводится и используется точно так же, как и `val`-переменная, но ключевое слово `val` не ставится. Пример показан в листинге 7.9.

Листинг 7.9. Промежуточное присваивание в выражении for

```
def grep(pattern: String) =
  for {
    file <- filesHere
    if file.getName.endsWith(".scala")
    line <- fileLines(file)
    trimmed = line.trim
    if trimmed.matches(pattern)
  } println(s"file: $trimmed")

grep(".*gcd.*")
```

В листинге 7.9 переменная по имени `trimmed` вводится в ходе выполнения выражения `for`. Ее инициализирует результат вызова метода `line.trim`. Затем остальная часть кода выражения `for` использует новую переменную в двух местах: в выражении `if` и в методе `println`.

Создание новой коллекции

Хотя во всех рассмотренных до сих пор примерах вы работали со значениями, получаемыми при обходе элементов, после чего о них уже не вспоминали, у вас есть возможность создать значение, чтобы запомнить результат каждой итерации. Для этого нужно перед телом выражения `for` поставить ключевое слово `yield`. Рассмотрим, к примеру, функцию, которая определяет файлы с расширением `.scala` и сохраняет их имена в массиве:

```
def scalaFiles =
  for {
    file <- filesHere
    if file.getName.endsWith(".scala")
  } yield file
```

При каждом выполнении тела выражения `for` создается одно значение, в данном случае это просто `file`. Когда выполнение выражения `for` завершится, результат будет включать все выданные значения, содержащиеся в единой коллекции. Тип получающейся коллекции задается на основе вида коллекции, обрабатываемой операторами итерации. В данном случае результат будет иметь тип `Array[File]`, поскольку `filesHere` является массивом, а выдаваемые выражением значения относятся к типу `File`.

Кстати, будьте внимательны при выборе места для ключевого слова `yield`. Синтаксис для выражения `for-yield` имеет следующий вид:

```
for условия yield тело
```

Ключевое слово `yield` ставится перед всем телом. Даже если последнее является блоком, заключенным в фигурные скобки, `yield` нужно ставить перед первой

фигурной скобкой, а не перед последним выражением блока. Избегайте создания кода, похожего на следующий:

```
for (file <- filesHere if file.getName.endsWith(".scala")) {
  yield file // Синтаксическая ошибка!
}
```

Например, выражение `for`, показанное в листинге 7.10, сначала преобразует объект типа `Array[File]` по имени `filesHere`, в котором содержатся имена всех файлов, которые есть в текущем каталоге, в объект, содержащий только имена файлов с расширением `.scala`. Для каждого из элементов создается объект типа `Array[String]`, являющийся результатом выполнения метода `fileLines`, определение которого показано в листинге 7.8. Каждый элемент этого объекта `Array[String]` содержит одну строку из текущего обрабатываемого файла. Данный объект превращается в другой объект типа `Array[String]`, содержащий только те строки, обработанные методом `trim`, которые включают подстроку `"for"`. И наконец, для каждого из них выдается целочисленное значение длины. Результатом этого выражения `for` становится объект, имеющий тип `Array[Int]` и содержащий эти значения длины.

Листинг 7.10. Преобразование объекта типа Array[File] в объект типа Array[Int] с помощью выражения for

```
val forLineLengths =
  for {
    file <- filesHere
    if file.getName.endsWith(".scala")
    line <- fileLines(file)
    trimmed = line.trim
    if trimmed.matches(".*for.*")
  } yield trimmed.length
```

Итак, основные свойства выражения `for`, применяемого в Scala, рассмотрены, но мы прошлись по ним слишком поверхностно. Более подробно о выражениях `for` поговорим в главе 23.

7.4. Обработка исключений с помощью выражений try

Исключения в Scala ведут себя практически так же, как во многих других языках. Вместо того чтобы возвращать значение (как это обычно происходит), метод может прервать работу с генерацией исключения. Код, вызвавший метод, может либо перехватить и обработать данное исключение, либо прекратить собственное выполнение, при этом передав исключение тому коду, который его вызвал. Исключение распространяется подобным образом, раскручивая стек вызова, до

тех пор, пока не встретится обрабатывающий его метод или вообще не останется ни одного метода.

Генерация исключений

Генерация исключений в Scala выглядит так же, как в Java. Создается объект исключения, который затем бросается с помощью ключевого слова `throw`:

```
throw new IllegalArgumentException
```

Как бы парадоксально это ни звучало, в Scala `throw` — это выражение, у которого есть результирующий тип. Рассмотрим пример, где этот тип играет важную роль:

```
val half =
  if (n % 2 == 0)
    n / 2
  else
    throw new RuntimeException("n должно быть четным числом ")
```

Здесь получается, что если `n` — четное число, то переменная `half` будет инициализирована половинным значением `n`. Если нечетное, то исключение сгенерируется еще до того, как переменная `half` вообще инициализируется каким-либо значением. Поэтому генерируемое исключение можно без малейших опасений рассматривать как абсолютно любое значение. Любой контекст, который пытается воспользоваться значением, возвращаемым из `throw`, никогда не сможет этого сделать, и потому никакого вреда ожидать не приходится.

С технической точки зрения сгенерированное исключение имеет тип `Nothing`. Генерацией исключения можно воспользоваться, даже если оно никогда и ни во что не будет вычислено. Подобные технические нюансы могут показаться несколько странными, но в случаях, подобных предыдущему примеру, зачастую могут пригодиться. Одно ответвление от `if` вычисляет значение, а другое выдает исключение и вычисляется в `Nothing`. Тогда типом всего выражения `if` является тип, вычисленный в той ветви, в которой проводилось вычисление. Тип `Nothing` дополнительно будет рассмотрен в разделе 11.3.

Перехват исключений

Перехват исключений выполняется с применением синтаксиса, показанного в листинге 7.11. Для выражений `catch` был выбран синтаксис с прицелом на совместимость с весьма важной частью Scala — *сопоставлением с образцом*. Этот механизм — весьма эффективное средство, которое вкратце рассматривается в данной главе, а более подробно — в главе 15.

Листинг 7.11. Применение в Scala конструкции try-catch

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException
```

```
try {
  val f = new FileReader("input.txt")
  // использование и закрытие файла
} catch {
  case ex: FileNotFoundException => // обработка ошибки отсутствия файла
  case ex: IOException => // обработка других ошибок ввода-вывода
}
```

Поведение данного выражения `try-catch` ничем не отличается от его поведения в других языках, использующих исключения. Если при выполнении тела генерируется исключение, то по очереди предпринимается попытка выполнить каждый вариант `case`. Если в данном примере исключение имеет тип `FileNotFoundException`, то будет выполнено первое условие, если тип `IOException` — то второе. Если исключение не относится ни к одному из этих типов, то выражение `try-catch` прервет свое выполнение и исключение будет распространено далее.

ПРИМЕЧАНИЕ

Одно из отличий Scala от Java, которое довольно просто заметить, заключается в том, что язык Scala не требует от вас перехватывать проверяемые исключения или их объявления в условии генерации исключений. При необходимости условие генерации исключений можно объявить с помощью аннотации @throws, но делать это не обязательно. Дополнительную информацию о @throws можно найти в разделе 31.2.

Условие finally

Если нужно, чтобы некий код выполнялся независимо от того, как именно завершилось выполнение выражения, то можно воспользоваться условием `finally`, заключив в него этот код. Например, может понадобиться гарантированное закрытие открытого файла, даже если выход из метода произошел с генерацией исключения. Пример показан в листинге 7.12[1].

Листинг 7.12. Применение в Scala условия try-finally

```
import java.io.FileReader

val file = new FileReader("input.txt")
try {
  // использование файла
} finally {
  file.close() // гарантированное закрытие файла
}
```

[1] Хотя инструкции case оператора catch всегда нужно заключать в фигурные скобки, try и finally не требуют использования фигурных скобок, если в них содержится только одно выражение. Например, можно написать: try t() catch { case e: Exception => ... } finally f().

ПРИМЕЧАНИЕ

В листинге 7.12 показан характерный для языка способ гарантированного закрытия ресурса, не имеющего отношения к оперативной памяти, например файла, сокета или подключения к базе данных. Сначала вы получаете ресурс. Затем запускается на выполнение блок try, в котором используется этот ресурс. И наконец, вы закрываете ресурс в блоке finally. Этот способ характерен для Scala точно так же, как и для Java, но в качестве альтернативного варианта достичь той же цели более лаконичным способом в Scala можно с помощью технологии под названием «шаблон временного пользования» (loan pattern). Он будет рассмотрен в разделе 9.4.

Выдача значения

Как и большинство других управляющих конструкций Scala, `try-catch-finally` выдает значение. Например, в листинге 7.13 показано, как можно попытаться разобрать URL, но при этом воспользоваться значением по умолчанию в случае плохого формирования URL. Результат получается при выполнении условия `try`, если не генерируется исключение, или же при выполнении связанного с ним условия `catch`, если исключение генерируется и перехватывается. Значение, вычисленное в условии `finally`, при наличии такового, отбрасывается. Как правило, условия `finally` выполняют какую-либо подчистку, например закрытие файла. Обычно они не должны изменять значение, вычисленное в основном теле или в `catch`-условии, связанном с `try`.

Листинг 7.13. Условие catch, выдающее значение

```
import java.net.URL
import java.net.MalformedURLException

def urlFor(path: String) =
  try {
    new URL(path)
  } catch {
    case e: MalformedURLException =>
      new URL("http://www.scala-lang.org")
}
```

Если вы знакомы с Java, то стоит отметить, что поведение Scala отличается от поведения Java только тем, что используемая в Java конструкция `try-finally` не возвращает в результате никакого значения. Как и в Java, если в условие `finally` включена в явном виде инструкция возвращения значения `return` или же в нем генерируется исключение, то это возвращаемое значение или исключение будут перевешивать все ранее выданное `try` или одним из его условий `catch`. Например, если взять вот такое несколько надуманное определение функции:

```
def f(): Int = try return 1 finally return 2
```

то при вызове `f()` будет получен результат `2`. Для сравнения, если взять определение:

```
def g(): Int = try 1 finally 2
```

то при вызове `g()` будет получен результат 1. Обе функции демонстрируют поведение, которое может удивить большинство программистов, поэтому все же лучше обойтись без значений, возвращаемых из условий `finally`. Условие `finally` более предпочтительно считать способом, который гарантирует выполнение какого-либо побочного эффекта, например закрытие открытого файла.

7.5. Выражения match

Используемое в Scala выражение сопоставления `match` позволяет выбрать из нескольких *альтернатив* (вариантов), как это делается в других языках с помощью инструкции `switch`. В общем, выражение `match` позволяет задействовать произвольные *шаблоны*, которые будут рассмотрены в главе 15. Общая форма может подождать. А пока нужно просто рассматривать использование `match` для выбора среди ряда альтернатив.

В качестве примера скрипт, показанный в листинге 7.14, считывает из списка аргументов название пищевого продукта и выводит пару к нему. Это выражение `match` анализирует значение переменной `firstArg`, которое установлено на первый аргумент, извлеченный из списка аргументов. Если это строковое значение `"salt"` («соль»), то оно выводит `"pepper"` («перец»), а если это `"chips"` («чипсы»), то `"salsa"` («острый соус») и т. д. Вариант по умолчанию указывается с помощью знака подчеркивания (`_`), который является подстановочным символом, часто используемым в Scala в качестве заместителя для неизвестного значения.

Листинг 7.14. Выражение сопоставления с побочными эффектами

```
val firstArg = if (args.length > 0) args(0) else ""

firstArg match {
  case "salt" => println("pepper")
  case "chips" => println("salsa")
  case "eggs" => println("bacon")
  case _ => println("huh?")
}
```

Есть несколько важных отличий от используемой в Java инструкции `switch`. Одно из них заключается в том, что в `case`-инструкциях Scala наряду с прочим могут применяться любые разновидности констант, а не только константы целочисленного типа, перечисления или строковые константы, как в `case`-инструкциях Java. В представленном выше листинге в качестве альтернатив используются строки. Еще одно отличие заключается в том, что в конце каждой альтернативы нет инструкции `break`. Она присутствует неявно, и нет «выпадения» (fall through) с одной альтернативы на следующую. Общий случай — без «выпадения» — становится короче, а частых ошибок удается избежать, поскольку программисты теперь не «выпадают» нечаянно.

Но, возможно, наиболее существенным отличием от `switch`-инструкции является то, что выражения сопоставления дают значение. В предыдущем примере

в каждой альтернативе в выражении сопоставления на стандартное устройство выводится значение. Как показано в листинге 7.15, данный вариант будет работать так же хорошо и выдавать значение вместо того, чтобы выводить его на устройство. Значение, получаемое из этого выражения сопоставления, сохраняется в переменной `friend`. Кроме того, что код становится короче (по крайней мере на несколько символов), теперь он выполняет две отдельные задачи: сначала выбирает продукт питания, а затем выводит его на устройство.

Листинг 7.15. Выражение сопоставления, выдающее значение

```
val firstArg = if (!args.isEmpty) args(0) else ""

val friend =
  firstArg match {
    case "salt" => "pepper"
    case "chips" => "salsa"
    case "eggs" => "bacon"
    case _ => "huh?"
  }

println(friend)
```

7.6. Программирование без break и continue

Вероятно, вы заметили, что здесь не упоминались ни `break`, ни `continue`. Из Scala эти инструкции исключены, поскольку плохо сочетаются с функциональными литералами, которые описываются в следующей главе. Назначение инструкции `continue` в цикле `while` понятно, но что она будет означать внутри функционального литерала? Так как в Scala поддерживаются оба стиля программирования — и императивный, и функциональный, — в данном случае из-за упрощения языка прослеживается небольшой перекос в сторону функционального программирования. Но волноваться не стоит. Существует множество способов писать программы, не прибегая к `break` и `continue`, и если воспользоваться функциональными литералами, то варианты с ними зачастую могут быть короче первоначального кода.

Простейший подход заключается в замене каждой инструкции `continue` условием `if`, а каждой инструкции `break` — булевой переменной. Последняя показывает, должен ли продолжаться охватывающий цикл `while`. Предположим, ведется поиск в списке аргументов строки, которая заканчивается на `.scala`, но не начинается с дефиса. В Java можно, отдавая предпочтение циклам `while`, а также инструкциям `break` и `continue`, написать следующий код:

```
int I = 0;    // Это код Java
boolean foundIt = false;
while (I < args.length) {
  if (args[i].startsWith("-")) {
    I = I + 1;
    continue;
```

```
  }
  if (args[i].endsWith(".scala")) {
    foundIt = true;
    break;
  }
  I = I + 1;
}
```

Данный фрагмент на Java можно перекодировать непосредственно в код Scala. Для этого, вместо того чтобы использовать условие `if` с последующей инструкцией `continue`, можно написать условие `if`, охватывающее всю оставшуюся часть цикла `while`. Чтобы избавиться от `break`, обычно добавляют булеву переменную, которая указывает на необходимость продолжения, но в данном случае можно задействовать уже существующую переменную `foundIt`. При использовании этих двух приемов код приобретает вид, показанный в листинге 7.16.

Листинг 7.16. Выполнение цикла без break или continue

```
var i = 0
var foundIt = false

while (i < args.length && !foundIt) {
  if (!args(i).startsWith("-")) {
    if (args(i).endsWith(".scala"))
      foundIt = true
  }
  i = i + 1
}
```

Код Scala, показанный в листинге 7.16, очень похож на первоначальный код Java. Основные части остались на месте и располагаются в том же порядке. Используются две переназначаемые переменные и цикл `while`. Внутри цикла выполняются проверки того, что `i` меньше `args.length`, а также наличия `"-"` и `".scala"`.

Если в коде листинга 7.16 нужно избавиться от `var`-переменных, то можно попробовать применить один из подходов, заключающийся в переписывании цикла в рекурсивную функцию. Можно, к примеру, определить функцию `searchFrom`, которая получает на входе целочисленное значение, выполняет поиск с указанной им позиции, а затем возвращает индекс желаемого аргумента. При использовании данного приема код приобретет вид, показанный в листинге 7.17.

Листинг 7.17. Рекурсивная альтернатива циклу с применением var-переменных

```
def searchFrom(i: Int): Int =
  if (i >= args.length) -1
  else if (args(i).startsWith("-")) searchFrom(i + 1)
  else if (args(i).endsWith(".scala")) i
  else searchFrom(i + 1)

val i = searchFrom(0)
```

В версии, показанной в данном листинге, в имени функции отображено ее назначение, изложенное в понятной человеку форме, а в качестве замены цикла

применяется рекурсия. Каждая инструкция `continue` заменена рекурсивным вызовом, в котором в качестве аргумента используется выражение `i + 1`, позволяющее эффективно переходить к следующему целочисленному значению. Многие программисты, привыкшие к рекурсиям, считают этот стиль программирования более наглядным.

ПРИМЕЧАНИЕ

В действительности компилятор Scala не будет выдавать для кода, показанного в листинге 7.17, рекурсивную функцию. Поскольку все рекурсивные вызовы находятся в хвостовой позиции, то компилятор создаст код, похожий на цикл while. Каждый рекурсивный вызов будет реализован как возврат к началу функции. Оптимизация хвостовых вызовов рассматривается в разделе 8.9.

Если после всего сказанного вам все еще нужен `break`, то поможет стандартная библиотека Scala. Для выхода из охватывающего блока кода, помеченного как `breakable`, можно задействовать метод `break`, который предлагается в классе `Breaks` из пакета `scala.util.control`. Пример использования этого метода, предоставляемого библиотекой, выглядит следующим образом:

```
import scala.util.control.Breaks._
import java.io._

val in = new BufferedReader(new InputStreamReader(System.in))

breakable {
  while (true) {
    println("? ")
    if (in.readLine() == "") break
  }
}
```

Этот код будет многократно считывать непустые строки со стандартного устройства ввода. Как только пользователь введет пустую строку, поток управления выполнит выход из охватывающего блока `breakable`, а вместе с ним и из цикла `while`.

В классе `Breaks` метод `break` реализуется путем генерации исключения, перехватываемого приложением, которое охватывает метод в блоке `breakable`. Поэтому вызову данного метода не нужно быть в том же методе, в котором вызывается блок `breakable`.

7.7. Область видимости переменных

Теперь, когда стала понятна суть встроенных управляющих конструкций Scala, мы воспользуемся ими, чтобы объяснить, как в этом языке работает область видимости.

УСКОРЕННЫЙ РЕЖИМ ЧТЕНИЯ ДЛЯ JAVA-ПРОГРАММИСТОВ

Если вы программировали на Java, то увидите, что правила области видимости в Scala почти идентичны правилам, которые действуют в Java. Единственное различие между языками состоит в том, что в Scala позволяется определять переменные с одинаковыми именами во вложенных областях видимости. Поэтому Java-программистам может быть интересно по крайней мере бегло просмотреть данный раздел.

У объявлений переменных в программах Scala есть *область видимости*, определяющая, где конкретно может использоваться имя переменной. Самый распространенный пример определения области видимости — применение фигурных скобок, которые, как правило, вводят новую область видимости, и все, что определяется внутри этих скобок, теряет видимость после закрывающей скобки[1]. Рассмотрим в качестве иллюстрации функцию, показанную в листинге 7.18.

Листинг 7.18. Область видимости переменных при выводе на стандартное устройство таблицы умножения

```
def printMultiTable() = {

  var i = 1
  // Здесь в области видимости только переменная i

  while (i <= 10) {

    var j = 1
    // Здесь в области видимости обе переменные, как i, так и j

    while (j <= 10) {

      val prod = (i * j).toString
      // Здесь в области видимости i, j и prod

      var k = prod.length
      // Здесь в области видимости i, j, prod и k

      while (k < 4) {
        print(" ")
        k += 1
      }

      print(prod)
      j += 1
    }
```

[1] Из этого правила есть несколько исключений, поскольку в Scala иногда можно использовать вместо круглых фигурные скобки. Один из примеров подобного использования фигурных скобок — альтернативный синтаксис выражения for, рассмотренный в разделе 7.3.

```
    // i и j все еще в области видимости, а prod и k — уже нет

    println()
    i += 1
  }

  // i все еще в области видимости, а j, prod и k — уже нет
}
```

Показанная здесь функция `printMultiTable` выводит таблицу умножения[1]. В первой инструкции этой функции вводится переменная `i`, которая инициализируется целым числом 1. Затем имя `i` можно использовать во всей остальной части функции.

Следующая инструкция в `printMultiTable` является циклом `while`:

```
while (i <= 10) {

  var j = 1
  ...
}
```

Переменная `i` может использоваться здесь, поскольку по-прежнему находится в области видимости. В первой инструкции внутри цикла `while` вводится еще одна переменная, которой дается имя `j`, и она также инициализируется значением `1`. Переменная `j` была определена внутри открытого блока с фигурными скобками, который относится к циклу `while`, поэтому может использоваться только внутри данного цикла `while`. При попытке что-либо сделать с `j` после закрывающей фигурной скобки цикла `while` после комментария, сообщающего, что `j`, `prod` и `k` уже вне области видимости, ваша программа не будет скомпилирована.

Все переменные, определенные в этом примере: `i`, `j`, `prod` и `k` — локальные. Они локальны по отношению к функциям, в которых определены. При каждом вызове функции используется новый набор локальных переменных.

После определения переменной определить новую переменную с таким же именем в той же области видимости уже нельзя. Например, следующий скрипт с двумя переменными по имени `a` в одной и той же области видимости скомпилирован не будет:

```
val a = 1
val a = 2 // Не скомпилируется
println(a)
```

В то же время во внешней области видимости вполне возможно определить переменную с точно таким же именем, что и во внутренней области видимости. Следующий скрипт будет скомпилирован и сможет быть запущен:

```
val a = 1;
{
```

[1] Функция printMultiTable, показанная в листинге 7.18, написана в императивном стиле. В следующем разделе она будет приведена к функциональному стилю.

```
  val a = 2 // Компилируется без проблем
  println(a)
}
println(a)
```

При выполнении данный скрипт выведет 2, а затем 1, поскольку переменная a, определенная внутри фигурных скобок, — это уже другая переменная, область видимости которой распространяется только до закрывающей фигурной скобки[1]. Следует отметить одно различие между Scala и Java. Оно состоит в том, что Java не позволит создать во внутренней области видимости переменную, имя которой совпадает с именем переменной во внешней области видимости. В программе на Scala внутренняя переменная, как говорят, *перекрывает* внешнюю переменную с точно таким же именем, поскольку внешняя переменная становится невидимой во внутренней области видимости.

Вы уже, вероятно, замечали что-либо подобное эффекту перекрытия в интерпретаторе:

```
scala> val a = 1
a: Int = 1

scala> val a = 2
a: Int = 2

scala> println(a)
2
```

Там имена переменных можно использовать повторно как вам угодно. Среди прочего, это позволяет вам передумать, если при первом определении переменной в интерпретаторе была допущена ошибка. Подобная возможность появляется благодаря тому, что интерпретатор концептуально создает для каждой введенной вами инструкции новую вложенную область видимости. Таким образом, можно визуализировать ранее введенный код следующим образом:

```
val a = 1;
{
  val a = 2;
  {
    println(a)
  }
}
```

Этот код будет скомпилирован и запущен как скрипт на Scala и, подобно коду, набранному в интерпретаторе, выведет 2. Следует помнить: такой код может сильно запутать читателей, поскольку имена переменных приобретают во вложенных областях видимости совершенно новый смысл. Зачастую вместо того, чтобы перекрывать внешнюю переменную, лучше выбрать для переменной новое узнаваемое имя.

---

[1] Кстати, в данном случае после первого определения нужно поставить точку с запятой, поскольку в противном случае действующий в Scala механизм, который подразумевает их использование, не сработает.

7.8. Рефакторинг кода, написанного в императивном стиле

Чтобы помочь вам вникнуть в функциональный стиль, в данном разделе мы проведем рефакторинг императивного подхода к выводу таблицы умножения, показанной в листинге 7.18. Наша функциональная альтернатива представлена в листинге 7.19.

Листинг 7.19. Функциональный способ создания таблицы умножения

```
// возвращение строчки в виде последовательности
def makeRowSeq(row: Int) =
  for (col <- 1 to 10) yield {
    val prod = (row * col).toString
    val padding = " " * (4 – prod.length)
    padding + prod
  }

// возвращение строчки в виде строкового значения
def makeRow(row: Int) = makeRowSeq(row).mkString

// возвращение таблицы в виде строковых значений, по одному значению
// на каждую строчку
def multiTable() = {

  val tableSeq = // последовательность строк из строчек таблицы
    for (row <- 1 to 10)
    yield makeRow(row)

  tableSeq.mkString("\n")
}
```

На наличие в листинге 7.18 (см. выше) императивного стиля указывают два момента. Первый — побочный эффект от вызова `printMultiTable` — вывод таблицы умножения на стандартное устройство. В листинге 7.19 функция реорганизована таким образом, чтобы возвращать таблицу умножения в виде строкового значения. Поскольку функция больше не занимается выводом на стандартное устройство, то переименована в `multiTable`. Как уже упоминалось, одно из преимуществ функций, не имеющих побочных эффектов, — упрощение их модульного тестирования. Для тестирования `printMultiTable` понадобилось бы каким-то образом переопределять `print` и `println`, чтобы можно было проверить вывод на корректность. А протестировать `multiTable` гораздо проще — проверив ее строковой результат.

Второй момент, служащий верным признаком императивного стиля в функции `printMultiTable` — ее цикл `while` и `var`-переменные. В отличие от этого в функции `multiTable` для выражений, вспомогательных *функций* и вызовов `mkString` используются `val`-переменные.

Чтобы облегчить чтение кода, мы выделили две вспомогательные функции: `makeRowSeq` и `makeRow`. Первая использует выражение `for`, генератор которого перебирает номера столбцов от 1 до 10. В теле этого выражения вычисляется

произведение значения строки на значение столбца, определяется отступ, необходимый для произведения, выдается результат объединения строк отступа и произведения. Результатом выражения `for` будет последовательность (один из подклассов `scala.Seq`), содержащая выданные строки в качестве элементов. Вторая вспомогательная функция, `makeRow`, просто вызывает метод `mkString` в отношении результата, возвращенного функцией `makeRowSeq`. Этот метод объединяет имеющиеся в последовательности строки, возвращая их в виде одной строки.

Метод `multiTable` сначала инициализирует `tableSeq` результатом выполнения выражения `for`, генератор которого перебирает числа от 1 до 10, чтобы для каждого вызова `makeRow` получалось строковое значение для данной строки таблицы. Именно эта строка и выдается, вследствие чего результатом выполнения данного выражения `for` будет последовательность строковых значений, представляющих строки таблицы. Остается лишь преобразовать последовательность строк в одну строку. Для выполнения этой задачи вызывается метод `mkString`, и, поскольку ему передается значение `"\n"`, мы получаем символ конца строки, вставленный после каждой строки. Передав строку, возвращенную `multiTable`, функции `println`, вы увидите, что выводится точно такая же таблица, что и при вызове функции `printMultiTable`:

```
  1  2  3  4  5  6  7  8  9  10
  2  4  6  8 10 12 14 16 18  20
  3  6  9 12 15 18 21 24 27  30
  4  8 12 16 20 24 28 32 36  40
  5 10 15 20 25 30 35 40 45  50
  6 12 18 24 30 36 42 48 54  60
  7 14 21 28 35 42 49 56 63  70
  8 16 24 32 40 48 56 64 72  80
  9 18 27 36 45 54 63 72 81  90
 10 20 30 40 50 60 70 80 90 100
```

Резюме

Перечень встроенных в Scala управляющих конструкций минимален, но они вполне справляются со своими задачами. Их работа похожа на действия их императивных эквивалентов, но, поскольку им свойственно выдавать значение, они поддерживают и функциональный стиль. Не менее важно и то, что управляющие конструкции внимательны к тому, что опускают, оставляя, таким образом, поле деятельности для одной из самых эффективных возможностей Scala — функциональных литералов, которые рассматриваются в следующей главе.

8 Функции и замыкания

По мере роста программ появляется необходимость разбивать их на небольшие части, которыми удобнее управлять. Разделение потока управления в Scala реализуется с помощью подхода, знакомого всем опытным программистам: разделение кода на функции. Фактически в Scala предлагается несколько способов определения функций, которых нет в Java. Кроме методов, представляющих собой функции, являющиеся частью объектов, есть также функции, вложенные в другие функции, функциональные литералы и функциональные значения. В данной главе вам предстоит познакомиться со всеми этими разновидностями функций, имеющимися в Scala.

8.1. Методы

Наиболее распространенный способ определения функций — включение их в состав объекта. Такая функция называется *методом*. В качестве примера в листинге 8.1 показаны два метода, которые вместе считывают данные из файла с заданным именем и выводят строки, длина которых превышает заданную. Перед каждой выведенной строкой указывается имя файла, в котором она появляется.

Листинг 8.1. LongLines с приватным методом processLine

```
import scala.io.Source

object LongLines {

  def processFile(filename: String, width: Int) = {
    val source = Source.fromFile(filename)
    for (line <- source.getLines())
      processLine(filename, width, line)
  }
```

```
  private def processLine(filename: String,
      width: Int, line: String) = {

    if (line.length > width)
      println(filename + ": " + line.trim)
  }
}
```

Метод `processFile` получает в качестве параметров имя файла и длину строки. Он создает объект `Source`, и в отношении этого объекта в генераторе выражения `for` вызывается метод `getLines`. Как упоминалось в шаге 12 главы 3, метод `getLines` возвращает итератор, который при каждой итерации выдает одну строку из файла, исключая при этом символ конца строки. Выражение `for` обрабатывает каждую из этих строк путем вызова вспомогательного метода `processLine`. Последний получает три параметра: имя файла (`filename`), длину строки (`width`) и строку (`line`). Он проверяет длину строки на превышение заданной длины и при положительном результате выводит имя файла, двоеточие и строку.

Чтобы использовать `LongLines` из командной строки, мы создадим приложение, которое ожидает применения длины строки в качестве первого аргумента командной строки и интерпретирует последующие аргументы в качестве имен файлов[1]:

```
object FindLongLines {
  def main(args: Array[String]) = {
    val width = args(0).toInt
    for (arg <- args.drop(1))
      LongLines.processFile(arg, width)
  }
}
```

Для поиска в `LongLines.scala` строк, длина которых превышает 45 символов (там всего одна такая строка), это приложение можно использовать следующим образом:

```
$ scala FindLongLines 45 LongLines.scala
LongLines.scala: def processFile(filename: String, width: Int) = {
```

До сих пор все это было очень похоже на то, что бы вы делали в любом объектно-ориентированном языке. Но концепция функции в Scala намного универсальнее применения простого метода. В данном языке программирования предлагаются другие способы выражения функций; они рассматриваются в следующих разделах.

---

[1] В примерах приложений, которые даются в книге, проверка аргументов командной строки чаще всего проводиться не будет как из соображений экономии лесных угодий, так и в силу стремления избавиться от повторяющегося кода, который может заслонить собой важный код примеров. Издержки такого решения будут выражаться в том, что вместо выдачи информативного сообщения об ошибке в случае введения неверных данных наши примеры приложений будут генерировать исключение.

8.2. Локальные функции

Конструкция метода `processFile` из предыдущего раздела показала важность принципов разработки, присущих функциональному стилю программирования: программы должны быть разбиты на множество небольших функций с четко определенными задачами. Зачастую отдельно взятая функция весьма невелика. Преимущество подобного стиля — в предоставлении программисту множества строительных блоков, позволяющих составлять гибкие композиции для решения более сложных задач. Такие строительные блоки должны быть довольно простыми, чтобы с ними было легче разобраться по отдельности.

Подобный подход выявляет одну проблему: имена всех вспомогательных функций могут засорять пространство имен программы. В интерпретаторе это проявляется не так ярко, но по мере упаковки функций в многократно применяемые классы и объекты желательно будет скрыть вспомогательные функции от пользователей класса. Зачастую, будучи отдельно взятыми, такие функции не имеют особого смысла, и возникает желание сохранять достаточную степень гибкости, чтобы можно было удалить вспомогательные функции, если позже класс будет переписан.

В Java основной инструмент для этого — приватный метод. Как было показано в листинге 8.1, точно такой же подход с использованием приватного метода работает и в Scala, но в этом языке предлагается и еще один: можно определить функцию внутри другой функции. Подобно локальным переменным, такие локальные функции видны только в пределах своего приватного блока. Рассмотрим пример:

```
def processFile(filename: String, width: Int) = {

  def processLine(filename: String,
      width: Int, line: String) = {

    if (line.length > width)
      println(filename + ": " + line.trim)
  }

  val source = Source.fromFile(filename)
  for (line <- source.getLines()) {
    processLine(filename, width, line)
  }
}
```

Здесь реорганизована исходная версия `LongLines`, показанная в листинге 8.1: приватный метод `processLine` был превращен в локальную функцию для `processFile`. Для этого был удален модификатор `private`, который мог быть применен (и нужен) только для членов класса, а определение функции `processLine` было помещено внутрь определения функции `processFile`. В качестве локальной функция `processLine` находится в области видимости внутри функции `processFile`, за пределами которой она недоступна.

Но теперь, когда функция `processLine` определена внутри функции `processFile`, появилась возможность улучшить кое-что еще. Вы заметили, что `filename` и `width` передаются вспомогательной функции, как и прежде? В этом нет никакой необходимости, поскольку локальные функции могут получать доступ к параметрам охватывающей их функции. Как показано в листинге 8.2, можно просто воспользоваться параметрами внешней функции `processFile`.

Листинг 8.2. LongLines с локальной функцией processLine

```
import scala.io.Source

object LongLines {

  def processFile(filename: String, width: Int) = {

    def processLine(line: String) = {
      if (line.length > width)
        println(filename + ": " + line.trim)
    }

    val source = Source.fromFile(filename)
    for (line <- source.getLines())
      processLine(line)
  }
}
```

Заметили, как упростился код? Такое использование параметров охватывающей функции — широко распространенный и весьма полезный пример универсальной вложенности, предоставляемой Scala. Вложенность и области видимости применительно ко всем конструкциям данного языка, включая функции, рассматриваются в разделе 7.7. Это довольно простой, но весьма эффективный принцип, особенно в языках, использующих функции первого класса.

8.3. Функции первого класса

В Scala есть *функции первого класса*. Вы можете не только определить их и вызвать, но и записать в виде безымянных *литералов*, после чего передать их в качестве *значений*. Понятие функциональных литералов было введено в главе 2, а их основной синтаксис показан на рис. 2.2 (см. выше).

Функциональный литерал компилируется в класс, который при создании экземпляра во время выполнения программы становится *функциональным значением*[1].

[1] Каждое функциональное значение является экземпляром какого-нибудь класса, который представляет собой расширение одного из нескольких трейтов FunctionN в пакете scala, например Function0 для функций без параметров, Function1 для функций с одним параметром и т. д. В каждом трейте FunctionN имеется метод apply, используемый для вызова функции.

Таким образом, разница между функциональными литералами и значениями состоит в том, что первые существуют в исходном коде, а вторые — в виде объектов во время выполнения программы. Эта разница во многом похожа на разницу между классами (исходным кодом) и объектами (создаваемыми во время выполнения программы).

Простой функциональный литерал, прибавляющий к числу единицу, имеет следующий вид:

```
(x: Int) => x + 1
```

Сочетание символов `=>` указывает на то, что эта функция превращает стоящее слева от данного сочетания (любое целочисленное значение `x`) в то, что находится справа от него (`x + 1`). Таким образом, данная функция отображает на любую целочисленную переменную `x` значение `x + 1`.

Функциональные значения — это объекты, следовательно, при желании их можно хранить в переменных. Они также являются функциями, следовательно, вы можете вызывать их, используя обычную форму записи вызова функций с применением круглых скобок. Вот как выглядят примеры обоих действий:

```
scala> var increase = (x: Int) => x + 1
increase: Int => Int = $$Lambda$988/1232424564@cf01c2e

scala> increase(10)
res0: Int = 11
```

Поскольку `increase` в данном примере является `var`-переменной, то позже ей можно присвоить другое значение:

```
scala> increase = (x: Int) => x + 9999
mutated increase

scala> increase(10)
res1: Int = 10009
```

Если нужно, чтобы в функциональном литерале использовалось более одной инструкции, то следует заключить его тело в фигурные скобки и поместить каждую инструкцию на отдельной строке, сформировав блок. Как и в случае создания метода, когда вызывается функциональное значение, будут выполнены все инструкции и значением, возвращаемым из функции, станет значение, получаемое при вычислении последнего выражения:

```
scala> increase = (x: Int) => {
         println("We")
         println("are")
         println("here!")
         x + 1
       }
mutated increase
```

```
scala> increase(10)
We
are
here!
res2: Int = 11
```

Итак, вы увидели все основные составляющие функциональных литералов и функциональных значений. Возможности их применения обеспечиваются многими библиотеками Scala. Например, методом `foreach`, доступным для всех коллекций[1]. Он получает функцию в качестве аргумента и вызывает ее в отношении каждого элемента своей коллекции. А вот как его можно использовать для вывода всех элементов списка:

```
scala> val someNumbers = List(-11, -10, -5, 0, 5, 10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumbers.foreach((x: Int) => println(x))
-11
-10
-5
0
5
10
```

В другом примере также используется имеющийся у типов коллекций метод `filter`. Он выбирает те элементы коллекции, которые проходят выполняемую пользователем проверку с применением функции. Например, фильтрацию можно осуществить с помощью функции `(x: Int) => x > 0`. Она отображает положительные целые числа в `true`, а все остальные числа — в `false`. Метод `filter` можно задействовать следующим образом:

```
scala> someNumbers.filter((x: Int) => x > 0)
res4: List[Int] = List(5, 10)
```

Более подробно о методах, подобных `foreach` и `filter`, поговорим позже. В главе 16 рассматривается их использование в классе `List`, а в главе 17 — применение с другими типами коллекций.

8.4. Краткие формы функциональных литералов

В Scala есть несколько способов избавления от избыточной информации, позволяющих записывать функциональные литералы более кратко. Не упускайте такой возможности, поскольку она позволяет вам убрать из своего кода ненужный хлам.

[1] Метод foreach определен в трейте Iterable, который является супертрейтом для List, Set, Array и Map. Подробности — в главе 17.

Один из способов писать функциональные литералы более лаконично заключается в отбрасывании типов параметров. При этом предыдущий пример с фильтром может быть написан следующим образом:

```
scala> someNumbers.filter((x) => x > 0)
res5: List[Int] = List(5, 10)
```

Компилятор Scala знает, что переменная `x` должна относиться к целым числам, поскольку видит, что вы сразу же применяете функцию для фильтрации списка целых чисел, на который ссылается объект `someNumbers`. Это называется *целевой типизацией*, поскольку целевому использованию выражения — в данном случае в качестве аргумента для `someNumbers.filter()` — разрешено влиять на типизацию выражения — в данном случае на определение типа параметра `x`. Подробности целевой типизации нам сейчас неважны. Вы можете просто приступить к написанию функционального литерала без типа аргумента и, если компилятор не сможет в нем разобраться, добавить тип. Со временем вы начнете понимать, в каких ситуациях компилятор сможет решить эту загадку, а в каких — нет.

Второй способ избавления от избыточных символов заключается в отказе от круглых скобок вокруг параметра, тип которого будет выведен автоматически. В предыдущем примере круглые скобки вокруг `x` совершенно излишни:

```
scala> someNumbers.filter(x => x > 0)
res6: List[Int] = List(5, 10)
```

8.5. Синтаксис заместителя

Можно сделать функциональный литерал еще короче, воспользовавшись знаком подчеркивания в качестве заместителя для одного или нескольких параметров при условии, что каждый параметр появляется внутри функционального литерала только один раз. Например, `_ > 0` — очень краткая форма записи для функции, проверяющей, что значение больше нуля:

```
scala> someNumbers.filter(_ > 0)
res7: List[Int] = List(5, 10)
```

Знак подчеркивания можно рассматривать как бланк, который следует заполнить. Он будет заполнен аргументом функции при каждом ее вызове. Например, при условии, что переменная `someNumbers` была здесь инициализирована значением `List(-11, -10, -5, 0, 5, 10)`, метод `filter` заменит бланк в `_ > 0` сначала значением `-11`, получив `-11 > 0`, затем значением `-10`, получив `-10 > 0`, затем значением `-5`, получив `-5 > 0`, и так далее до конца списка `List`. Таким образом, функциональный литерал `_ > 0` является, как здесь показано, эквивалентом немного более пространного литерала `x => x > 0`:

```
scala> someNumbers.filter(x => x > 0)
res8: List[Int] = List(5, 10)
```

Иногда при использовании знаков подчеркивания в качестве заместителей параметров у компилятора может оказаться недостаточно информации для вывода неуказанных типов параметров. Предположим, к примеру, что вы сами написали `_ + _`:

```
scala> val f = _ + _
               ^
       error: missing parameter type for expanded function
((x$1: <error>, x$2) => x$1.$plus(x$2))
```

В таких случаях нужно указать типы, используя двоеточие:

```
scala> val f = (_: Int) + (_: Int)
f: (Int, Int) => Int = $$Lambda$1075/1481958694@289fff3c

scala> f(5, 10)
res9: Int = 15
```

Следует заметить, что `_ + _` расширяется в литерал для функции, получающей два параметра. Поэтому сокращенную форму можно использовать, только если каждый параметр применяется в функциональном литерале не более одного раза. Несколько знаков подчеркивания означают наличие нескольких параметров, а не многократное использование одного и того же параметра. Первый знак подчеркивания представляет первый параметр, второй знак — второй параметр, третий знак — третий параметр и т. д.

8.6. Частично примененные функции

В предыдущих примерах знаки подчеркивания ставились вместо отдельных параметров, но можно заменить знаком подчеркивания и весь список параметров. Например, вместо `println(_)` можно воспользоваться кодом `println _`. Рассмотрим пример:

```
someNumbers.foreach(println _)
```

В Scala эта краткая форма интерпретируется так, будто вы воспользовались следующим кодом:

```
someNumbers.foreach(x => println(x))
```

Таким образом, знак подчеркивания в данном случае не является заместителем отдельного параметра. Он замещает весь список параметров. Не забудьте поставить пробел между именем функции и знаком подчеркивания. В противном случае компилятор решит, что вы подразумеваете ссылку на другое обозначение, такое как обозначение метода с именем `println_`, которого, скорее всего, не существует.

При использовании знака подчеркивания подобным образом создается *частично примененная функция*. В Scala, когда при вызове функции в нее передаются

любые необходимые аргументы, вы *применяете* эту функцию к этим аргументам. Например, если есть следующая функция:

```
scala> def sum(a: Int, b: Int, c: Int) = a + b + c
sum: (a: Int, b: Int, c: Int)Int
```

то функцию `sum` можно применить к аргументам `1`, `2` и `3` таким образом:

```
scala> sum(1, 2, 3)
res10: Int = 6
```

Частично примененная функция — выражение, в котором не содержатся все аргументы, необходимые функции. Вместо этого в ней есть лишь некоторые из них или вообще нет никаких необходимых аргументов. Например, чтобы создать выражение частично примененной функции, вызывающее `sum`, в котором вы не предоставляете ни одного из трех требуемых аргументов, вы помещаете знак подчеркивания после `sum`. Получившуюся функцию можно затем сохранить в переменной. Рассмотрим пример:

```
scala> val a = sum _
a: (Int, Int, Int) => Int = $$Lambda$1091/1149618736@6415112c
```

Если данный код есть, то компилятор Scala создает экземпляр функционального значения, который получает три целочисленных параметра, не указанных в выражении частично примененной функции, `sum _`, и присваивает ссылку на это новое функциональное значение переменной `a`. Когда к этому новому значению применяются три аргумента, оно развернется и вызовет `sum`, передав в нее те же три аргумента:

```
scala> a(1, 2, 3)
res11: Int = 6
```

Происходит следующее: переменная по имени `a` ссылается на объект функционального значения. Это функциональное значение является экземпляром класса, сгенерированного автоматически компилятором Scala из `sum _` — выражения частично примененной функции. Класс, сгенерированный компилятором, имеет метод `apply`, получающий три аргумента[1]. Имеющийся метод `apply` получает три аргумента, поскольку это и есть количество аргументов, отсутствующих в выражении `sum _`. Компилятор Scala транслирует выражение `a(1, 2, 3)` в вызов метода `apply`, принадлежащего объекту функционального значения, передавая ему три аргумента: `1`, `2` и `3`. Таким образом, `a(1, 2, 3)` — краткая форма следующего кода:

```
scala> a.apply(1, 2, 3)
res12: Int = 6
```

[1] Сгенерированный класс является расширением трейта Function3, в котором объявлен метод apply, предусматривающий использование трех аргументов.

Этот метод `apply`, который определен в автоматически генерируемом компилятором Scala классе из выражения `sum _`, просто передает дальше эти три отсутствовавших параметра функции `sum` и возвращает результат. В данном случае метод `apply` вызывает `sum(1, 2, 3)` и возвращает то, что возвращает функция `sum`, то есть число `6`.

Данный вид выражений, в которых знак подчеркивания используется для представления всего списка параметров, можно представить себе и по-другому — в качестве способа преобразования `def` в функциональное значение. Например, если имеется локальная функция, скажем, `sum(a: Int, b: Int, c: Int): Int`, то ее можно завернуть в функциональное значение, чей метод `apply` имеет точно такие же типы списка параметров и результата. Это функциональное значение, будучи примененным к неким аргументам, в свою очередь, применяет `sum` для тех же самых аргументов и возвращает результат. Вы не можете присвоить переменной метод или вложенную функцию или передать их в качестве аргументов другой функции. Однако все это можно сделать, если завернуть метод или вложенную функцию в функциональное значение, поместив знак подчеркивания после имени метода или вложенной функции.

Теперь, несмотря на то что `sum _` действительно является частично примененной функцией, вам может быть не вполне понятно, почему она называется именно так. Это потому, что она применяется не ко всем своим аргументам. Что касается `sum _`, то она не применяется *ни к одному* из своих аргументов. Но вы также можете выразить частично примененной функцию, предоставив ей только *некоторые* из требуемых аргументов. Рассмотрим пример:

```
scala> val b = sum(1, _: Int, 3)
b: Int => Int = $$Lambda$1092/457198113@280aa1bd
```

В данном случае функции `sum` предоставлены первый и последний аргументы, а средний аргумент не указан. Поскольку пропущен только один аргумент, то компилятор Scala сгенерирует новый функциональный класс, чей метод `apply` получает один аргумент. При вызове с этим одним аргументом метод `apply` сгенерированной функции вызывает функцию `sum`, передавая ей `1`, затем аргумент, переданный функции, и, наконец, `3`. Рассмотрим несколько примеров:

```
scala> b(2)
res13: Int = 6
```

В этом случае `b.apply` вызывает `sum(1, 2, 3)`:

```
scala> b(5)
res14: Int = 9
```

А в этом случае `b.apply` вызывает `sum(1, 5, 3)`.

Если функция требуется в конкретном месте кода, то при написании выражения частично примененной функции, в котором не указан ни один параметр, например `println _` или `sum _`, его можно записать более кратко, не указывая знак

подчеркивания. Например, вместо следующей записи вывода каждого числа, имеющегося в объекте `someNumbers` (который определен в данном коде):

```
someNumbers.foreach(println _)
```

можно просто воспользоваться кодом:

```
someNumbers.foreach(println)
```

Последняя форма допустима только в тех местах, где требуется функция, например, как вызов `foreach` в данном примере. Компилятор знает, что в данном случае нужна функция, поскольку `foreach` требует ее передачи в качестве аргумента. В тех ситуациях, когда функция не нужна, попытки применить данную форму записи приведут к ошибке компиляции. Рассмотрим пример:

```
scala> val c = sum
       error: missing argument list for method sum
       Unapplied methods are only converted to functions when
       a function type is expected.
       You can make this conversion explicit by writing `sum
       _` or `sum(_,_,_)` instead of `sum`.

scala> val d = sum _
d: (Int, Int, Int) => Int = $$Lambda$1095/598308875@12223aed

scala> d(10, 20, 30)
res14: Int = 60
```

8.7. Замыкания

Все рассмотренные до сих пор в этой главе примеры функциональных литералов ссылались только на передаваемые параметры. Так, в выражении `(x: Int) => x > 0` в теле функции `x > 0` использовалась только одна переменная, `x`, которая объявлена как параметр функции. Но вы можете ссылаться на переменные, объявленные и в других местах:

```
(x: Int) => x + more // На сколько больше?
```

Эта функция прибавляет значение переменной `more` к своему аргументу, но что такое `more`? С точки зрения данной функции `more` — *свободная переменная*, поскольку в самом функциональном литерале значение ей не присваивается. В отличие от нее переменная `x` является *связанной*, поскольку в контексте функции имеет значение: определена как единственный параметр функции, имеющий тип `Int`. Если попытаться воспользоваться этим функциональным литералом в чистом виде, без каких-либо определений в его области видимости, то компилятор выразит недовольство:

```
scala> (x: Int) => x + more
                       ^
       error: not found: value more
```

Зачем нужен конечный знак подчеркивания

Синтаксис, используемый в Scala для частично примененных функций, подчеркивает разницу между компромиссами дизайна Scala и классических функциональных языков, таких как Haskell или ML. В этих языках использование частично примененных функций в порядке вещей. Более того, в них применяется весьма строгая система статической типизации, которая обычно позволяет обнаружить каждую ошибку, допущенную вами в использовании частично примененных функций. Scala в этом смысле больше похож на императивные языки, такие как Java, где метод, не применяемый ко всем своим аргументам, считается ошибкой. Более того, объектно-ориентированные традиции порождения подтипов и универсальный корневой тип, допускают программы, которые в классических функциональных языках считались бы ошибочными.

К примеру, вы неправильно применили метод `drop(n: Int)` класса `List` для `tail`, забыв, что нужно передать число для метода `drop`. Вы могли создать код `println(xs.drop)`. Если бы Scala придерживался классической функциональной традиции, согласно которой частично примененные функции могут находиться где угодно, то этот код проходил бы проверку типа. Но, возможно, к вашему удивлению, инструкция `println` всегда бы выводила `<function>`! Произошло то, что выражение `drop` стало рассматриваться как функциональный объект. Поскольку `println` принимает объекты любого типа, код успешно проходит компиляцию, но при этом возвращает весьма неожиданный результат.

Чтобы избежать подобных ситуаций, в Scala обычно требуется обозначить специально пропущенные аргументы функции, даже если в качестве такого обозначения используется просто знак подчеркивания (`_`). Scala позволяет обходиться даже без него, но только когда ожидается функциональный тип.

С другой стороны, тот же функциональный литерал будет нормально работать, пока будет доступно нечто с именем `more`:

```
scala> var more = 1
more: Int = 1

scala> val addMore = (x: Int) => x + more
addMore: Int => Int = $$Lambda$1103/2125513028@11cb348c

scala> addMore(10)
res16: Int = 11
```

Функциональное значение (объект), создаваемое во время выполнения программы из этого функционального литерала, называется *замыканием*. Данное название появилось из-за «замыкания» функционального литерала путем «захвата» привязок его свободных переменных. Функциональный литерал, не имеющий свободных переменных, например `(x: Int) => x + 1`, называется *замкнутым термом*, где *терм* — это фрагмент исходного кода. Таким образом, функциональное значение, созданное во время выполнения программы из этого функционального литерала, строго говоря, не является замыканием, поскольку функциональный литерал

`(x: Int) => x + 1` всегда замкнут уже по факту его написания. Но любой функциональный литерал со свободными переменными, например `(x: Int) => x + more`, является *открытым термом*. Поэтому любое функциональное значение, созданное во время выполнения программы из `(x: Int) => x + more`, будет по определению требовать, чтобы привязка его свободной переменной, `more`, была захвачена. Получившееся функциональное значение, в котором может содержаться ссылка на захваченную переменную `more`, называется замыканием, поскольку функциональное значение — конечный продукт замыкания открытого терма, `(x: Int) => x + more`.

Этот пример вызывает вопрос: что случится, если значение `more` изменится после создания замыкания? В Scala можно ответить, что замыкание видит изменение, например:

```
scala> more = 9999
mutated more

scala> addMore(10)
res17: Int = 10009
```

На интуитивном уровне понятно, что замыкания в Scala перехватывают сами переменные, а не те значения, на которые те ссылаются[1]. Как показано в предыдущем примере, замыкание, созданное для `(x: Int) => x + more`, видит изменение `more` за пределами замыкания. То же самое справедливо и в обратном направлении. Изменения, вносимые в захваченную переменную, видимы за пределами замыкания. Рассмотрим пример:

```
scala> val someNumbers = List(-11, -10, -5, 0, 5, 10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> var sum = 0
sum: Int = 0

scala> someNumbers.foreach(sum += _)

scala> sum
res19: Int = -11
```

Здесь используется обходной способ сложения чисел в списке типа `List`. Переменная `sum` находится в области видимости, охватывающей функциональный литерал `sum += _`, который прибавляет числа к `sum`. Несмотря на то что замыкание модифицирует `sum` во время выполнения программы, получающийся конечный результат `-11` по-прежнему виден за пределами замыкания.

А что, если замыкание обращается к некой переменной, у которой во время выполнения программы есть несколько копий? Например, что, если замыкание

[1] В отличие от этого внутренние классы Java вообще не позволяют обращаться к модифицируемым переменным в окружающей области видимости, и потому нет никакой разницы между захватом переменной и захватом того значения, которое в ней находится в данный момент.

использует локальную переменную некой функции и последняя вызывается множество раз? Какой из экземпляров этой переменной будет задействован при каждом обращении?

С остальной частью языка согласуется только один ответ: задействуется тот экземпляр, который был активен на момент создания замыкания. Рассмотрим, к примеру, функцию, создающую и возвращающую замыкания прироста значения:

```
def makeIncreaser(more: Int) = (x: Int) => x + more
```

При каждом вызове эта функция будет создавать новое замыкание. Каждое замыкание будет обращаться к той переменной `more`, которая была активна при создании замыкания.

```
scala> val inc1 = makeIncreaser(1)
inc1: Int => Int = $$Lambda$1126/1042315811@4262a8d2

scala> val inc9999 = makeIncreaser(9999)
inc9999: Int => Int = $$Lambda$1126/1042315811@4c8bbc5e
```

При вызове `makeIncreaser(1)` создается и возвращается замыкание, захватывающее в качестве привязки к `more` значение `1`. По аналогии с этим при вызове `makeIncreaser(9999)` возвращается замыкание, захватывающее для `more` значение `9999`. Когда эти замыкания применяются к аргументам (в данном случае имеется только один передаваемый аргумент, `x`), получаемый результат зависит от того, как переменная `more` была определена в момент создания замыкания:

```
scala> inc1(10)
res20: Int = 11

scala> inc9999(10)
res21: Int = 10009
```

И неважно, что `more` в данном случае — параметр вызова метода, из которого уже произошел возврат. В подобных случаях компилятор Scala осуществляет реорганизацию, которая позволяет захваченным параметрам продолжать существовать в динамической памяти (куче), а не в стеке, и пережить таким образом создавший их метод. Данная реорганизация происходит в автоматическом режиме, поэтому вам о ней не стоит беспокоиться. Захватывайте какую угодно переменную: `val` или `var` или любой параметр.

8.8. Специальные формы вызова функций

Большинство встречающихся вам функций и вызовов функций будут аналогичны уже увиденным в этой главе. У функции будет фиксированное число параметров, у вызова будет точно такое же количество аргументов, и аргументы будут указаны в точно таких же порядке и количестве, что и параметры.

Но поскольку вызовы функций в программировании на Scala весьма важны, то для обеспечения некоторых специальных потребностей в язык были добавлены

несколько специальных форм определений и вызовов функций. В Scala поддерживаются повторяющиеся параметры, именованные аргументы и аргументы со значениями по умолчанию.

Повторяющиеся параметры

В Scala допускается указание на то, что последний параметр функции может повторяться. Это позволяет клиентам передавать функции список аргументов переменной длины. Чтобы обозначить повторяющийся параметр, поставьте после типа параметра знак звездочки, например:

```
scala> def echo(args: String*) =
         for (arg <- args) println(arg)
echo: (args: String*)Unit
```

Определенная таким образом функция `echo` может быть вызвана с нулем и большим количеством аргументов типа `String`:

```
scala> echo()

scala> echo("one")
one

scala> echo("hello", "world!")
hello
world!
```

Внутри функции типом повторяющегося параметра является `Seq` из элементов объявленного типа параметра. Таким образом, типом переменной `args` внутри функции `echo`, которая объявлена как тип `String*`, фактически является `Seq[String]`. Несмотря на это, если у вас имеется массив подходящего типа, при попытке передать его в качестве повторяющегося параметра будет получена ошибка компиляции:

```
scala> val seq = Seq("What's", "up", "doc?")
seq: Seq[String] = Seq(What's, up, doc?)

scala> echo(seq)
            ^
       error: type mismatch;
        found   : Seq[String]
        required: String
```

Чтобы добиться успеха, нужно после аргумента-массива поставить двоеточие, знак подчеркивания и знак звездочки:

```
scala> echo(seq: _*)
What's
up
doc?
```

Эта форма записи заставит компилятор передать в `echo` каждый элемент массива `seq` в виде самостоятельного аргумента, а не передавать его целиком в виде одного аргумента.

Именованные аргументы

При обычном вызове функции аргументы в вызове поочередно сопоставляются в указанном порядке с параметрами вызываемой функции:

```
scala> def speed(distance: Float, time: Float): Float =
         distance / time
speed: (distance: Float, time: Float)Float

scala> speed(100, 10)
res27: Float = 10.0
```

В данном вызове `100` сопоставляется с `distance`, а `10` — с `time`. Сопоставление `100` и `10` производится в том же порядке, в котором перечислены формальные параметры.

Именованные аргументы позволяют передавать аргументы функции в ином порядке. Синтаксис просто предусматривает, что перед каждым аргументом указывается имя параметра и знак равенства. Например, следующий вызов функции `speed` эквивалентен вызову `speed(100,10)`:

```
scala> speed(distance = 100, time = 10)
res28: Float = 10.0
```

При вызове с именованными аргументами эти аргументы можно поменять местами, не изменяя их значения:

```
scala> speed(time = 10, distance = 100)
res29: Float = 10.0
```

Можно также смешивать позиционные и именованные аргументы. В этом случае сначала указываются позиционные аргументы. Именованные чаще всего используются в сочетании со значениями параметров по умолчанию.

Значения параметров по умолчанию

Scala позволяет указать для параметров функции значения по умолчанию. Аргумент для такого параметра может быть произвольно опущен из вызова функции, в таком случае соответствующий аргумент будет заполнен значением по умолчанию.

Пример показан в листинге 8.3. У функции `printTime` есть один параметр, `out`, имеющий значение по умолчанию `Console.out`.

Листинг 8.3. Параметр со значением по умолчанию

```
def printTime(out: java.io.PrintStream = Console.out) =
  out.println("time = " + System.currentTimeMillis())
```

Если функцию вызвать как `printTime()`, то есть без указания аргумента, используемого для `out`, то для этого параметра будет установлено его значение по умолчанию `Console.out`. Можно также вызвать функцию с явно указанным выходным потоком. Например, вызвав функцию в виде `printTime(Console.err)`, можно отправить регистрационную запись на стандартное устройство вывода сообщений об ошибках.

Параметры по умолчанию особенно полезны, когда применяются в сочетании с именованными параметрами. В листинге 8.4 у функции `printTime2` имеется два необязательных параметра. У `out` есть значение по умолчанию `Console.out`, а у `divisor` — значение по умолчанию `1`.

Листинг 8.4. Функция с двумя параметрами, у которых имеются значения по умолчанию

```
def printTime2(out: java.io.PrintStream = Console.out,
               divisor: Int = 1) =
  out.println("time = " + System.currentTimeMillis()/divisor)
```

Чтобы оба параметра заполнялись их значениями по умолчанию, функцию `printTime2` можно вызывать как `printTime2()`. Но при использовании именованных аргументов может быть указан любой из параметров, а для другого останется значение по умолчанию. Когда необходимо указать выходной поток, вызов выглядит следующим образом:

```
printTime2(out = Console.err)
```

Указать делитель времени можно с помощью следующего вызова:

```
printTime2(divisor = 1000)
```

8.9. Хвостовая рекурсия

В разделе 7.2 упоминалось, что для преобразования цикла `while`, который обновляет значение `var`-переменных в код более функционального стиля, использующий только `val`-переменные, временами может понадобиться прибегнуть к рекурсии. Рассмотрим пример рекурсивной функции, которая вычисляет приблизительное значение, повторяя уточнение приблизительного расчета, пока не будет получен приемлемый результат:

```
def approximate(guess: Double): Double =
  if (isGoodEnough(guess)) guess
  else approximate(improve(guess))
```

С соответствующими реализациями `isGoodEnough` и `improve` подобные функции часто используются при решении задач поиска. Если нужно, чтобы функция `approximate` выполнялась быстрее, то может возникнуть желание написать ее с циклом `while`:

```
def approximateLoop(initialGuess: Double): Double = {
  var guess = initialGuess
  while (!isGoodEnough(guess))
    guess = improve(guess)
  guess
}
```

Какая из двух версий `approximate` более предпочтительна? Если требуется лаконичность и нужно избавиться от использования `var`-переменных, то выиграет первый, функциональный вариант. Но, может быть, более эффективным окажется императивный подход? На самом деле, если измерить время выполнения, окажется, что они практически одинаковы!

Этот результат может показаться неожиданным, поскольку рекурсивный вызов выглядит намного более затратным, чем простой переход из конца цикла в его начало. Но в показанном ранее вычислении приблизительного значения компилятор Scala может применить очень важную оптимизацию. Обратите внимание на то, что в вычислении тела функции `approximate` рекурсивный вызов стоит в самом конце. Функции наподобие `approximate`, которые в качестве последнего действия вызывают сами себя, называются *функциями с хвостовой рекурсией*. Компилятор Scala обнаруживает хвостовую рекурсию и заменяет ее переходом к началу функции после обновления параметров функции новыми значениями.

Из этого можно сделать вывод, что избегать использования рекурсивных алгоритмов для решения ваших задач не стоит. Зачастую рекурсивное решение выглядит более элегантно и лаконично, чем решение на основе цикла. Если в решении задействована хвостовая рекурсия, то расплачиваться за него издержками производительности во время выполнения программы не придется.

Трассировка функций с хвостовой рекурсией

Для каждого вызова функции с хвостовой рекурсией новый фрейм стека создаваться не будет, все вызовы станут выполняться с использованием одного и того же фрейма. Это обстоятельство может вызвать удивление у программиста, который исследует трассировку стека программы, давшей сбой. Например, данная функция вызывает себя несколько раз и генерирует исключение:

```
def boom(x: Int): Int =
  if (x == 0) throw new Exception("boom!")
  else boom(x - 1) + 1
```

Эта функция не относится к функциям с хвостовой рекурсией, поскольку после рекурсивного вызова выполняет операцию инкремента. При ее запуске будет получен вполне ожидаемый результат:

```
scala> boom(3)
java.lang.Exception: boom!
      at .boom(<console>:5)
      at .boom(<console>:6)
      at .boom(<console>:6)
      at .boom(<console>:6)
      at .<init>(<console>:6)
...
```

Оптимизация хвостового вызова

Код, скомпилированный для `approximate`, по сути, такой же, как и код, скомпилированный для `approximateLoop`. Обе функции компилируются в одни и те же 13 инструкций байт-кода Java. Если просмотреть байт-коды, сгенерированные компилятором Scala для метода с хвостовой рекурсией `approximate`, то можно увидеть, что, хотя и `isGoodEnough`, и `improve` вызываются в теле метода, `approximate` там не вызывается. При оптимизации компилятор Scala убирает рекурсивный вызов:

```
public double approximate(double);
  Code:
   0:   aload_0
   1:   astore_3
   2:   aload_0
   3:   dload_1
   4:   invokevirtual #24; //Method isGoodEnough:(D)Z
   7:   ifeq    12
   10:   dload_1
   11:   dreturn
   12:   aload_0
   13:   dload_1
   14:   invokevirtual #27; //Method improve:(D)D
   17:   dstore_1
   18:   goto    2
```

Если теперь внести изменения в `boom` так, чтобы в ней появилась хвостовая рекурсия:

```
def bang(x: Int): Int =
  if (x == 0) throw new Exception("bang!")
  else bang(x - 1)
```

то получится следующий результат:

```
scala> bang(5)
java.lang.Exception: bang!
      at .bang(<console>:5)
      at .<init>(<console>:6) ...
```

На сей раз вы видите только фрейм стека для `bang`. Можно подумать, `bang` дает сбой перед своим собственным вызовом, но это не так. Если вы считаете, что при просмотре трассировки стека оптимизации хвостовых вызовов могут вас запутать, то их можно выключить, указав для оболочки `scala` или компилятора `scalac` следующий ключ:

```
-g:notailcalls
```

Указав этот ключ, вы получите удлиненную трассировку стека:

```
scala> bang(5)
java.lang.Exception: bang!
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .<init>(<console>:6) ...
```

Ограничения хвостовой рекурсии

Использование хвостовой рекурсии в Scala строго ограничено, поскольку набор инструкций виртуальной машины Java (JVM) существенно затрудняет реализацию более сложных форм хвостовых рекурсий. Оптимизация в Scala касается только непосредственных рекурсивных вызовов той же самой функции, из которой выполняется вызов. Если рекурсия косвенная, как в следующем примере, где применяются две взаимно рекурсивные функции, то ее оптимизация невозможна:

```
def isEven(x: Int): Boolean =
  if (x == 0) true else isOdd(x – 1)
def isOdd(x: Int): Boolean =
  if (x == 0) false else isEven(x – 1)
```

Получить оптимизацию хвостового вызова невозможно и в том случае, если завершающий вызов делается в отношении функционального значения. Рассмотрим, к примеру, такой рекурсивный код:

```
val funValue = nestedFun _
def nestedFun(x: Int) : Unit = {
  if (x != 0) { println(x); funValue(x – 1) }
}
```

Переменная `funValue` ссылается на функциональное значение, которое, по сути, заключает в себе вызов функции `nestedFun`. В момент применения функционального значения к аргументу все изменяется и `nestedFun` применяется к тому же самому аргументу, возвращая результат. Поэтому вы можете понадеяться на то, что компилятор Scala выполнит оптимизацию хвостового вызова, но в данном случае этого не произойдет. Оптимизация хвостовых вызовов ограничивается ситуациями, когда метод или вложенная функция вызывают сами себя непосредственно

в качестве своей последней операции, не обращаясь к функциональному значению или через какого-то другого посредника. (Если вы еще не усвоили, что такое хвостовая рекурсия, то перечитайте раздел 8.9.)

Резюме

В данной главе мы представили довольно подробный обзор использования функций в Scala. Кроме методов, этот язык предоставляет локальные функции, функциональные литералы и функциональные значения. В дополнение к обычным вызовам функций в Scala используются частично примененные функции и функции с повторяющимися параметрами. При благоприятной возможности вызовы функций реализуются в виде оптимизированных хвостовых вызовов, благодаря чему многие привлекательные рекурсивные функции выполняются практически так же быстро, как и оптимизированные вручную версии, использующие циклы `while`. В следующей главе на основе этих основных положений мы покажем, как имеющаяся в Scala расширенная поддержка функций помогает абстрагировать процессы управления.

9 Управляющие абстракции

В главе 7 мы отметили, что встроенных управляющих абстракций в Scala не так уж много, поскольку этот язык позволяет вам создавать собственные управляющие абстракции. В предыдущей главе мы рассмотрели функциональные значения. В этой покажем способы применения функциональных значений в целях создания новых управляющих абстракций. Попутно рассмотрим карринг и передачу параметров по имени.

9.1. Сокращение повторяемости кода

Каждую функцию можно разделить на общую часть, одинаковую для всех вызовов функции, и особую часть, которая может варьироваться от одного вызова функции к другому. Общая часть находится в теле функции, а особая должна предоставляться через аргументы. Когда в качестве аргумента используется функциональное значение, особая часть алгоритма сама по себе является еще одним алгоритмом! При каждом вызове такой функции ей можно передавать в качестве аргумента другое функциональное значение, и вызванная функция в этом случае будет вызывать переданное функциональное значение. Такие *функции высшего порядка*, то есть функции, которые получают функции в качестве параметров, обеспечивают вам дополнительные возможности по сокращению и упрощению кода.

Одним из преимуществ функций высшего порядка является то, что они предоставляют вам возможность создавать управляющие абстракции, которые позволяют избавиться от повторяющихся фрагментов кода. Предположим, вы создаете браузер файлов и должны разработать API, разрешающий пользователям искать файлы, соответствующие какому-либо критерию. Сначала вы добавляете средство поиска тех файлов, чьи имена заканчиваются конкретной строкой. Это даст пользователям возможность найти, к примеру, все файлы с расширением `.scala`. Такой API можно создать путем определения публичного метода `filesEnding` внутри следующего объекта-одиночки:

```
object FileMatcher {
  private def filesHere = (new java.io.File(".")).listFiles
```

```
  def filesEnding(query: String) =
    for (file <- filesHere; if file.getName.endsWith(query))
      yield file
}
```

Метод `filesEnding` получает список всех файлов, находящихся в текущем каталоге, применяя приватный вспомогательный метод `filesHere`, затем фильтрует этот список по признаку, завершается ли имя файла тем содержимым, которое указано в пользовательском запросе. Поскольку `filesHere` является приватным методом, то метод `filesEnding` — единственный доступный метод, определенный в `FileMatcher`, то есть в API, который вы предлагаете своим пользователям.

Пока все идет неплохо — повторяющегося кода нет. Но чуть позже вы хотите разрешить пользователям искать по любой части имени файла. Такой поиск пригодится, когда пользователи не смогут вспомнить, как именно они назвали файл, `phb-important.doc`, `stupid-phb-report.doc`, `may2003salesdoc.phb` или совершенно иначе, и единственное, в чем они уверены, — что где-то в имени фигурирует phb. Вы возвращаетесь к работе, и к `FileMatcher` API добавляете соответствующую функцию:

```
def filesContaining(query: String) =
  for (file <- filesHere; if file.getName.contains(query))
    yield file
```

Данная функция работает точно так же, как и `filesEnding`. Она ищет текущие файлы с помощью `filesHere`, проверяет имя и возвращает файл, если его имя соответствует критерию поиска. Единственное отличие — функция использует метод `contains` вместо метода `endsWith`. Проходит несколько месяцев, и программа набирает популярность. Со временем вы уступаете просьбам некоторых активных пользователей, желающих вести поиск с помощью регулярных выражений. У этих нерадивых пользователей образовались огромные каталоги с тысячами файлов, и им хочется получить возможность искать все PDF-файлы, в названии которых где-нибудь имеется сочетание oopsla. Чтобы позволить им сделать это, вы создаете следующую функцию:

```
def filesRegex(query: String) =
  for (file <- filesHere; if file.getName.matches(query))
    yield file
```

Опытные программисты могут обратить внимание на все допущенные повторения и удивиться тому, что они не были сведены в общую вспомогательную функцию. Но если подходить к решению этой задачи в лоб, то ничего не получится. Можно было бы придумать следующее:

```
def filesMatching(query: String, метод) =
  for (file <- filesHere; if file.getName.метод(query))
    yield file
```

В некоторых динамичных языках такой подход сработал бы, но в Scala не разрешается вставлять подобный код во время выполнения. Что же делать?

Ответ дают функциональные значения. Передавать имя метода в качестве значения нельзя, но точно такой же эффект можно получить, если передать функциональное значение, вызывающее для вас этот метод. В этом случае к методу,

единственной задачей которого будет проверка соответствия имени файла запросу, добавляется параметр `matcher`:

```
def filesMatching(query: String,
    matcher: (String, String) => Boolean) = {

  for (file <- filesHere; if matcher(file.getName, query))
    yield file
}
```

В данной версии метода условие `if` теперь использует параметр `matcher` для проверки соответствия имени файла запросу. Что именно проверяется, зависит от того, что указано в качестве `matcher`. А теперь посмотрите на тип самого этого параметра. Это функция, вследствие чего в типе имеется обозначение `=>`. Функция получает два строковых аргумента, имя файла и запрос, и возвращает булево значение, следовательно, типом этой функции является `(String, String) => Boolean`.

Располагая новым вспомогательным методом по имени `filesMatching`, можно упростить три поисковых метода, заставив их вызывать вспомогательный метод, передавая в него соответствующую функцию:

```
def filesEnding(query: String) =
  filesMatching(query, _.endsWith(_))

def filesContaining(query: String) =
  filesMatching(query, _.contains(_))

def filesRegex(query: String) =
  filesMatching(query, _.matches(_))
```

Функциональные литералы, показанные в данном примере, задействуют синтаксис заместителя, рассмотренный в предыдущей главе, который может быть вам еще не совсем привычен. Поэтому поясним, как применяются заместители: функциональный литерал `_.endsWith(_)`, используемый в методе `filesEnding`, означает то же самое, что и следующий код:

```
(fileName: String, query: String) => fileName.endsWith(query)
```

Поскольку `filesMatching` получает функцию, требующую два `String`-аргумента, то типы аргументов указывать не нужно — можно просто воспользоваться кодом `(filename, query) => filename.endsWith(query)`. А так как в теле функции каждый из параметров используется только раз (то есть первый параметр, `filename`, применяется в теле первым, второй параметр, `query`, — вторым), можно прибегнуть к синтаксису заместителей — `_.endsWith(_)`. Первый знак подчеркивания станет заместителем для первого параметра — имени файла, а второй — заместителем для второго параметра — строки запроса.

Этот код уже упрощен, но может стать еще короче. Обратите внимание: аргумент `query` передается `filesMatching`, но данная функция ничего с ним не делает, за исключением того, что передает его назад переданной функции `matcher`. Такая передача туда-сюда необязательна, поскольку вызывающий код с самого начала знает о `query`! Это позволяет удалить параметр `query` как из `filesMatching`, так и из `matcher`, упростив код до состояния, показанного в листинге 9.1.

Листинг 9.1. Использование замыканий для сокращения повторяемости кода

```
object FileMatcher {
  private def filesHere = (new java.io.File(".")).listFiles

  private def filesMatching(matcher: String => Boolean) =
    for (file <- filesHere; if matcher(file.getName))
      yield file

  def filesEnding(query: String) =
    filesMatching(_.endsWith(query))

  def filesContaining(query: String) =
    filesMatching(_.contains(query))

  def filesRegex(query: String) =
    filesMatching(_.matches(query))
}
```

В этом примере показан способ, позволяющий функциям первого класса помочь вам избавиться от дублирующегося кода, что без них сделать было бы очень трудно. В Java, к примеру, можно создать интерфейс, который содержит метод, получающий одно `String`-значение и возвращающий значение типа `Boolean`, а затем создать и передать функции `filesMatching` экземпляры анонимного внутреннего класса, реализующие этот интерфейс. Такой подход позволит избавиться от дублирующегося кода, чего, собственно, вы и добивались, но в то же время приведет к добавлению чуть ли не большего количества нового кода. Стало быть, цена вопроса сведет на нет все преимущества и лучше будет, вероятно, смириться с повторяемостью кода.

Кроме того, данный пример показывает, как замыкания способны помочь сократить повторяемость кода. Функциональные литералы, использованные в предыдущем примере, такие как `_.endsWith(_)` и `_.contains(_)`, во время выполнения программы становятся экземплярами функциональных значений, которые *не* являются замыканиями, поскольку не захватывают никаких свободных переменных. К примеру, обе переменные, использованные в выражении `_.endsWith(_)`, представлены в виде знаков подчеркивания, следовательно, берутся из аргументов функции. Таким образом, в `_.endsWith(_)` задействуются не свободные, а две связанные переменные. В отличие от этого в функциональном литерале `_.endsWith(query)`, использованном в самом последнем примере, содержатся одна связанная переменная, а именно аргумент, представленный знаком подчеркивания, и одна свободная переменная по имени `query`. Возможность убрать параметр `query` из `filesMatching` в этом примере, тем самым еще больше упростив код, появилась у вас только потому, что в Scala поддерживаются замыкания.

9.2. Упрощение клиентского кода

В предыдущем примере было показано, что применение функций высшего порядка способствует сокращению повторяемости кода по мере реализации API. Еще один важный способ использовать функции высшего порядка — поместить их в сам API

с целью повысить лаконичность клиентского кода. Хорошим примером могут послужить методы организации циклов специального назначения, принадлежащие имеющимся в Scala типам коллекций[1]. Многие из них перечислены в табл. 3.1, но сейчас, чтобы понять, почему эти методы настолько полезны, внимательно рассмотрите только один пример.

Рассмотрим `exists` — метод, определяющий факт наличия переданного значения в коллекции. Разумеется, искать элемент можно, инициализировав `var`-переменную значением `false` и выполнив перебор элементов коллекции, проверяя каждый из них и присваивая `var`-переменной значение `true`, если будет найден предмет поиска. Метод, в котором такой подход используется с целью определить, имеется ли в переданном списке `List` отрицательное число, выглядит следующим образом:

```
def containsNeg(nums: List[Int]): Boolean = {
  var exists = false
  for (num <- nums)
    if (num < 0)
      exists = true
  exists
}
```

Если определить этот метод в интерпретаторе, то его можно вызвать следующими командами:

```
scala> containsNeg(List(1, 2, 3, 4))
res0: Boolean = false

scala> containsNeg(List(1, 2, -3, 4))
res1: Boolean = true
```

Но более лаконичный способ определения метода предусматривает вызов в отношении списка `List` функции высшего порядка `exists`:

```
def containsNeg(nums: List[Int]) = nums.exists(_ < 0)
```

Эта версия `containsNeg` выдает те же результаты, что и предыдущая:

```
scala> containsNeg(Nil)
res2: Boolean = false

scala> containsNeg(List(0, -1, -2))
res3: Boolean = true
```

Метод `exists` представляет собой управляющую абстракцию. Это специализированная циклическая конструкция, которая не встроена в язык Scala, как `while` или `for`, а предоставляется библиотекой Scala. В предыдущем разделе функция высшего порядка `filesMatching` позволила сократить повторяемость кода в реализации объекта `FileMatcher`. Метод `exists` обеспечивает такое же преимущество, но

[1] Эти специализированные методы циклической обработки определены в трейте Iterable, который является расширением классов List, Set и Map. Более подробно данный вопрос рассматривается в главе 17.

поскольку это публичный метод в API коллекций Scala, то сокращение повторяемости относится к клиентскому коду этого API. Если бы метода `exists` не было и потребовалось бы написать метод выявления наличия в списке четных чисел `containsOdd`, то это можно было бы сделать так:

```
def containsOdd(nums: List[Int]): Boolean = {
  var exists = false
  for (num <- nums)
    if (num % 2 == 1)
      exists = true
  exists
}
```

Сравнивая тело метода `containsNeg` с телом метода `containsOdd`, можно заметить повторяемость во всем, за исключением условия проверки в выражении `expression`. С помощью метода `exists` вместо этого можно воспользоваться следующим кодом:

```
def containsOdd(nums: List[Int]) = nums.exists(_ % 2 == 1)
```

Тело кода в этой версии также практически идентично телу соответствующего метода `containsNeg` (той его версии, в которой используется `exists`), за исключением того, что условие, по которому выполняется поиск, иное. И тем не менее объем повторяющегося кода значительно уменьшился, поскольку вся инфраструктура организации цикла убрана в метод `exists`.

В стандартной библиотеке Scala имеется множество других методов для организации цикла. Как и `exists`, они зачастую могут сократить объем вашего кода, если появится возможность их применения.

9.3. Карринг

В главе 1 говорилось, что Scala позволяет создавать новые управляющие абстракции, которые воспринимаются как естественная языковая поддержка. Хотя показанные до сих пор примеры фактически и были управляющими абстракциями, вряд ли кто-то смог бы воспринять их как естественную поддержку со стороны языка. Чтобы понять, как создаются управляющие абстракции, больше похожие на расширения языка, сначала нужно разобраться с приемом функционального программирования, который называется *карринг*.

Каррированная функция применяется не к одному, а к нескольким спискам аргументов. В листинге 9.2 показана обычная, некаррированная функция, складывающая два `Int`-параметра, `x` и `y`.

Листинг 9.2. Определение и вызов обычной функции

```
scala> def plainOldSum(x: Int, y: Int) = x + y
plainOldSum: (x: Int, y: Int)Int

scala> plainOldSum(1, 2)
res4: Int = 3
```

В листинге 9.3 показана аналогичная, но уже каррированная функция. Вместо списка из двух параметров типа `Int` эта функция применяется к двум спискам, в каждом из которых содержится по одному параметру типа `Int`.

Листинг 9.3. Определение и вызов каррированной функции

```
scala> def curriedSum(x: Int)(y: Int) = x + y
curriedSum: (x: Int)(y: Int)Int

scala> curriedSum(1)(2)
res5: Int = 3
```

Здесь при вызове `curriedSum` вы фактически получаете два обычных вызова функции, следующие непосредственно друг за другом. Первый получает единственный параметр `Int` по имени `x` и возвращает функциональное значение для второй функции. А та получает `Int`-параметр `y`. Здесь действие функции по имени `first` соответствует тому, что должно было происходить при вызове первой традиционной функции `curriedSum`:

```
scala> def first(x: Int) = (y: Int) => x + y
first: (x: Int)Int => Int
```

Применение первой функции к числу 1, иными словами, вызов первой функции и передача ей значения `1`, образует вторую функцию:

```
scala> val second = first(1)
second: Int => Int = $$Lambda$1044/1220897602@5c6fae3c
```

Применение второй функции к числу 2 дает результат:

```
scala> second(2)
res6: Int = 3
```

Функции `first` и `second` всего лишь показывают процесс карринга. Они не связаны непосредственно с функцией `curriedSum`. И тем не менее это способ получить фактическую ссылку на вторую функцию из `curriedSum`. Чтобы воспользоваться `curriedSum` в выражении частично примененной функцией, можно обратиться к форме записи с заместителем:

```
scala> val onePlus = curriedSum(1)_
onePlus: Int => Int = $$Lambda$1054/711248671@3644d12a
```

Знак подчеркивания в `curriedSum(1)_` является заместителем для второго списка, используемого в качестве параметра[1]. В результате получается ссылка на функцию, при вызове которой единица прибавляется к ее единственному `Int`-аргументу, и возвращается результат:

```
scala> onePlus(2)
res7: Int = 3
```

[1] В предыдущей главе, где форма записи с заместителем использовалась в отношении традиционных методов наподобие println _, между именем и знаком подчеркивания нужно было ставить пробел. Здесь в этом нет необходимости, поскольку в Scala println_ является допустимым идентификатором, а curriedSum(1)_ — нет.

А вот как можно получить функцию, прибавляющую число 2 к ее единственному Int-аргументу:

```
scala> val twoPlus = curriedSum(2)_
twoPlus: Int => Int = $$Lambda$1055/473485349@48b85dc5

scala> twoPlus(2)
res8: Int = 4
```

9.4. Создание новых управляющих конструкций

В языках, использующих функции первого класса, даже если синтаксис языка устоялся, есть возможность эффективно создавать новые управляющие конструкции. Нужно лишь создать методы, получающие функции в виде аргументов.

Например, в фрагменте кода ниже показана удваивающая управляющая конструкция — она повторяет операцию два раза и возвращает результат:

```
scala> def twice(op: Double => Double, x: Double) = op(op(x))
twice: (op: Double => Double, x: Double)Double

scala> twice(_ + 1, 5)
res9: Double = 7.0
```

Типом op в данном примере является Double => Double. Это значит, функция получает одно Double-значение в качестве аргумента и возвращает другое Double-значение.

Каждый раз, замечая шаблон управления, повторяющийся в разных частях вашего кода, вы должны задуматься о его реализации в виде новой управляющей конструкции. Ранее в этой главе был показан filesMatching, узкоспециализированный шаблон управления. Теперь рассмотрим более широко применяющийся шаблон программирования: открытие ресурса, работа с ним, а затем закрытие ресурса. Все это можно собрать в управляющую абстракцию, прибегнув к методу, показанному ниже:

```
def withPrintWriter(file: File, op: PrintWriter => Unit) = {
  val writer = new PrintWriter(file)
  try {
    op(writer)
  } finally {
    writer.close()
  }
}
```

При наличии такого метода им можно воспользоваться так:

```
withPrintWriter(
  new File("date.txt"),
  writer => writer.println(new java.util.Date)
)
```

Преимущества применения этого метода состоят в том, что закрытие файла в конце работы гарантируется withPrintWriter, а не пользовательским кодом.

Поэтому забыть закрыть файл просто невозможно. Данная технология называется *шаблоном временного пользования* (loan pattern), поскольку функция управляющей абстракции, такая как `withPrintWriter`, открывает ресурс и отдает его функции во временное пользование. Так, в предыдущем примере `withPrintWriter` отдает во временное пользование `PrintWriter` функции `op`. Когда функция завершает работу, она сигнализирует, что ей уже не нужен одолженный ресурс. Затем в блоке `finally` ресурс закрывается; это гарантирует его безусловное закрытие независимо от того, как завершилась работа функции — успешно или с генерацией исключения.

Один из способов придать клиентскому коду вид, который делает его похожим на встроенную управляющую конструкцию, предусматривает заключение списка аргументов в фигурные, а не в круглые скобки. Если в Scala при каждом вызове метода ему передается строго один аргумент, то можно заключить его не в круглые, а в фигурные скобки. Например, вместо:

```
scala> println("Hello, world!")
Hello, world!
```

можно написать:

```
scala> println { "Hello, world!" }
Hello, world!
```

Во втором примере аргумент для `println` вместо круглых скобок заключен в фигурные. Но такой прием использования фигурных скобок будет работать только при передаче одного аргумента. Попытка нарушить это правило приводит к следующему результату:

```
scala> val g = "Hello, world!"
g: String = Hello, world!

scala> g.substring { 7, 9 }
                   ^
       error: ';' expected but ',' found.
```

Поскольку была предпринята попытка передать функции `substring` два аргумента, то при их заключении в фигурные скобки выдается ошибка. Вместо фигурных в данном случае нужно использовать круглые скобки:

```
scala> g.substring(7, 9)
res12: String = wo
```

Назначение такой возможности заменить круглые скобки фигурными при передаче одного аргумента — позволить программистам-клиентам записать в фигурных скобках функциональный литерал. Тем самым можно сделать вызов метода похожим на управляющую абстракцию. В качестве примера можно взять определенный ранее метод `withPrintWriter`. В своем самом последнем виде метод `withPrintWriter` получает два аргумента, поэтому использовать фигурные скобки нельзя. Тем не менее, поскольку функция, переданная `withPrintWriter`, является последним аргументом в списке, можно воспользоваться каррингом, чтобы переместить первый аргумент типа `File` в отдельный список аргументов. Тогда функция останется единственным параметром второго списка параметров. Способ переопределения `withPrintWriter` показан в листинге 9.4.

Листинг 9.4. Применение шаблона временного пользования для записи в файл

```
def withPrintWriter(file: File)(op: PrintWriter => Unit) = {
  val writer = new PrintWriter(file)
  try {
    op(writer)
  } finally {
    writer.close()
  }
}
```

Новая версия отличается от старой всего лишь тем, что теперь есть два списка параметров, по одному параметру в каждом, а не один список из двух параметров. Загляните между двумя параметрами. В показанной здесь прежней версии `withPrintWriter` вы видите `...File, op...`. Но в этой версии вы видите `...File)(op...`. Благодаря определению, приведенному ранее, метод можно вызвать с помощью более привлекательного синтаксиса:

```
val file = new File("date.txt")

withPrintWriter(file) { writer =>
  writer.println(new java.util.Date)
}
```

В этом примере первый список аргументов, в котором содержится один аргумент типа `File`, заключен в круглые скобки. А второй список аргументов, содержащий функциональный аргумент, заключен в фигурные скобки.

9.5. Передача параметров по имени

Метод `withPrintWriter`, рассмотренный в предыдущем разделе, отличается от встроенных управляющих конструкций языка, таких как `if` и `while`, тем, что код между фигурными скобками получает аргумент. Функция, переданная `withPrintWriter`, требует одного аргумента типа `PrintWriter`. Этот аргумент показан в следующем коде как `writer =>`:

```
withPrintWriter(file) { writer =>
  writer.println(new java.util.Date)
}
```

А что нужно сделать, если понадобится реализовать нечто больше похожее на `if` или `while`, где между фигурными скобками нет значения для передачи в код? Помочь справиться с подобными ситуациями могут имеющиеся в Scala *параметры, передаваемые по имени* (by-name parameters).

В качестве конкретного примера представим, будто нужно реализовать конструкцию утверждения под названием `myAssert`[1]. Функция `myAssert` будет получать

[1] Здесь используется название myAssert, а не assert, поскольку имя assert предоставляется самим языком Scala. Соответствующее описание будет дано в разделе 14.1.

в качестве ввода функциональное значение и обращаться к флагу, чтобы решить, что делать. Если флаг установлен, то `myAssert` вызовет переданную функцию и проверит, что она возвращает `true`. Если сброшен, то `myAssert` будет молча бездействовать.

Не прибегая к использованию параметров, передаваемых по имени, конструкцию `myAssert` можно создать следующим образом:

```
var assertionsEnabled = true

def myAssert(predicate: () => Boolean) =
  if (assertionsEnabled && !predicate())
    throw new AssertionError
```

С определением все в порядке, но пользоваться им неудобно:

```
myAssert(() => 5 > 3)
```

Конечно, лучше было бы обойтись в функциональном литерале без пустого списка параметров и обозначения `=>` и создать следующий код:

```
myAssert(5 > 3) // Не будет работать из-за отсутствия () =>
```

Именно для воплощения задуманного и существуют параметры, передаваемые по имени. Чтобы создать такой параметр, задавать тип параметра нужно с обозначения `=>`, а не с `() =>`. Например, можно заменить в `myAssert` параметр `predicate` параметром, передаваемым по имени, изменив его тип `() => Boolean` на `=> Boolean`. Как это должно выглядеть, показано в листинге 9.5.

Листинг 9.5. Использование параметра, передаваемого по имени

```
def byNameAssert(predicate: => Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

Теперь в свойстве, по поводу которого нужно высказать утверждение, можно избавиться от пустого параметра. В результате этого использование `byNameAssert` выглядит абсолютно так же, как встроенная управляющая конструкция:

```
byNameAssert(5 > 3)
```

Тип «по имени» (by-name), в котором отбрасывается пустой список параметров `()`, допустимо использовать только в отношении параметров. Никаких by-name-переменных или by-name-полей не существует.

Можно, конечно, удивиться, почему нельзя просто написать функцию `myAssert`, воспользовавшись для ее параметров старым добрым типом `Boolean` и создав следующий код:

```
def boolAssert(predicate: Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

Разумеется, такая формулировка тоже будет работать, и код, использующий эту версию `boolAssert`, будет выглядеть точно так же, как и прежде:

```
boolAssert(5 > 3)
```

И все же эти два подхода различаются весьма значительным образом. Для параметра `boolAssert` используется тип `Boolean`, и потому выражение внутри круглых скобок в `boolAssert(5 > 3)` вычисляется *до* вызова `boolAssert`. Выражение `5 > 3` выдает значение `true`, которое передается в `boolAssert`. В отличие от этого, поскольку типом параметра `predicate` функции `byNameAssert` является `=> Boolean`, выражение внутри круглых скобок в `byNameAssert(5 > 3)` до вызова `byNameAssert` не вычисляется. Вместо этого будет создано функциональное значение, чей метод `apply` станет вычислять `5 > 3`, и это функциональное значение будет передано функции `byNameAssert`.

Таким образом, разница между двумя подходами состоит в том, что при отключении утверждений вам будут видны любые побочные эффекты, которые могут быть в выражении внутри круглых скобок в `boolAssert`, но `byNameAssert` это не касается. Например, если утверждения отключены, то попытки утверждать, что `x / 0 == 0`, в случае использования `boolAssert` приведут к генерации исключения:

```
scala> val x = 5
x: Int = 5

scala> assertionsEnabled = false
mutated assertionsEnabled

scala> boolAssert(x / 0 == 0)
java.lang.ArithmeticException: / by zero
  ... 27 elided
```

Но попытки утверждать это на основе того же самого кода в случае использования `byNameAssert` *не* приведут к генерации исключения:

```
scala> byNameAssert(x / 0 == 0)
```

Резюме

В этой главе мы показали, как с помощью богатого инструментария функций Scala строить управляющие абстракции. Функции внутри вашего кода можно применять для избавления от распространенных шаблонов управления, а чтобы повторно задействовать шаблоны управления, часто встречающиеся в вашем программном коде, можно прибегнуть к функциям высшего порядка из библиотеки Scala. Кроме того, мы обсудили приемы использования каррининга и параметров, передаваемых по имени, которые позволяют применять в весьма лаконичном синтаксисе собственные функции высшего порядка.

В двух последних главах мы рассмотрели довольно много всего, что относится к функциям. В следующих нескольких главах вернемся к рассмотрению дополнительных объектно-ориентированных средств языка.

10 Композиция и наследование

В главе 6 мы представили часть основных объектно-ориентированных аспектов Scala. В текущей продолжим рассматривать эту тему с того места, где остановились в главе 6, и дадим углубленное и гораздо более полное представление об имеющейся в Scala поддержке объектно-ориентированного программирования.

Нам предстоит сравнить два основных вида взаимоотношений между классами: композицию и наследование. Композиция означает, что один класс содержит ссылку на другой и использует класс, на который ссылается, в качестве вспомогательного средства для выполнения своей миссии. Наследование — это отношения «суперкласс/подкласс» (родительский/дочерний класс).

Помимо этих тем, мы рассмотрим абстрактные классы, методы без параметров, расширение классов, переопределение методов и полей, параметрические поля, вызов конструкторов суперкласса, полиморфизм и динамическое связывание, финальные члены и классы, а также фабричные объекты и методы.

10.1. Библиотека двумерной разметки

В качестве рабочего примера в этой главе мы создадим библиотеку для построения и вывода на экран двумерных элементов разметки. Каждый элемент будет представлен прямоугольником, заполненным текстом. Для удобства библиотека будет предоставлять фабричные методы по имени `elem`, который создает новые элементы из переданных данных. Например, вы сможете создать элемент разметки, содержащий строку, используя фабричный метод со следующей сигнатурой:

```
elem(s: String): Element
```

Как видите, элементы будут моделироваться с помощью типа данных по имени `Element`. Чтобы получить новый элемент, объединяющий два элемента, вы можете вызвать в отношении элемента операторы `above` или `beside`, передавая им второй

элемент. Например, выражение, показанное ниже, создаст более крупный элемент, содержащий два столбца, каждый высотой в два элемента:

```
val column1 = elem("hello") above elem("***")
val column2 = elem("***") above elem("world")
column1 beside column2
```

Вывод результата этого выражения даст следующий результат:

```
hello ***
*** world
```

Элементы разметки — хороший пример системы, в которой объекты могут создаваться из простых частей с помощью операторов композиции. В данной главе будут определены классы, позволяющие создавать объекты элементов из массивов, рядов и прямоугольников. Эти объекты базовых элементов будут простыми деталями. Вдобавок будут определены операторы композиции `above` и `beside`. Такие операторы зачастую называют *комбинаторами*, поскольку они комбинируют элементы некой области в новые элементы.

Подходить к проектированию библиотеки лучше всего в понятиях комбинаторов: они позволяют осмыслить основные способы конструирования объектов в прикладной области. Что представляют собой простые объекты? Какими способами из простых объектов могут создаваться более интересные объекты? Как комбинаторы должны сочетаться друг с другом? Что должны представлять собой наиболее общие комбинации? Удовлетворяют ли они всем выдвигаемым правилам? Если у вас есть хорошие ответы на все эти вопросы, то вы на правильном пути.

10.2. Абстрактные классы

Нашей первой задачей будет определить тип `Element`, представляющий элементы разметки. Элементы — двумерные прямоугольники из символов, поэтому имеет смысл включить в класс метод по имени `contents`, ссылающийся на содержимое элемента разметки. Данный метод может быть представлен в виде массива строк, где каждая строка обозначает ряд. Отсюда типом возвращаемого `contents` значения должен быть `Array[String]`. Как он будет выглядеть, показано в листинге 10.1.

Листинг 10.1. Определение абстрактного метода и класса

```
abstract class Element {
  def contents: Array[String]
}
```

В этом классе `contents` объявляется в качестве метода, у которого нет реализации. Иными словами, метод является *абстрактным* членом класса `Element`. Класс с абстрактными членами сам по себе должен быть объявлен абстрактным; это можно сделать, указав модификатор `abstract` перед ключевым словом `class`:

```
abstract class Element ...
```

Модификатор `abstract` указывает на то, что у класса могут быть абстрактные члены, не имеющие реализации. Поэтому создать экземпляр абстрактного класса невозможно. При попытке сделать это будет получена ошибка компиляции:

```
scala> new Element
<console>:5: error: class Element is abstract;
  cannot be instantiated
    new Element
        ^
```

Чуть позже в этой главе будет показан способ создания подклассов класса `Element`, экземпляры которых можно будет создавать, поскольку они заполняют отсутствующее определение метода `contents`.

Обратите внимание на то, что у метода `contents` класса `Element` нет модификатора `abstract`. Метод абстрактный, если у него нет реализации (то есть знака равенства и тела). В отличие от Java здесь при объявлении методов не нужны (не разрешены) модификаторы `abstract`. Методы, имеющие реализацию, называются *конкретными*.

И еще одно различие в терминологии между *объявлениями* и *определениями*. Класс `Element` *объявляет* абстрактный метод `contents`, но пока *не определяет* никаких конкретных методов. Но в следующем разделе класс `Element` будет усилен определением нескольких конкретных методов.

10.3. Определяем методы без параметров

В качестве следующего шага к `Element` будут добавлены методы, показывающие его ширину (`width`) и высоту (`height`) (листинг 10.2). Метод `height` возвращает количество рядов в содержимом. Метод `width` возвращает длину первого ряда или, при отсутсвии рядов в элементе, нуль. Это значит, нельзя определить элемент с нулевой высотой и ненулевой шириной.

Листинг 10.2. Определение не имеющих параметров методов width и height

```
abstract class Element {
  def contents: Array[String]
  def height: Int = contents.length
  def width: Int = if (height == 0) 0 else contents(0).length
}
```

Обратите внимание: ни в одном из трех методов класса `Element` нет списка параметров, даже пустого. Например, вместо:

```
def width(): Int
```

метод определен без круглых скобок:

```
def width: Int
```

Такие *методы без параметров* встречаются в Scala довольно часто. В отличие от них методы, определенные с пустыми круглыми скобками, например

`def height(): Int`, называются *методами с пустыми скобками*. Согласно имеющимся рекомендациям методы без параметров следует использовать, когда параметры отсутствуют *и* метод обращается к изменяемому состоянию только для чтения полей содержащего его объекта (притом он не модифицирует изменяемое состояние). По этому соглашению поддерживается *принцип единообразного доступа*[1], который гласит, что на клиентский код не должен влиять способ реализации атрибута в виде поля или метода.

Например, можно реализовать `width` и `height` в виде полей, а не методов, просто заменив `def` в каждом определении на `val`:

```
abstract class Element {
  def contents: Array[String]
  val height = contents.length
  val width =
    if (height == 0) 0 else contents(0).length
}
```

С точки зрения клиента, две пары определений абсолютно эквивалентны. Единственное различие заключается в том, что доступ к полю может осуществляться немного быстрее вызова метода, поскольку значения полей предварительно вычислены при инициализации класса, а не вычисляются при каждом вызове метода. В то же время полям в каждом объекте `Element` требуется дополнительное пространство памяти. Поэтому вопрос о том, как лучше представить атрибут, в виде поля или в виде метода, зависит от способа использования класса клиентами, который со временем может измениться. Главное, чтобы на клиенты класса `Element` никак не воздействовали изменения, вносимые во внутреннюю реализацию класса.

В частности, клиент класса `Element` не должен испытывать необходимости в перезаписи кода, если поле данного класса было переделано в функцию доступа, при условии, что это *чистая* функция доступа (то есть не имеет никаких побочных эффектов и не зависит от изменяемого состояния). Как бы то ни было, клиент не должен решать какие-либо проблемы.

Пока у нас все получается. Но есть небольшое осложнение, связанное с методами работы Java. Дело в том, что в данном языке не реализуется принцип единообразного доступа. Поэтому `string.length()` в Java — это не то же самое, что `string.length`, и даже `array.length` — не то же самое, что `array.length()`. Вряд ли стоит говорить, насколько это усложняет дело.

Преодолеть это препятствие языку Scala помогает то, что он весьма мягко относится к смешиванию методов без параметров и методов с пустыми круглыми скобками. В частности, можно заменить метод без параметров методом с пустыми круглыми скобками и *наоборот*. Можно также не ставить пустые круглые скобки при вызове любой функции, не получающей аргументов. Например, в Scala одинаково допустимо применение двух следующих строк кода:

```
Array(1, 2, 3).toString
"abc".length
```

[1] *Meyer B.* Object-Oriented Software Construction. — Prentice Hall, 2000.

В принципе, в вызовах функций в Scala можно вообще не ставить пустые круглые скобки. Но их все же рекомендуется использовать, когда вызываемый метод представляет нечто большее, чем свойство своего объекта-получателя. Например, пустые круглые скобки уместны, если метод выполняет ввод-вывод, записывает переназначаемые переменные (var-переменные) или считывает var-переменные, не являющиеся полями объекта-получателя, как непосредственно, так и косвенно, используя изменяемые объекты. Таким образом, список параметров служит визуальным признаком того, что вызов инициирует некие примечательные вычисления, например:

```
"hello".length   // Нет (), поскольку побочные эффекты отсутствуют
println()        // Лучше () не отбрасывать
```

Подводя итоги, следует отметить, что в Scala приветствуется определение методов, не получающих параметров и не имеющих побочных эффектов, в виде методов без параметров, то есть без пустых круглых скобок. В то же время никогда не нужно определять метод, имеющий побочные эффекты, без круглых скобок, поскольку вызов этого метода будет выглядеть как выбор поля. Следовательно, ваши клиенты могут быть удивлены, столкнувшись с побочными эффектами.

По аналогии с этим при вызове функции, имеющей побочные эффекты, не забудьте при написании вызова поставить пустые круглые скобки. Есть еще один способ представить данное правило: если вызываемая функция выполняет операцию, то используйте круглые скобки. Но если она просто предоставляет доступ к свойству, то круглые скобки следует отбросить.

10.4. Расширяем классы

Нам по-прежнему нужна возможность создавать объекты-элементы. Вы уже видели, что новый класс Element не приспособлен для этого в силу своей абстрактности. Поэтому для получения экземпляра элемента необходимо создать подкласс, расширяющий класс Element и реализующий абстрактный метод contents. Как это делается, показано в листинге 10.3.

Листинг 10.3. Определение класса ArrayElement в качестве подкласса класса Element

```
class ArrayElement(conts: Array[String]) extends Element {
  def contents: Array[String] = conts
}
```

Класс ArrayElement определен в целях *расширения* класса Element. Как и в Java, для выражения данного обстоятельства после имени класса указывается уточнение extends:

```
... extends Element ...
```

Использование уточнения extends заставляет класс ArrayElement *унаследовать* у класса Element все его неприватные элементы и превращает тип ArrayElement

в *подтип* типа `Element`. Поскольку `ArrayElement` расширяет `Element`, то класс `ArrayElement` называется *подклассом* класса `Element`. В свою очередь, `Element` — *суперкласс* `ArrayElement`. Если не указать уточнение `extends`, то компилятор Scala, безусловно, предположит, что ваш класс является расширением класса `scala.AnyRef`, который на платформе Java будет соответствовать классу `java.lang.Object`. Получается, класс `Element` неявно расширяет класс `AnyRef`. Эти отношения наследования показаны на рис. 10.1.

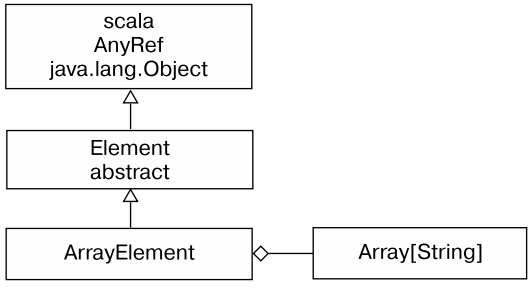


Рис. 10.1. Схема классов для ArrayElement

Наследование означает, что все элементы суперкласса являются также элементами подкласса, но с двумя исключениями. Первое: приватные элементы суперкласса не наследуются подклассом. Второе: элемент суперкласса не наследуется, если элемент с такими же именем и параметрами уже реализован в подклассе. В таком случае говорится, что элемент подкласса *переопределяет* элемент суперкласса. Если элемент в подклассе является конкретным, а в суперклассе — абстрактным, также говорится, что конкретный элемент — это *реализация* абстрактного элемента.

Например, метод `contents` в `ArrayElement` переопределяет (или, в ином толковании, реализует) абстрактный метод `contents` класса `Element`[1]. В отличие от этого класс `ArrayElement` наследует у класса `Element` методы `width` и `height`. Например, располагая `ArrayElement`-объектом `ae`, можно запросить его ширину, используя выражение `ae.width`, как будто метод `width` был определен в классе `ArrayElement`:

```
scala> val ae = new ArrayElement(Array("hello", "world"))
ae: ArrayElement = ArrayElement@5ae66c98

scala> ae.width
res0: Int = 5
```

[1] Один из недостатков данной конструкции заключается в том, что из-за изменяемости возвращаемого массива клиенты могут его модифицировать. В книге мы старались ничего не усложнять, но будь ArrayElement частью реального проекта, можно было бы рассмотреть возвращение вместо этого копии массива, которая позволяет защититься от изменений. Еще одна проблема заключается в том, что мы пока не гарантируем одинаковой длины массива contents каждого String-элемента. Задачу можно решить, проверяя соблюдение предварительного условия в первичном конструкторе и генерируя исключения при его нарушении.

Создание подтипов означает, что значение подкласса может быть использовано там, где требуется значение суперкласса. Например:

```
val e: Element = new ArrayElement(Array("hello"))
```

Переменная `e` определена как принадлежащая типу `Element`, следовательно, значение, используемое для ее инициализации, также должно быть типа `Element`. А фактически типом этого значения является класс `ArrayElement`. Это нормально, поскольку он расширяет класс `Element` и в результате тип `ArrayElement` совместим с типом `Element`[1].

На рис. 10.1 также показано отношение *композиции* между `ArrayElement` и `Array[String]`. Оно так называется, поскольку класс `ArrayElement` состоит из `Array[String]`, то есть компилятор Scala помещает в генерируемый им для `ArrayElement` двоичный класс поле, содержащее ссылку на переданный массив `conts`.

Некоторые моменты, касающиеся композиции и наследования, будут рассмотрены чуть позже — в разделе 10.11.

10.5. Переопределяем методы и поля

Принцип единообразного доступа является одним из тех аспектов, где Scala подходит к полям и методам единообразно — не так, как Java. Еще одно отличие заключается в том, что в Scala поля и методы принадлежат одному и тому же пространству имен. Этот позволяет полю переопределить метод без параметров. Например, можно, как показано в листинге 10.4, изменить реализацию `contents` в классе `ArrayElement` из метода в поле, не модифицируя определение абстрактного метода `contents` в классе `Element`.

Листинг 10.4. Переопределение метода без параметров в поле

```
class ArrayElement(conts: Array[String]) extends Element {
  val contents: Array[String] = conts
}
```

Поле `contents` (определенное с ключевым словом `val`) в этой версии `ArrayElement` — вполне подходящая реализация метода без параметров `contents` (объявленное с ключевым словом `def`) в классе `Element`. В то же время в Scala запрещено в одном и том же классе определять поле и метод с одинаковыми именами, а в Java это разрешено.

Например, этот класс в Java пройдет компиляцию вполне успешно:

```
// Это код Java
class CompilesFine {
  private int f = 0;
```

[1] Чтобы получить более четкое представление о разнице между подклассом и подтипом, обратитесь к статье глоссария о подтипах.

```
  public int f() {
    return 1;
  }
}
```

Но соответствующий класс в Scala не скомпилируется:

```
class WontCompile {
  private var f = 0 // Не пройдет компиляцию, поскольку поле
  def f = 1         // и метод имеют одинаковые имена
}
```

В принципе, в Scala вместо четырех имеющихся в Java пространств имен для определений есть только два пространства. Четыре пространства имен в Java — это поля, методы, типы и пакеты. Напротив, двумя пространствами имен в Scala являются:

- значения (поля, методы, пакеты и объекты-одиночки);
- типы (имена классов и трейтов).

Причина, по которой поля и методы в Scala помещаются в одно и то же пространство имен, заключается в предоставлении возможности переопределить методы без параметров в `val`-поля, чего нельзя сделать в Java[1].

10.6. Определяем параметрические поля

Рассмотрим еще раз определение класса `ArrayElement`, показанное в предыдущем разделе. В нем имеется параметр `conts`, единственное предназначение которого — его копирование в поле `contents`. Имя `conts` было выбрано для параметра, чтобы походило на имя поля `contents`, но не вступало с ним в конфликт имен. Это «код с душком» — признак того, что в вашем коде может быть некая совершенно ненужная избыточность и повторяемость.

От этого кода сомнительного качества можно избавиться, скомбинировав параметр и поле в едином определении *параметрического поля*, что и показано в листинге 10.5.

Листинг 10.5. Определение contents в качестве параметрического поля

```
class ArrayElement(
  val contents: Array[String]
) extends Element
```

Обратите внимание: параметр `contents` имеет префикс `val`. Это сокращенная форма записи, определяющая одновременно параметр и поле с одним и тем же

[1] Причиной того, что пакеты в Scala используют общее с полями и методами пространство имен, является стремление предоставить вам возможность получать доступ к импорту пакетов (а не только к именам типов), а также к полям и методам объектов-одиночек. Это тоже входит в перечень того, что невозможно сделать в Java. Подробности будут рассмотрены в разделе 13.3.

именем. Если выразиться более конкретно, то класс `ArrayElement` теперь имеет поле `contents` (непереназначаемое), доступ к которому может быть получен за пределами класса. Поле инициализировано значением параметра. Похоже, будто бы класс был написан следующим образом:

```
class ArrayElement(x123: Array[String]) extends Element {
  val contents: Array[String] = x123
}
```

где `x123` — произвольное свежее имя для параметра.

Кроме того, можно поставить перед параметром класса префикс `var`, и тогда соответствующее поле станет переназначаемым. И наконец, подобным параметризованным полям, как и любым другим членам класса, можно добавлять такие модификаторы, как `private`, `protected`[1] или `override`. Рассмотрим, к примеру, следующие определения классов:

```
class Cat {
  val dangerous = false
}
class Tiger(
  override val dangerous: Boolean,
  private var age: Int
) extends Cat
```

Определение класса `Tiger` — сокращенная форма для следующего альтернативного определения класса с переопределяемым элементом `dangerous` и приватным элементом `age`:

```
class Tiger(param1: Boolean, param2: Int) extends Cat {
  override val dangerous = param1
  private var age = param2
}
```

Оба элемента инициализируются соответствующими параметрами. Имена для этих параметров, `param1` и `param2`, были выбраны произвольно. Главное, чтобы они не конфликтовали с какими-либо другими именами в пространстве имен.

10.7. Вызываем конструктор суперкласса

Теперь вы располагаете полноценной системой из двух классов: абстрактного класса `Element`, который расширяется конкретным классом `ArrayElement`. Можно также наметить иные способы выражения элемента. Например, клиенту может понадобиться создать элемент разметки, содержащий один ряд, задаваемый строкой. Объектно-ориентированное программирование упрощает расширение системы новыми вариантами данных. Можно просто добавить подклассы. Например, в листинге 10.6 показан класс `LineElement`, расширяющий класс `ArrayElement`.

[1] Модификатор protected, предоставляющий доступ к подклассам, будет подробно рассмотрен в главе 13.

Листинг 10.6. Вызов конструктора суперкласса

```
class LineElement(s: String) extends ArrayElement(Array(s)) {
  override def width = s.length
  override def height = 1
}
```

Поскольку `LineElement` расширяет `ArrayElement`, а конструктор `ArrayElement` получает параметр (`Array[String]`), то `LineElement` нужно передать аргумент первичному конструктору своего суперкласса. В целях вызова конструктора суперкласса аргумент или аргументы, которые нужно передать, просто помещаются в круглые скобки, стоящие за именем суперкласса. Например, класс `LineElement` передает аргумент `Array(s)` первичному конструктору класса `ArrayElement`, поместив его в круглые скобки и указав после имени суперкласса `ArrayElement`:

```
... extends ArrayElement(Array(s)) ...
```

С появлением нового подкласса иерархия наследования для элементов разметки приобретает вид, показанный на рис. 10.2.

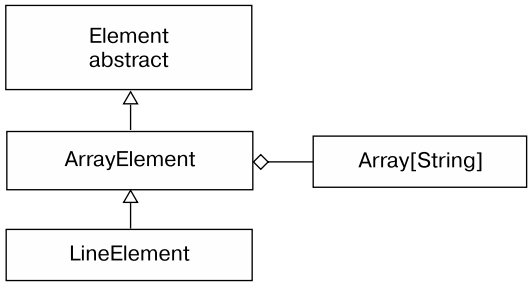


Рис. 10.2. Схема классов для LineElement

10.8. Используем модификатор override

Обратите внимание: определения `width` и `height` в `LineElement` имеют модификатор `override`. В разделе 6.3 он встречался в определении метода `toString`. В Scala такой модификатор требуется для всех элементов, переопределяющих конкретный член родительского класса. Если элемент является реализацией абстрактного элемента с тем же именем, то модификатор указывать не обязательно. Применять модификатор запрещено, если член не переопределяется или не является реализацией какого-либо другого члена базового класса. Поскольку `height` и `width` в классе `LineElement` переопределяют конкретные определения в классе `Element`, то модификатор `override` указывать обязательно.

Соблюдение этого правила дает полезную информацию для компилятора, которая помогает избежать некоторых трудноотлавливаемых ошибок и сделать развитие системы более безопасным. Например, если вы опечатались в названии

метода или случайно указали для него не тот список параметров, то компилятор тут же отреагирует, выдав сообщение об ошибке:

```
$ scalac LineElement.scala
.../LineElement.scala:50:
error: method hight overrides nothing
  override def hight = 1
               ^
```

Соглашение о применении модификатора `override` приобретает еще большее значение, когда дело доходит до развития системы. Скажем, вы определили библиотеку методов рисования двумерных фигур, сделали ее общедоступной, и она стала использоваться довольно широко. Посмотрев на эту библиотеку позже, вы захотели добавить к основному классу `Shape` новый метод с такой сигнатурой:

```
def hidden(): Boolean
```

Новый метод будут использовать различные методы рисования, чтобы определить необходимость отрисовки той или иной фигуры. Тем самым можно существенно ускорить работу, но сделать это, не рискуя вывести из строя клиентский код, невозможно. Кроме всего прочего, у клиента может быть определен подкласс класса `Shape` с другой реализацией `hidden`. Возможно, клиентский метод фактически заставит объект-получатель просто исчезнуть вместо того, чтобы проверять, не является ли он невидимым. Две версии `hidden` переопределяют друг друга, поэтому ваши методы рисования в итоге заставят объекты исчезать, что совершенно не соответствует задуманному!

Эти «случайные переопределения» — наиболее распространенное проявление проблемы так называемого хрупкого базового класса. Проблема в том, что, если вы добавите новые члены в базовые классы (которые мы обычно называем суперклассами), вы рискуете вывести из строя клиентский код. Полностью разрешить проблему хрупкого базового класса в Scala невозможно, но по сравнению с Java ситуация несколько лучше[1]. Если библиотека рисования и ее клиентский код созданы в Scala, то у клиентской исходной реализации `hidden` не мог использоваться модификатор `override`, поскольку на момент его применения не могло быть другого метода с таким же именем.

Когда вы добавите метод `hidden` во вторую версию вашего класса фигур, при повторной компиляции кода клиента будет выдана следующая ошибка:

```
.../Shapes.scala:6: error: error overriding method
    hidden in class Shape of type ()Boolean;
method hidden needs 'override' modifier
def hidden(): Boolean =
^
```

То есть вместо неверного поведения ваш клиент получит ошибку в ходе компиляции, и такой исход обычно более предпочтителен.

---

[1] В Java 1.5 была введена аннотация @Override, работающая аналогично имеющемуся в Scala модификатору override, но, в отличие от Scala-модификатора override, применять ее не обязательно.

10.9. Полиморфизм и динамическое связывание

В разделе 10.4 было показано, что переменная типа `Element` может ссылаться на объект типа `ArrayElement`. Этот феномен называется *полиморфизмом*, что означает «множество форм». В данном случае объекты типа `Element` могут иметь множество форм[1].

Ранее вам уже попадались две такие формы: `ArrayElement` и `LineElement`. Можно создать еще больше форм `Element`, определив новые подклассы класса `Element`. Например, можно определить новую форму `Element` с заданными шириной и высотой (`width` и `height`) и полностью заполненную заданным символом:

```
class UniformElement(
  ch: Char,
  override val width: Int,
  override val height: Int
) extends Element {
  private val line = ch.toString * width
  def contents = Array.fill(height)(line)
}
```

Иерархия наследования для класса `Element` теперь приобретает вид, показанный на рис. 10.3. В результате Scala примет все следующие присваивания, поскольку тип выражения присваивания соответствует типу определяемой переменной:

```
val e1: Element = new ArrayElement(Array("hello", "world"))
val ae: ArrayElement = new LineElement("hello")
val e2: Element = ae
val e3: Element = new UniformElement('x', 2, 3)
```

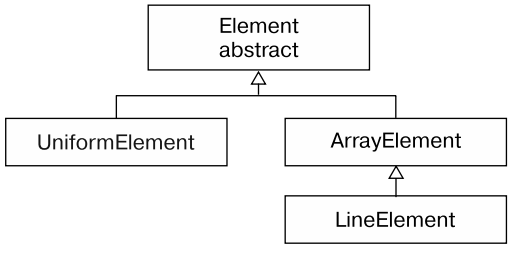


Рис. 10.3. Иерархия классов элементов разметки

Если изучить иерархию наследования, то окажется, что в каждом из этих четырех `val`-определений тип выражения справа от знака равенства принадлежит типу `val`-переменной, инициализируемой слева от знака равенства.

Есть еще одна сторона вопроса: вызовы методов с переменными и выражениями *динамически связаны*. Это значит, текущая реализация вызываемого метода

[1] Эта разновидность полиморфизма называется полиморфизмом подтипов. Еще одна разновидность полиморфизма в Scala, которая называется универсальным полиморфизмом, рассматривается в главе 19.

определяется во время выполнения программы на основе класса объекта, а не типа переменной или выражения. Чтобы продемонстрировать данное поведение, мы временно уберем все существующие члены из наших классов `Element` и добавим к `Element` метод по имени `demo`. Затем переопределим `demo` в `ArrayElement` и `LineElement`, но не в `UniformElement`:

```
abstract class Element {
  def demo() = {
    println("Вызвана реализация, определенная в Element")
  }
}

class ArrayElement extends Element {
  override def demo() = {
    println("Вызвана реализация, определенная в ArrayElement")
  }
}

class LineElement extends ArrayElement {
  override def demo() = {
    println("Вызвана реализация, определенная в LineElement")
  }
}

// UniformElement наследует demo из Element
class UniformElement extends Element
```

Если ввести данный код в интерпретатор, то можно будет определить метод, который получает объект типа `Element` и вызывает в отношении него метод `demo`:

```
def invokeDemo(e: Element) = {
  e.demo()
}
```

Если передать методу `invokeDemo` объект типа `ArrayElement`, то будет показано сообщение, свидетельствующее о вызове реализации `demo` из класса `ArrayElement`, даже если типом переменной `e`, в отношении которой был вызван метод `demo`, являлся `Element`:

```
scala> invokeDemo(new ArrayElement)
Вызвана реализация, определенная в ArrayElement
```

Аналогично, если передать `invokeDemo` объект типа `LineElement`, будет показано сообщение, свидетельствующее о вызове той реализации `demo`, которая была определена в классе `LineElement`:

```
scala> invokeDemo(new LineElement)
Вызвана реализация, определенная в LineElement
```

Поведение при передаче объекта типа `UniformElement` может на первый взгляд показаться неожиданным, но оно вполне корректно:

```
scala> invokeDemo(new UniformElement)
Вызвана реализация, определенная в Element
```

В классе `UniformElement` метод `demo` не переопределяется, поэтому в нем наследуется реализация `demo` из его суперкласса `Element`. Таким образом, реализация, определенная в классе `Element`, — это правильная реализация вызова `demo`, когда классом объекта является `UniformElement`.

10.10. Объявляем финальные элементы

Иногда при проектировании иерархии наследования нужно обеспечить невозможность переопределения элемента подклассом. В Scala, как и в Java, это делается путем добавления к элементу модификатора `final`. Как показано в листинге 10.7, модификатор `final` можно указать для метода `demo` класса `ArrayElement`.

Листинг 10.7. Объявление финального метода

```
class ArrayElement extends Element {
  final override def demo() = {
    println("Вызвана реализация, определенная в ArrayElement")
  }
}
```

При наличии данной версии в классе `ArrayElement` попытка переопределить `demo` в его подклассе `LineElement` не пройдет компиляцию:

```
elem.scala:18: error: error overriding method demo
    in class ArrayElement of type ()Unit;
method demo cannot override final member
    override def demo() = {
                 ^
```

Порой вы должны быть уверены, что создать подкласс для класса в целом невозможно. Для этого нужно просто сделать весь класс финальным, добавив к объявлению класса модификатор `final`. Например, в листинге 10.8 показано, как должен быть объявлен финальный класс `ArrayElement`.

Листинг 10.8. Объявление финального класса

```
final class ArrayElement extends Element {
  override def demo() = {
    println("Вызвана реализация, определенная в ArrayElement")
  }
}
```

При наличии данной версии класса `ArrayElement` любая попытка определить подкласс не пройдет компиляцию:

```
elem.scala: 18: error: illegal inheritance from final class
    ArrayElement
  class LineElement extends ArrayElement {
                            ^
```

Теперь мы удалим модификаторы `final` и методы `demo` и вернемся к прежней реализации семейства классов `Element`. Далее в главе мы сконцентрируемся на завершении создания работоспособной версии библиотеки разметки.

10.11. Используем композицию и наследование

Композиция и наследование — два способа определить новый класс в понятиях другого уже существующего класса. Если вы ориентируетесь преимущественно на повторное использование кода, то, как правило, предпочтение нужно отдавать композиции, а не наследованию. Только ему свойственна проблема хрупкого базового класса, вследствие которой можно ненароком сделать неработоспособными подклассы, внося изменения в суперкласс.

Насчет отношения наследования нужно задаться лишь одним вопросом: не моделируется ли им взаимоотношение типа `is-a` (является)[1]. Например, нетрудно будет заметить, что класс `ArrayElement` *является* разновидностью `Element`. Можно задаться еще одним вопросом: придется ли клиентам использовать тип подкласса в качестве типа суперкласса[2]. Применительно к классу `ArrayElement` не вызывает никаких сомнений, что клиентам потребуется задействовать объекты типа `ArrayElement` в качестве объектов типа `Element`.

Если задаться этими вопросами относительно отношений наследования, показанных на рис. 10.3, то не покажутся ли вам какие-либо из них подозрительными? В частности, насколько для вас очевидно, что `LineElement` *является* (`is-a`) `ArrayElement`? Как вы думаете, понадобится ли когда-нибудь клиентам воспользоваться типом `LineElement` в качестве типа `ArrayElement`?

Фактически класс `LineElement` был определен как подкласс класса `ArrayElement` преимущественно для повторного использования имеющегося в `ArrayElement` определения `contents`. Поэтому, возможно, будет лучше определить `LineElement` в качестве прямого подкласса класса `Element`:

```
class LineElement(s: String) extends Element {
  val contents = Array(s)
  override def width = s.length
  override def height = 1
}
```

В предыдущей версии `LineElement` состоял в отношении наследования с `ArrayElement`, откуда наследовал метод `contents`. Теперь же у него отношение композиции с классом `Array`: в нем содержится ссылка на строковый массив из его собственного поля `contents`[3]. При наличии этой реализации класса `LineElement` иерархия наследования для `Element` приобретет вид, показанный на рис. 10.4.

---

[1] *Meyers S.* Effective C++. — Addison-Wesley, 1991.

[2] *Эккель Б.* Философия Java. — СПб.: Питер, 2021.

[3] Класс ArrayElement также имеет отношение композиции с классом Array, поскольку его параметрическое поле contents содержит ссылку на строковый массив. Код для ArrayElement показан в листинге 10.5 (см. выше). Его отношение композиции представлено в схеме классов в виде ромба, к примеру, на рис. 10.1 (см. выше).

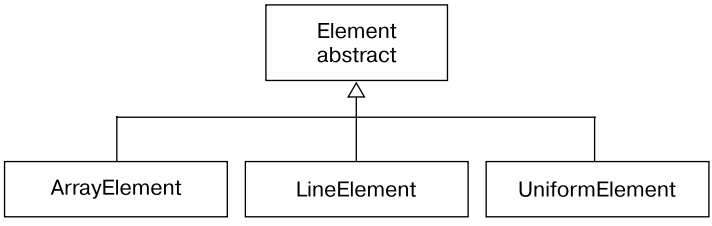


Рис. 10.4. Иерархия класса с пересмотренным определением подкласса LineElement

10.12. Реализуем методы above, beside и toString

В качестве следующего шага в классе `Element` будет реализован метод `above`. Поместить один элемент выше другого с помощью метода `above` означает объединить два значения содержимого элементов, представленного `contents`. Поэтому первый, черновой вариант метода `above` может иметь следующий вид:

```
def above(that: Element): Element =
  new ArrayElement(this.contents ++ that.contents)
```

Операция `++` объединяет два массива. Массивы в Scala представлены в виде массивов Java, но поддерживают значительно большее количество методов. В частности, массивы в Scala могут быть преобразованы в экземпляры класса `scala.Seq`, которые представляют собой последовательные структуры и содержат несколько методов для доступа к последовательностям и их преобразования. В этой главе мы рассмотрим и некоторые другие методы, а развернутое представление о них дадим в главе 17.

В действительности показанный ранее код нельзя считать достаточным, поскольку он не позволяет помещать друг на друга элементы разной ширины. Но чтобы в этом разделе ничего не усложнять, оставим все как есть и станем передавать в метод `above` только элементы одинаковой длины. В разделе 10.14 мы усовершенствуем его, чтобы клиенты могли с его помощью объединять элементы разной ширины.

Следующим будет реализован метод `beside`. Чтобы поставить элементы рядом друг с другом, создадим новый элемент, в котором каждый ряд будет получен путем объединения соответствующих рядов двух элементов. Как и прежде, во избежание усложнений начнем с предположения, что высота двух элементов одинакова. Тогда стуктура метода `beside` приобретет такой вид:

```
def beside(that: Element): Element = {
  val contents = new Array[String](this.contents.length)
  for (I <- 0 until this.contents.length)
    contents(i) = this.contents(i) + that.contents(i)
  new ArrayElement(contents)
}
```

Метод `beside` сначала выделяет новый массив `contents` и заполняет его объединением соответствующих элементов массивов `this.contents` и `that.contents`. В итоге получается новый `ArrayElement`, имеющий новое содержимое `contents`.

Хотя эта реализация `beside` вполне работоспособна, в ней используется императивный стиль программирования, о чем явно свидетельствует наличие цикла, обходящего элементы массивов по индексам. В альтернативном варианте метод можно сократить до одного выражения:

```
new ArrayElement(
  for (
    (line1, line2) <- this.contents zip that.contents
  ) yield line1 + line2
)
```

Здесь с помощью оператора `zip` массивы `this.contents` и `that.contents` преобразуются в массив пар (так называются кортежи `Tuple2`). Оператор `zip` выбирает соответствующие элементы двух своих операндов и формирует массив пар. Например, выражение:

```
Array(1, 2, 3) zip Array("a", "b")
```

будет вычислено в следующее:

```
Array((1, "a"), (2, "b"))
```

Если один из двух массивов-операндов длиннее другого, то оставшиеся элементы оператор `zip` просто отбрасывает. В показанном ранее выражении третий элемент левого операнда, `3`, не формирует пару результата, поскольку для него не находится соответствующий элемент в правом операнде.

Затем выполняется обход элементов объединенного массива с помощью выражения `for`. Здесь синтаксис `for ((line1, line2) <- ...)` позволяет указать имена обоих элементов пары в одном *паттерне* (то есть теперь `line1` обозначает первый элемент пары, а `line2` — второй). Имеющаяся в Scala система сопоставления с образцом (паттерном) будет подробнее рассмотрена в главе 15. А сейчас все это можно представить себе как способ определения для каждого шага итерации двух `val`-переменных: `line1` и `line2`.

У выражения `for` есть часть `yield`, и поэтому оно выдает (англ. yields) результат. Данный результат того же вида, что и перебираемый выражением объект (то есть данный массив). Каждый элемент — результат объединения соответствующих рядов: `line1` и `line2`. Следовательно, конечный результат выполнения этого кода получается таким же, как и результат выполнения первой версии `beside`, но, поскольку в нем удалось обойтись без явной индексации массива, результат достигается способом, при котором допускается меньше ошибок.

Но вам все еще нужен способ отображения элементов. Как обычно, отображение выполняется с помощью определения метода `toString`, возвращающего элемент, отформатированный в виде строки. Его определение выглядит так:

```
override def toString = contents mkString "\n"
```

В реализации `toString` используется метод `mkString`, который определен для всех последовательностей, включая массивы. В разделе 7.8 было показано: такое

выражение, как `arr mkString sep`, возвращает строку, состоящую из всех элементов массива `arr`. Каждый элемент отображается на строку путем вызова его метода `toString`. Между последовательными элементами строк вставляется разделитель `sep`. Следовательно, выражение `contents mkString "\n"` форматирует содержимое массива как строку, где каждый элемент массива появляется в собственном ряду.

Обратите внимание: метод `toString` не требует указания пустого списка параметров. Это соответствует рекомендациям по соблюдению принципа единообразного доступа, поскольку `toString` — чистый метод, не получающий никаких параметров. После добавления этих трех методов класс `Element` приобретет вид, показанный в листинге 10.9.

Листинг 10.9. Класс Element с методами above, beside и toString

```
abstract class Element {

  def contents: Array[String]

  def width: Int =
    if (height == 0) 0 else contents(0).length

  def height: Int = contents.length

  def above(that: Element): Element =
    new ArrayElement(this.contents ++ that.contents)

  def beside(that: Element): Element =
    new ArrayElement(
      for (
        (line1, line2) <- this.contents zip that.contents
      ) yield line1 + line2
    )

  override def toString = contents mkString "\n"
}
```

10.13. Определяем фабричный объект

Теперь у вас есть иерархия классов для элементов разметки. Можно предоставить ее вашим клиентам как есть или выбрать технологию сокрытия иерархии за фабричным объектом.

В фабричном объекте содержатся методы, конструирующие другие объекты. Затем с помощью этих методов клиенты будут конструировать объекты вместо того, чтобы делать это, непосредственно используя ключевое слово `new`. Преимущество такого подхода заключается в возможности централизации создания объектов и в сокрытии способа представления объектов с помощью классов. Такое сокрытие сделает вашу библиотеку понятнее для клиентов, поскольку в открытом виде будет предоставлено меньше подробностей. Вдобавок оно обеспечит вам больше воз-

можностей вносить последующие изменения реализации библиотеки, не нарушая работу клиентского кода.

Первая задача при конструировании фабрики для элементов разметки — выбор места, в котором должны располагаться фабричные методы. Чьими элементами они должны быть — объекта-одиночки или класса? Как должен быть назван содержащий их объект или класс? Существует множество возможностей. Самое простое решение — создать объект-компаньон класса `Element` и превратить его в фабричный объект для элементов разметки. Таким образом, клиентам нужно предоставить только комбинацию «класс — объект `Element`», а реализацию трех классов `ArrayElement`, `LineElement` и `UniformElement` можно скрыть.

В листинге 10.10 представлена структура объекта `Element`, соответствующего этой схеме. В объекте `Element` содержатся три переопределяемых варианта метода `elem`, которые конструируют различный вид объекта разметки.

Листинг 10.10. Фабричный объект с фабричными методами

```
object Element {

  def elem(contents: Array[String]): Element =
    new ArrayElement(contents)

  def elem(chr: Char, width: Int, height: Int): Element =
    new UniformElement(chr, width, height)

  def elem(line: String): Element =
    new LineElement(line)
}
```

С появлением этих фабричных методов наметился смысл изменить реализацию класса `Element` таким образом, чтобы в нем вместо явного создания новых экземпляров `ArrayElement` выполнялись фабричные методы `elem`. Чтобы вызвать фабричные методы, не указывая с ними имя объекта-одиночки `Element`, мы импортируем в верхней части кода исходный файл `Element.elem`. Иными словами, вместо вызова фабричных методов с помощью указания `Element.elem` внутри класса `Element` мы импортируем `Element.elem`, чтобы можно было просто вызвать фабричные методы по имени `elem`. Код класса `Element` после внесения изменений показан в листинге 10.11.

Листинг 10.11. Класс Element, реорганизованный для использования фабричных методов

```
import Element.elem

abstract class Element {

  def contents: Array[String]

  def width: Int =
    if (height == 0) 0 else contents(0).length

  def height: Int = contents.length
```

```
  def above(that: Element): Element =
    elem(this.contents ++ that.contents)

  def beside(that: Element): Element =
    elem(
      for (
        (line1, line2) <- this.contents zip that.contents
      ) yield line1 + line2
    )

  override def toString = contents mkString "\n"
}
```

Кроме того, благодаря наличию фабричных методов теперь подклассы `ArrayElement`, `LineElement` и `UniformElement` могут стать приватными, поскольку отпадет надобность непосредственного обращения к ним со стороны клиентов. В Scala классы и объекты-одиночки можно определять внутри других классов и объектов-одиночек. Один из способов превратить подклассы класса `Element` в приватные — поместить их внутрь объекта-одиночки `Element` и объявить их там приватными. Классы по-прежнему будут доступны трем фабричным методам `elem` там, где в них есть надобность. Как это будет выглядеть, показано в листинге 10.12.

Листинг 10.12. Сокрытие реализации с помощью использования приватных классов

```
object Element {

  private class ArrayElement(
    val contents: Array[String]
  ) extends Element

  private class LineElement(s: String) extends Element {
    val contents = Array(s)
    override def width = s.length
    override def height = 1
  }

  private class UniformElement(
    ch: Char,
    override val width: Int,
    override val height: Int
  ) extends Element {
    private val line = ch.toString * width
    def contents = Array.fill(height)(line)
  }

  def elem(contents: Array[String]): Element =
    new ArrayElement(contents)

  def elem(chr: Char, width: Int, height: Int): Element =
    new UniformElement(chr, width, height)

  def elem(line: String): Element =
    new LineElement(line)
}
```

10.14. Методы heighten и widen

Нам нужно внести еще одно, последнее усовершенствование. Версия `Element`, показанная в листинге 10.11 (см. выше), не может всецело нас устроить, поскольку не позволяет клиентам помещать друг на друга элементы разной ширины или помещать рядом друг с другом элементы разной высоты.

Например, вычисление следующего выражения не будет работать корректно, так как второй ряд в объединенном элементе длиннее первого:

```
new ArrayElement(Array("hello")) above
new ArrayElement(Array("world!"))
```

Аналогично этому вычисление следующего выражения не будет работать правильно из-за того, что высота первого элемента `ArrayElement` составляет два ряда, а второго — только один:

```
new ArrayElement(Array("one", "two")) beside
new ArrayElement(Array("one"))
```

В листинге 10.13 показан приватный вспомогательный метод по имени `widen`, который получает ширину и возвращает объект `Element` указанной ширины. Результат включает в себя содержимое этого объекта, которое для достижения нужной ширины отцентрировано за счет создания отступов справа и слева с помощью любого нужного для этого количества пробелов. В листинге также показан похожий метод `heighten`, выполняющий то же в вертикальном направлении. Метод `widen` вызывается методом `above`, чтобы обеспечить одинаковую ширину элементов, которые помещаются друг над другом. Аналогично этому метод `heighten` вызывается методом `beside`, чтобы обеспечить одинаковую высоту элементов, помещаемых рядом друг с другом. После внесения этих изменений библиотека разметки будет готова к использованию.

Листинг 10.13. Класс Element с методами widen и heighten

```
import Element.elem

abstract class Element {
  def contents: Array[String]

  def width: Int = contents(0).length
  def height: Int = contents.length

  def above(that: Element): Element = {
    val this1 = this widen that.width
    val that1 = that widen this.width
    elem(this1.contents ++ that1.contents)
  }

  def beside(that: Element): Element = {
    val this1 = this heighten that.height
    val that1 = that heighten this.height
    elem(
      for ((line1, line2) <- this1.contents zip that1.contents)
      yield line1 + line2)
  }
```

```
  def widen(w: Int): Element =
    if (w <= width) this
    else {
      val left = elem(' ', (w - width) / 2, height)
      val right = elem(' ', w - width - left.width, height)
      left beside this beside right
    }

  def heighten(h: Int): Element =
    if (h <= height) this
    else {
      val top = elem(' ', width, (h - height) / 2)
      val bot = elem(' ', width, h - height - top.height)
      top above this above bot
    }

  override def toString = contents mkString "\n"
}
```

10.15. Собираем все воедино

Интересным способом применения почти всех элементов библиотеки разметки будет написание программы, рисующей спираль с заданным количеством ребер. Ее созданием займется программа `Spiral`, показанная в листинге 10.14.

Листинг 10.14. Приложение Spiral

```
import Element.elem

object Spiral {

  val space = elem(" ")
  val corner = elem("+")

  def spiral(nEdges: Int, direction: Int): Element = {
    if (nEdges == 1)
      elem("+")
    else {
      val sp = spiral(nEdges - 1, (direction + 3) % 4)
      def verticalBar = elem('|', 1, sp.height)
      def horizontalBar = elem('-', sp.width, 1)
      if (direction == 0)
        (corner beside horizontalBar) above (sp beside space)
      else if (direction == 1)
        (sp above space) beside (corner above verticalBar)
      else if (direction == 2)
        (space beside sp) above (horizontalBar beside corner)
      else
        (verticalBar above corner) beside (space above sp)
    }
  }
```

```
  def main(args: Array[String]) = {
    val nSides = args(0).toInt
    println(spiral(nSides, 0))
  }
}
```

Поскольку `Spiral` является самостоятельным объектом с методом `main`, имеющим надлежащую сигнатуру, этот код можно считать приложением, написанным на Scala. `Spiral` получает один аргумент командной строки в виде целого числа и рисует спираль с указанным количеством граней. Например, можно нарисовать шестигранную спираль, как показано слева, и более крупную спираль, как показано справа.

```
$ scala Spiral 6
+-----
|
| +-+
| + |
|   |
+---+
```

```
$ scala Spiral 11
+----------
|
| +------+
| |      |
| | +--+ |
| | |  | |
| | ++ | |
| |    | |
| +----+ |
|        |
+--------+
```

```
$ scala Spiral 17
+-----------------
|
| +------------+
| |            |
| | +--------+ |
| | |        | |
| | | +----+ | |
| | | |    | | |
| | | | ++ | | |
| | | |  | | | |
| | | +--+ | | |
| | |      | | |
| | +------+ | |
| |          | |
| +----------+ |
|              |
+--------------+
```

Резюме

В этой главе мы рассмотрели дополнительные концепции объектно-ориентированного программирования на языке Scala. Среди них — абстрактные классы, наследование и создание подтипов, иерархии классов, параметрические поля и переопределение методов. У вас должно было выработаться понимание способов создания в Scala оригинальных иерархий классов. А к работе с библиотекой раскладки мы еще вернемся в главе 14.

11 Иерархия Scala

После изучения в предыдущей главе подробностей наследования классов самое время немного отступить и посмотреть на иерархию классов Scala в целом. В Scala каждый класс наследуется от общего суперкласса по имени `Any`. Поскольку каждый класс является подклассом `Any`, то методы, определенные в классе `Any`, универсальны: их можно вызвать в отношении любого объекта. В самом низу иерархии в Scala также определяются довольно интересные классы `Null` и `Nothing`, которые, по сути, выступают в роли общих *под*классов. Например, в то время, как `Any` — суперкласс для всех классов, `Nothing` — подкласс для любого класса. В данной главе мы проведем экскурсию по имеющейся в Scala иерархии классов.

11.1. Иерархия классов Scala

На рис. 11.1 в общих чертах показана иерархия классов Scala. На вершине иерархии находится класс `Any`; в нем определяются методы, в число которых входят:

```
final def ==(that: Any): Boolean
final def !=(that: Any): Boolean
def equals(that: Any): Boolean
def ##: Int
def hashCode: Int
def toString: String
```

Все классы — наследники класса `Any`, поэтому каждый объект в программе на Scala можно подвергуть сравнению с помощью методов `==`, `!=` или `equals`, хешированию с использованием методов `##` или `hashCode` и форматированию, прибегнув к методу `toString`. Методы определения равенства и неравенства `==` и `!=` объявлены в классе `Any` как `final`, следовательно, переопределить их в подклассах невозможно. Метод `==` — по сути, то же самое, что и метод `equals`, а метод `!=` всегда является отрицанием метода `equals`[1]. Таким образом, отдельные классы могут перекроить

[1] Единственный случай, когда использование == не приводит к непосредственному вызову equals, относится к упакованным числовым классам Java, таким как Integer или Long. В Java new Integer(1) не эквивалентен new Long(1) даже в случае применения

смысл значения метода `==` или `!=`, переопределив метод `equals`. Соответствующий пример будет показан в этой главе чуть позже.

У корневого класса `Any` имеется два подкласса: `AnyVal` и `AnyRef`. Класс `AnyVal` является родительским для *классов значений* в Scala. Наряду с возможностью определять собственные классы значений (см. раздел 11.4), Scala имеет девять встроенных классов значений: `Byte`, `Short`, `Char`, `Int`, `Long`, `Float`, `Double`, `Boolean` и `Unit`. Первые восемь из них соответствуют примитивным типам Java, и их значения во время выполнения программы представляются в виде примитивных значений Java. Все экземпляры этих классов написаны в Scala в виде литералов. Например, `42` — экземпляр класса `Int`, `'x'` — экземпляр `Char`, а `false` — экземпляр класса `Boolean`. Создать экземпляры этих классов, используя ключевое слово `new`, невозможно. Этому препятствует особый прием, состоящий в том, что все классы значений определены одновременно и как абстрактные, и как финальные.

Поэтому, если воспользоваться следующим кодом:

```
scala> new Int
```

то будет получен такой результат:

```
<console>:5: error: class Int is abstract; cannot be
instantiated
       new Int
       ^
```

Еще один класс значений, `Unit`, примерно соответствует имеющемуся в Java типу `void` — он используется в качестве результирующего типа выполнения метода, который не возвращает содержательного результата. Как упоминалось в разделе 7.2, у `Unit` имеется единственное значение экземпляра, которое записывается как `()`.

В соответствии с объяснениями, изложенными в главе 5, в классах значений в качестве методов поддерживаются обычные арифметические и логические (булевы) операторы. Например, у класса `Int` имеются методы `+` и `*`, а у класса `Boolean` — методы `||` и `&&`. Классы значений также наследуют все методы из класса `Any`. Это можно протестировать в интерпретаторе:

```
scala> 42.toString
res1: String = 42

scala> 42.hashCode
res2: Int = 42

scala> 42 equals 42
res3: Boolean = true
```

примитивных значений 1 == 1L. Поскольку Scala — более регулярный язык, чем Java, появилась необходимость скорректировать это несоответствие, задействовав для этих классов особую версию метода ==. Точно так же метод ## обеспечивает Scala-версию хеширования и похож на Java-метод hashCode, за исключением того, что для упакованных числовых типов он всегда работает с методом ==. Например, для new Integer(1) и new Long(1) метод ## вычисляет один и тот же хеш, тогда как Java-методы hashCode вычисляют разный хеш-код.

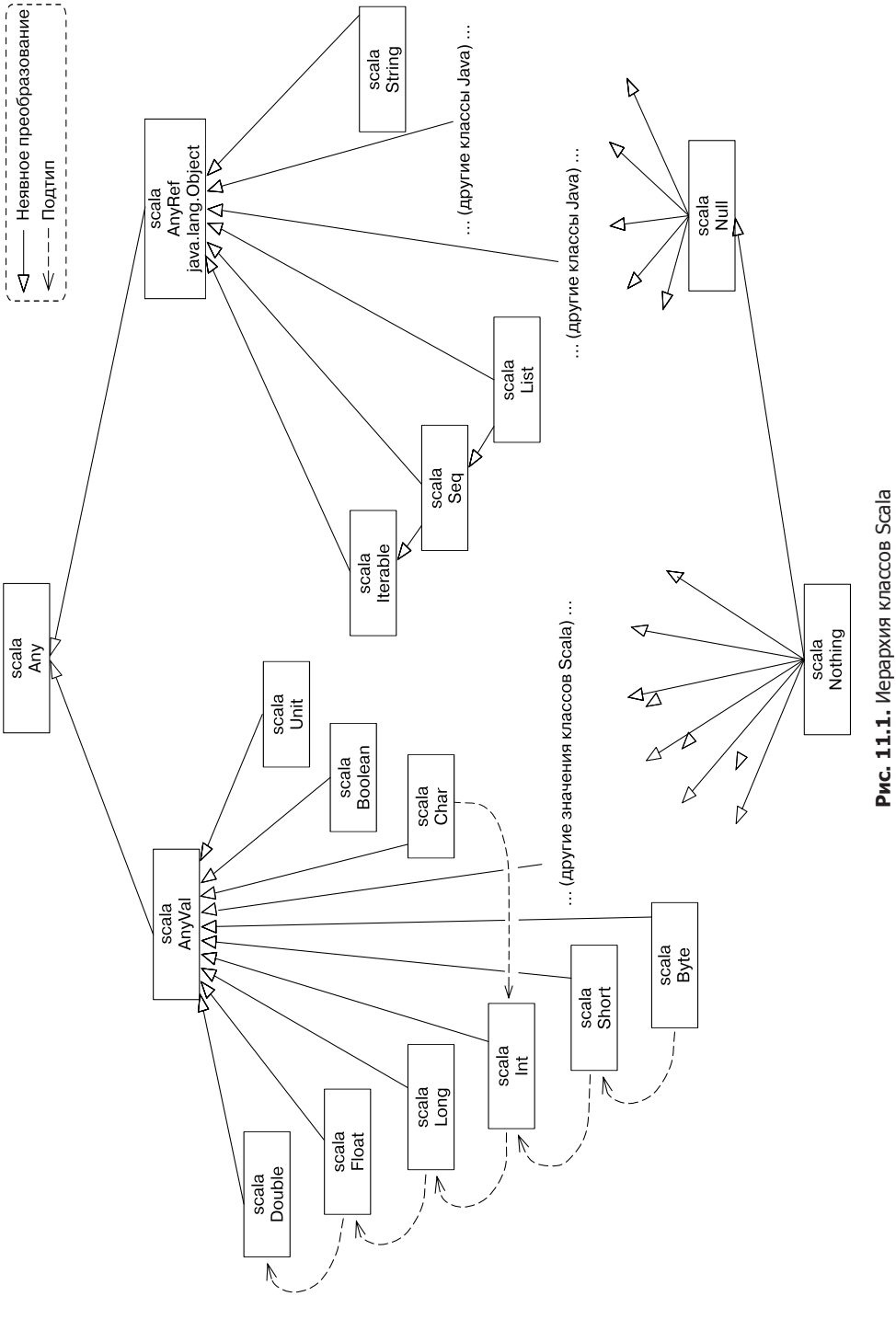


Рис. 11.1. Иерархия классов Scala

Следует отметить, что пространство классов значений плоское: все классы значений являются подтипами `scala.AnyVal`, но не являются подклассами друг друга. Вместо этого между различными типами классов значений существует неявное преобразование типов. Например, экземпляр класса `scala.Int`, когда это требуется, автоматически расширяется (путем неявного преобразования) в экземпляр класса `scala.Long`.

Как упоминалось в разделе 5.10, неявное преобразование используется также для добавления дополнительных функциональных возможностей к типам значений. Например, тип `Int` поддерживает все показанные далее операции:

```
scala> 42 max 43
res4: Int = 43

scala> 42 min 43
res5: Int = 42

scala> 1 until 5
res6: scala.collection.immutable.Range = Range 1 until 5

scala> 1 to 5
res7: scala.collection.immutable.Range.Inclusive
  = Range 1 to 5

scala> 3.abs
res8: Int = 3

scala> (-3).abs
res9: Int = 3
```

Работает это следующим образом: все методы `min`, `max`, `until`, `to` и `abs` определены в классе `scala.runtime.RichInt`, а между классами `Int` и `RichInt` существует неявное преобразование. Оно применяется при вызове в отношении `Int`-объекта метода, который определен не в классе `Int`, а в классе `RichInt`. По аналогии с этим «классы-усилители» и неявные преобразования существуют и для других классов значений. Более подробно неявные преобразования будут рассматриваться в главе 21.

Другой подкласс корневого класса `Any` — класс `AnyRef`. Это базовый класс всех *ссылочных классов* в Scala. Как упоминалось ранее, на платформе Java `AnyRef` фактически является псевдонимом класса `java.lang.Object`. Следовательно, все классы, написанные на Java и Scala, — наследники класса `AnyRef`[1]. Поэтому о `java.lang.Object` можно думать как о способе реализации `AnyRef` на платформе Java. Следовательно, хоть `Object` и `AnyRef` и можно взаимозаменяемо использовать в программах Scala на платформе Java, рекомендуемым стилем будет повсеместное применение `AnyRef`.

[1] Причина существования псевдонима AnyRef, заменяющего использование имени java.lang.Object, заключается в том, что Scala изначально разрабатывался для работы как на платформе Java, так и на платформе .NET. На платформе .NET AnyRef был псевдонимом для System.Object.

11.2. Как реализованы примитивы

Как все это реализовано? Фактически в Scala целочисленные значения хранятся так же, как и в Java, — в виде 32-разрядных слов. Это необходимо для эффективной работы виртуальной машины Java (JVM), а также обеспечения возможности совместной работы с библиотеками Java. Такие стандартные операции, как сложение или умножение, реализуются в качестве примитивных операций. Однако Scala использует «дублирующий» класс `java.lang.Integer` везде, где целое число должно выглядеть как (Java) объект. Так происходит, например, при вызове в отношении целого числа метода `toString` или присваивании целого числа переменной типа `Any`. При необходимости целочисленные значения типа `Int` явно преобразуются в упакованные целые числа типа `java.lang.Integer`.

Это во многом походит на автоупаковку (auto-boxing) в Java 5, два процесса действительно очень похожи. Но все-таки есть одно коренное различие: упаковка в Scala гораздо менее заметна, чем упаковка в Java. Попробуйте выполнить в Java следующий код:

```
// Это код на языке Java
boolean isEqual(int x, int y) {
  return x == y;
}
System.out.println(isEqual(421, 421));
```

В результате, конечно же, будет получено значение `true`. А теперь измените типы аргументов `isEqual` на `java.lang.Integer` (или с аналогичным результатом на на `Object`):

```
// Это код на языке Java
boolean isEqual(Integer x, Integer y) {
  return x == y;
}
System.out.println(isEqual(421, 421));
```

Получите результат `false`! Оказывается, число 421 было упаковано дважды, поэтому аргументами для `x` и `y` стали два разных объекта. Применение `==` в отношении ссылочных типов означает равенство ссылок, а `Integer` — ссылочный тип, вследствие чего в результате получается `false`. Это один из аспектов, свидетельствующий о том, что Java не является чистым объектно-ориентированным языком. Существует четко видимая разница между примитивными и ссылочными типами.

Теперь попробуйте провести тот же самый эксперимент на Scala:

```
scala> def isEqual(x: Int, y: Int) = x == y
isEqual: (x: Int, y: Int)Boolean

scala> isEqual(421, 421)
res10: Boolean = true
```

```
scala> def isEqual(x: Any, y: Any) = x == y
isEqual: (x: Any, y: Any)Boolean

scala> isEqual(421, 421)
res11: Boolean = true
```

Операция равенства в Scala разработана так, чтобы быть понятной относительно представления типа. Для типов значений (числовых или логических) это вполне естественное равенство. Для ссылочных типов, отличающихся от упакованных числовых типов Java, `==` рассматривается в качестве псевдонима метода `equals`, унаследованного от класса `Object`. Данный метод изначально определен в целях выявления равенства ссылок, но во многих подклассах переопределяется для реализации их естественных представлений о равенстве. Это также означает, что в Scala вы никогда не попадете в хорошо известную в Java ловушку, касающуюся сравнения строк. В Scala оно работает вполне корректно:

```
scala> val x = "abcd".substring(2)
x: String = cd

scala> val y = "abcd".substring(2)
y: String = cd

scala> x == y
res12: Boolean = true
```

В Java результатом сравнения `x` с `y` будет `false`. В таком случае программисты вынуждены пользоваться методом `equals`, но данную тонкость нетрудно упустить из виду.

Может сложиться и такая ситуация, при которой вместо равенства, определенного пользователем, нужно проверить равенство ссылок. Так, в некоторых ситуациях, когда эффективность важнее всего, вы можете захотеть выполнить конструирование хеша *(hash cons)* некоторых классов и проверить их экземпляры на равенство ссылок[1]. Для таких случаев в классе `AnyRef` определен дополнительный метод `eq`, который не может быть переопределен и реализован как проверка равенства ссылок (то есть для ссылочных типов ведет себя подобно `==` в Java). Существует также отрицание `eq`, которое называется `ne`, например:

```
scala> val x = new String("abc")
x: String = abc

scala> val y = new String("abc")
y: String = abc
```

[1] Вы можете выполнить конструирование хеша экземпляров класса путем кэширования всех созданных экземпляров в слабую коллекцию. Затем, как только потребуется новый экземпляр класса, сначала проверяется кэш. Если в нем уже есть элемент, равный тому, который вы намереваетесь создать, то можно повторно воспользоваться существующим экземпляром. В результате такой систематизации любые два экземпляра, равенство которых определяется с помощью метода equals(), также равны на основе равенства ссылок.

```
scala> x == y
res13: Boolean = true

scala> x eq y
re14: Boolean = false

scala> x ne y
res15: Boolean = true
```

Более подробно равенство в Scala рассматривается в главе 30.

11.3. Низшие типы

Внизу иерархии на рис. 11.1 (см. выше) показаны два класса: `scala.Null` и `scala.Nothing`. Это особые типы, единообразно сглаживающие острые углы объектно-ориентированной системы типов в Scala.

Класс `Null` — тип нулевой ссылки `null`: он представляет собой подкласс каждого ссылочного класса, то есть каждого класса, который сам является наследником класса `AnyRef`. Класс `Null` несовместим с типами значений. Нельзя, к примеру, присвоить значение `null` целочисленной переменной:

```
scala> val i: Int = null
                    ^
       error: an expression of type Null is ineligible for implicit conversion
```

Тип `Nothing` находится в самом низу иерархии классов Scala: он представляет собой подтип любого другого типа. Но значений этого типа вообще не существует. А зачем нужен тип без значений? Как говорилось в разделе 7.4, `Nothing` используется, в частности, для того, чтобы сигнализировать об аварийном завершении операции.

Например, в объекте `sys` стандартной библиотеки Scala есть метод `error`, имеющий такое определение:

```
def error(message: String): Nothing =
  throw new RuntimeException(message)
```

Возвращаемым типом метода `error` является `Nothing`, что говорит пользователю о ненормальном возвращении из метода (вместо этого метод сгенерировал исключение). Поскольку `Nothing` — подтип любого другого типа, то методы, подобные `error`, допускают весьма гибкое использование, например:

```
def divide(x: Int, y: Int): Int =
  if (y != 0) x / y
  else sys.error("деление на нуль невозможно")
```

Ветка `then` данного условия, представленная выражением `x / y`, имеет тип `Int`, а ветка `else`, то есть вызов `error`, имеет тип `Nothing`. Поскольку `Nothing` — подтип `Int`, то типом всего условного выражения, как и требовалось, является `Int`.

11.4. Определение собственных классов значений

В разделе 11.1 говорилось, что в дополнение к встроенным классам значений можно определять собственные. Как и экземпляры встроенных, экземпляры ваших классов значений будут, как правило, компилироваться в байт-код Java, который не задействует класс-оболочку. В том контексте, где нужна оболочка, например, при использовании обобщенного кода, значения будут упаковываться и распаковываться автоматически.

Классами значений можно сделать только вполне определенные классы. Чтобы класс стал классом значений, он должен иметь только один параметр и не должен иметь внутри ничего, кроме `def`-определений. Более того, класс значений не может расширяться никакими другими классами и в нем не могут переопределяться методы `equals` или `hashCode`.

Чтобы определить класс значений, его нужно сделать подклассом класса `AnyVal` и поставить перед его единственным параметром префикс `val`. Пример класса значений выглядит так:

```
class Dollars(val amount: Int) extends AnyVal {
  override def toString() = "$" + amount
}
```

В соответствии с описанием, приведенным в разделе 10.6, префикс `val` позволяет иметь доступ к параметру `amount` как к полю. Например, следующий код создает экземпляр класса значений, а затем извлекает из него `amount`:

```
scala> val money = new Dollars(1000000)
money: Dollars = $1000000
scala> money.amount
res16: Int = 1000000
```

В данном примере `money` ссылается на экземпляр класса значений. Эта переменная в исходном коде Scala имеет тип `Dollars`, но скомпилированный байт-код Java будет напрямую использовать тип `Int`.

В этом примере определяется метод `toString`, и компилятор понимает, когда его использовать. Именно поэтому вывод значения `money` дает результат `$1000000`, со знаком доллара, а вывод `money.amount` дает результат `1000000`. Можно даже определить несколько типов значений, и все они будут опираться на одно и то же `Int`-значение, например:

```
class SwissFrancs(val amount: Int) extends AnyVal {
  override def toString() = s"$amount CHF"
}
```

Хотя и `Dollars`, и `SwissFrancs` представлены в виде целых чисел, ничто не мешает использовать их в одной и той же области видимости:

```
scala> val dollars = new Dollars(1000)
dollars: Dollars = $1000
scala> val francs = new SwissFrancs(1000)
francs: SwissFrancs = 1000 CHF
```

Уход от монокультурности типов

Чтобы получить наибольшие преимущества от использования иерархии классов Scala, старайтесь для каждого понятия предметной области определять новый класс, несмотря на то что будет возможность повторно применять один и тот же класс для различных целей. Даже если он относится к так называемому *крошечному (tiny) muny*, не имеющему методов или полей, определение дополнительного класса поможет компилятору принести вам больше пользы.

Предположим, вы написали некий код для генерации HTML. В HTML название стиля представлено в виде строки. То же самое касается и идентификаторов привязки. Сам код HTML также является строкой, поэтому при желании представить все здесь перечисленное можно с помощью определения вспомогательного кода, используя строки наподобие этих:

```
def title(text: String, anchor: String, style: String): String =
  s"<a id='$anchor'><h1 class='$style'>$text</h1></a>"
```

В данной сигнатуре типа четыре строки! Такой *строчно типизированный* код с технической точки зрения является строго типизированным. Однако все, что находится здесь в поле зрения, относится к типу `String`, поэтому компилятор не может помочь вам отличить один элемент структуры от другого. Например, не сможет уберечь вас от следующего искажения структуры:

```
scala> title("chap:vcls", "bold", "Value Classes")
res17: String = <a id='bold'><h1 class='Value
    Classes'>chap:vcls</h1></a>
```

Код HTML нарушен. При намерении вывести на экран текст `Value Classes` эта строка была использована в качестве класса стиля, а на экран был выведен текст `chap.vcls`, для которого предполагалась роль гипертекстовой ссылки. В довершение ко всему в качестве идентификатора гипертекстовой ссылки выступила строка `bold`, для которой предполагалась роль класса стиля. Несмотря на всю эту комедию ошибок, компилятор никак этому не воспротивился.

Если определить для каждого понятия предметной области крошечный тип, то компилятор сможет принести больше пользы. Например, можно определить собственный небольшой класс для стилей, идентификаторов гипертекстовых ссылок, отображаемого текста и кода HTML. Поскольку эти классы имеют один параметр и не имеют элементов, то могут быть определены как классы значений:

```
class Anchor(val value: String) extends AnyVal
class Style(val value: String) extends AnyVal
class Text(val value: String) extends AnyVal
class Html(val value: String) extends AnyVal
```

Наличие этих классов позволяет создать версию `title`, обладающую менее тривиальной сигнатурой типов наподобие такой:

```
def title(text: Text, anchor: Anchor, style: Style): Html =
  new Html(
    s"<a id='${anchor.value}'>" +
```

```
        s"<h1 class='${style.value}'>" +
        text.value +
        "</h1></a>"
)
```

Теперь при попытке воспользоваться этой версией с аргументами, указанными в неверном порядке, компилятор сможет обнаружить ошибку, например:

```
scala> title(new Anchor("chap:vcls"), new Style("bold"),
           new Text("Value Classes"))
             ^
error: type mismatch;
 found   : Anchor
 required: Text
                                      ^
error: type mismatch;
 found   : Style
 required: Anchor
    new Text("Value Classes"))
    ^
On line 2: error: type mismatch;
 found   : Text
 required: Style
```

Резюме

В этой главе мы показали классы, находящиеся на самом верху и в самом низу иерархии классов Scala. Теперь, получив твердую базу знаний о наследовании в Scala, вы готовы понять композицию примесей. Следующая глава будет посвящена трейтам.

12 Трейты

Трейты в Scala являются фундаментальными повторно используемыми блоками кода. В трейте инкапсулируются определения тех методов и полей, которые затем могут повторно использоваться путем их примешивания в классы. В отличие от наследования классов, в котором каждый класс должен быть наследником только одного суперкласса, в класс может примешиваться любое количество трейтов. В этой главе мы покажем, как работают трейты. Далее рассмотрим два наиболее распространенных способа их применения: расширение «тонких» интерфейсов и превращение их в «толстые», а также определение наращиваемых модификаций. Здесь мы покажем, как используется трейт `Ordered`, и сравним механизм трейтов с множественным наследованием, имеющимся в других языках.

12.1. Как работают трейты

Определение трейта похоже на определение класса, за исключением того, что в нем используется ключевое слово `trait`. Пример показан в листинге 12.1.

Листинг 12.1. Определение трейта Philosophical

```
trait Philosophical {
  def philosophize() = {
    println("На меня тратится память, следовательно, я существую!")
  }
}
```

Данный трейт называется `Philosophical`. В нем не объявлен суперкласс, следовательно, как и у класса, у него есть суперкласс по умолчанию — `AnyRef`. В нем определяется один конкретный метод по имени `philosophize`. Это простой трейт, и его вполне достаточно, чтобы показать, как работают трейты.

После того как трейт определен, он может быть *примешан в* класс с помощью ключевого слова: либо `extends`, либо `with`. Программисты, работающие со Scala, примешивают трейты, а не наследуют их, поскольку примешивание трейта весьма

отличается от множественного наследования, встречающегося во многих других языках. Этот вопрос рассматривается в разделе 12.6. Например, в листинге 12.2 показан класс, в который с помощью ключевого слова `extends` примешивается трейт `Philosophical`.

Листинг 12.2. Примешивание трейта с использованием ключевого слова extends

```
class Frog extends Philosophical {
  override def toString = "зеленая"
}
```

Примешивать трейт можно с помощью ключевого слова `extends`, в таком случае происходит неявное наследование суперкласса трейта. Например, в листинге 12.2 класс `Frog` (Лягушка) становится подклассом `AnyRef` (это суперкласс для трейта `Philosophical`) и примешивает в себя трейт `Philosophical`. Методы, унаследованные от трейта, могут использоваться точно так же, как и методы, унаследованные от суперкласса. Рассмотрим пример:

```
scala> val frog = new Frog
frog: Frog = зеленая

scala> frog.philosophize()
На меня тратится память, следовательно, я существую!
```

Трейт также определяет тип. Рассмотрим пример, в котором `Philosophical` используется как тип:

```
scala> val phil: Philosophical = frog
phil: Philosophical = зеленая

scala> phil.philosophize()
На меня тратится память, следовательно, я существую!
```

Типом `phil` является `Philosophical`, то есть трейт. Таким образом, переменная `phil` может быть инициализирована любым объектом, в чей класс примешан трейт `Philosophical`.

Если нужно примешать трейт в класс, который явно расширяет суперкласс, то ключевое слово `extends` используется для указания суперкласса, а для примешивания трейта — ключевое слово `with`. Пример показан в листинге 12.3. Если нужно примешать сразу несколько трейтов, то дополнительные трейты указываются с помощью ключевого слова `with`. Например, располагая трейтом `HasLegs`, вы, как показано в листинге 12.4, можете примешать в класс `Frog` как трейт `Philosophical`, так и трейт `HasLegs`.

Листинг 12.3. Примешивание трейта с использованием ключевого слова with

```
class Animal

class Frog extends Animal with Philosophical {
  override def toString = "зеленая"
}
```

Листинг 12.4. Примешивание нескольких трейтов

```
class Animal
trait HasLegs

class Frog extends Animal with Philosophical with HasLegs {
  override def toString = "зеленая"
}
```

В показанных ранее примерах класс `Frog` наследовал реализацию метода `philosophize` из трейта `Philosophical`. В качестве альтернативного варианта метод `philosophize` в классе `Frog` может быть переопределен. Синтаксис выглядит точно так же, как и при переопределении метода, объявленного в суперклассе. Рассмотрим пример:

```
class Animal

class Frog extends Animal with Philosophical {
  override def toString = "зеленая"
  override def philosophize() = {
    println("Мне живется нелегко, потому что я " + toString + "!")
  }
}
```

В новое определение класса `Frog` по-прежнему примешивается трейт `Philosophical`, поэтому его, как и раньше, можно использовать из переменной данного типа. Но, так как во `Frog` переопределено определение метода `philosophize`, которое было дано в трейте `Philosophical`, при вызове будет получено новое поведение:

```
scala> val phrog: Philosophical = new Frog
phrog: Philosophical = зеленая

scala> phrog.philosophize()
Мне нелегко живется, потому что я зеленая!
```

Теперь можно прийти к философскому умозаключению, что трейты подобны Java-интерфейсам с конкретными методами, но фактически их возможности гораздо шире. Так, в трейтах можно объявлять поля и сохранять состояние. Фактически в определении трейта можно делать то же самое, что и в определении класса, и синтаксис выглядит почти так же, но с двумя исключениями.

Начнем с того, что в трейте не может быть никаких присущих классу параметров (то есть параметров, передаваемых первичному конструктору класса). Иными словами, хотя у вас есть возможность определить класс таким вот образом:

```
class Point(x: Int, y: Int)
```

следующая попытка определить трейт окажется неудачной:

```
trait NoPoint(x: Int, y: Int) // Не пройдет компиляцию
```

Способ обойти это ограничение будет показан в разделе 20.5.

Второе отличие классов от трейтов заключается в том, что в классах вызовы `super` имеют статическую привязку, а в трейтах — динамическую. Если в классе

воспользоваться кодом `super.toString`, то вы будете точно знать, какая именно реализация метода будет вызвана. Но когда точно такой же код применяется в трейте, то вызываемая с помощью `super` реализация метода при определении трейта еще не определена. Вызываемая реализация станет определяться заново при каждом примешивании трейта в конкретный класс. Такое своеобразное поведение `super` является ключевым фактором, позволяющим трейтам работать в качестве *наращиваемых модификаций*, и рассматривается в разделе 12.5. А правила разрешения вызовов `super` будут изложены в разделе 12.6.

12.2. Сравнение «тонких» и «толстых» интерфейсов

Чаще всего трейты используются для автоматического добавления к классу методов в дополнение к тем методам, которые в нем уже имеются. То есть трейты способны расширить *«тонкий»* интерфейс, превратив его в *«толстый»*.

Противопоставление «тонких» интерфейсов «толстым» представляет собой компромисс, который довольно часто встречается в объектно-ориентированном проектировании. Это компромисс между теми, кто реализует интерфейс, и теми, кто им пользуется. В «толстых» интерфейсах имеется множество методов, обеспечивающих удобство применения для тех, кто их вызывает. Клиенты могут выбрать метод, целиком отвечающий их функциональным запросам. В то же время «тонкий» интерфейс имеет незначительное количество методов и поэтому проще обходится тем, кто их реализует. Но клиентам, обращающимся к «тонким» интерфейсам, приходится создавать больше собственного кода. При более скудном выборе доступных для вызова методов им приходится выбирать то, что хотя бы в какой-то мере отвечает их потребностям, а чтобы использовать выбранный метод, им требуется создавать дополнительный код.

Java-интерфейсы чаще «тонкие», чем «толстые». Например, интерфейс `CharSequence`, появившийся в Java 1.4, — «тонкий», общий для всех строкоподобных классов, которые содержат последовательность символов. А вот как выглядит определение этого интерфейса, если его рассматривать в качестве Scala-трейта:

```
trait CharSequence {
  def charAt(index: Int): Char
  def length: Int
  def subSequence(start: Int, end: Int): CharSequence
  def toString(): String
}
```

Хотя к любой последовательности символов, для которой предназначен интерфейс `CharSequence`, может применяться большинство из десятков методов, имеющихся в классе `String`, в Java-интерфейсе `CharSequence` объявляются всего лишь четыре метода. Если бы в `CharSequence` был включен полный интерфейс `String`, то это бы стало гораздо более весомой нагрузкой на реализаторов `CharSequence`.

Каждому программисту, реализующему `CharSequence` в Java, пришлось бы определять десятки дополнительных методов. Поскольку в трейтах Scala могут содержаться конкретные методы, они делают «толстые» интерфейсы намного более удобными. Добавление в трейт конкретного метода уводит компромисс «тонкий — толстый» в сторону более «толстых» интерфейсов. В отличие от Java, добавление конкретного метода к Scala-трейту — одноразовое действие. Вам нужно единожды реализовать тот или иной метод, сделав это в самом трейте, вместо того чтобы возиться с его повторной реализацией для каждого класса, в который примешивается трейт. Таким образом, создание «толстых» интерфейсов в Scala требует меньше работы, чем в языках без трейтов.

Чтобы расширить интерфейс с помощью трейтов, просто определите трейт с небольшим количеством абстрактных методов — тонкую часть интерфейса трейта — и с потенциально большим количеством конкретных методов, реализованных в терминах абстрактных методов. Затем можно будет примешать расширяющий трейт в класс, реализовав тонкую часть интерфейса, и получить в результате класс, позволяющий обеспечить доступ ко всему доступному «толстому» интерфейсу.

12.3. Пример: прямоугольные объекты

В графических библиотеках часто содержится множество различных классов, представляющих что-либо прямоугольное. В качестве примеров можно привести окна, растровые изображения и области, выбираемые с помощью мыши. Чтобы такие прямоугольные объекты использовать с бо́льшим удобством, было бы неплохо, если бы библиотека предоставляла ответы на запросы о геометрических свойствах, таких как ширина (`width`), высота (`height`), левый край (`left`), правый край (`right`), верхний левый угол (`topLeft`) и т. д. Однако существует очень много таких методов, которые было бы хорошо иметь, поэтому на разработчиков библиотеки ложится тяжкий груз предоставить все эти методы для всех прямоугольных объектов из библиотеки Java. Если такая же библиотека была написана на Scala, то, напротив, ее создатели без особого труда могли бы предоставить все эти удобные методы каким угодно классам, воспользовавшись трейтами.

Чтобы понять, как это делается, сначала представим, как бы все это могло выглядеть, если не использовать трейты. Должны существовать какие-то основные геометрические классы наподобие точки — `Point` — и прямоугольника — `Rectangle`:

```
class Point(val x: Int, val y: Int)

class Rectangle(val topLeft: Point, val bottomRight: Point) {
  def left = topLeft.x
  def right = bottomRight.x
  def width = right - left
  // и множество других геометрических методов...
}
```

Класс `Rectangle` получает в свой первичный конструктор две точки: координаты верхнего левого и нижнего правого углов. Затем с помощью простых вычислений, основанных на координатах этих двух точек, он реализует множество удобных методов, таких как `left`, `right` и `width`.

Еще один класс, который может войти в состав графической библиотеки, — это двумерный графический виджет:

```
abstract class Component {
  def topLeft: Point
  def bottomRight: Point

  def left = topLeft.x
  def right = bottomRight.x
  def width = right - left
  // и множество других геометрических методов...
}
```

Обратите внимание: определения `left`, `right` и `width` в обоих классах абсолютно одинаковы. Практически одинаковыми, с незначительными вариациями, они будут и в любых других классах прямоугольных объектов.

Уменьшить эту повторяемость можно, прибегнув к расширяющему трейту. Он будет иметь два абстрактных метода: один станет возвращать верхнюю левую координату объекта, а другой — нижнюю правую. Затем в нем могут быть предоставлены конкретные реализации всех других геометрических запросов. На что это будет похоже, показано в листинге 12.5.

Листинг 12.5. Определение расширяющего трейта

```
trait Rectangular {
  def topLeft: Point
  def bottomRight: Point

  def left = topLeft.x
  def right = bottomRight.x
  def width = right - left
  // и множество других геометрических методов...
}
```

Этот трейт можно примешать в класс `Component`, чтобы получить все геометрические методы, предоставляемые `Rectangular`:

```
abstract class Component extends Rectangular {
  // другие методы...
}
```

Точно так же трейт может примешиваться в сам `Rectangle`:

```
class Rectangle(val topLeft: Point, val bottomRight: Point)
  extends Rectangular {

  // другие методы...
}
```

Располагая этими определениями, можно создать объект `Rectangle` и вызвать в отношении него геометрические методы `width` и `left`:

```
scala> val rect = new Rectangle(new Point(1, 1),
           new Point(10, 10))
rect: Rectangle = Rectangle@5f5da68c

scala> rect.left
res2: Int = 1

scala> rect.right
res3: Int = 10

scala> rect.width
res4: Int = 9
```

12.4. Трейт Ordered

Сравнение — еще одна область, в которой может пригодиться «толстый» интерфейс. Сравнивая два упорядочиваемых объекта, было бы удобно воспользоваться вызовом одного метода, чтобы выяснить результаты желаемого сравнения. Если нужно использовать сравнение «меньше», то предпочтительнее было бы вызвать `<`, а если требуется применить сравнение «меньше или равно» — то вызвать `<=`. С «тонким» интерфейсом можно располагать только методом `<`, и тогда временами приходилось бы создавать код наподобие `(x < y) || (x == y)`. «Толстый» интерфейс предоставит вам все привычные операторы сравнения, позволяя напрямую создавать код вроде `x <= y`.

Прежде чем посмотреть на трейт `Ordered`, представим, что можно сделать без него. Предположим, вы взяли класс `Rational` из главы 6 и добавили к нему операции сравнения. У вас должен получиться примерно такой код[1]:

```
class Rational(n: Int, d: Int) {
  // ...
  def < (that: Rational) =
    this.numer * that.denom < that.numer * this.denom
  def > (that: Rational) = that < this
  def <= (that: Rational) = (this < that) || (this == that)
  def >= (that: Rational) = (this > that) || (this == that)
}
```

В данном классе определяются четыре оператора сравнения (`<`, `>`, `<=` и `>=`), и это классическая демонстрация стоимости определения «толстого» интерфейса. Сначала обратите внимание на то, что три оператора сравнения определены на основе

[1] Этот пример основан на использовании класса Rational, показанного в листинге 6.5, с методами equals, hashCode и модификациями, гарантирующими положительный denom.

первого оператора. Например, оператор > определен как противоположность оператора <, а оператор <= определен буквально как «меньше или равно». Затем обратите внимание на то, что все три этих метода будут такими же для любого другого класса, объекты которого могут сравниваться друг с другом. В отношении оператора <= для рациональных чисел не прослеживается никаких особенностей. В контексте сравнения оператор <= *всегда* используется для обозначения «меньше или равно». В общем, в этом классе есть довольно много шаблонного кода, который будет точно таким же в любом другом классе, реализующем операции сравнения.

Данная проблема встречается настолько часто, что в Scala предоставляется трейт, помогающий справиться с ее решением. Этот трейт называется `Ordered`. Чтобы воспользоваться им, нужно заменить все отдельные методы сравнений одним методом `compare`. Затем на основе одного этого метода определить в трейте `Ordered` методы <, >, <= и >=. Таким образом, трейт `Ordered` позволит вам расширить класс методами сравнений с помощью реализации всего одного метода по имени `compare`.

Если определить операции сравнения в `Rational` путем использования трейта `Ordered`, то код будет иметь следующий вид:

```
class Rational(n: Int, d: Int) extends Ordered[Rational] {
  // ...
  def compare(that: Rational) =
    (this.numer * that.denom) - (that.numer * this.denom)
}
```

Нужно выполнить две задачи. Начнем с того, что в `Rational` примешивается трейт `Ordered`. В отличие от трейтов, которые встречались до сих пор, `Ordered` требует от вас при примешивании указать *параметр типа*. Параметры типов до главы 19 подробно рассматриваться не будут, а пока все, что нужно знать при примешивании `Ordered`, сводится к следующему: фактически следует выполнять примешивание `Ordered[`*C*`]`, где *C* обозначает класс, элементы которого сравниваются. В данном случае в `Rational` примешивается `Ordered[Rational]`.

Вторая задача, требующая выполнения, заключается в определении метода `compare` для сравнения двух объектов. Этот метод должен сравнивать получатель `this` с объектом, переданным методу в качестве аргумента. Возвращать он должен целочисленное значение, которое равно нулю, если объекты одинаковы, отрицательное число, если получатель меньше аргумента, и положительное, если получатель больше аргумента.

В данном случае метод сравнения класса `Rational` использует формулу, основанную на приведении чисел к общему знаменателю с последующим вычитанием получившихся числителей. Теперь при наличии этого примешивания и определения метода `compare` класс `Rational` имеет все четыре метода сравнения:

```
scala> val half = new Rational(1, 2)
half: Rational = 1/2

scala> val third = new Rational(1, 3)
third: Rational = 1/3
```

```
scala> half < third
res5: Boolean = false

scala> half > third
res6: Boolean = true
```

При каждой реализации класса с какой-либо возможностью упорядоченности путем сравнения нужно рассматривать вариант примешивания в него трейта `Ordered`. Если выполнить это примешивание, то пользователи класса получат богатый набор методов сравнений.

Имейте в виду, что трейт `Ordered` не определяет за вас метод `equals`, поскольку не способен на это. Дело в том, что реализация `equals` в терминах `compare` требует проверки типа переданного объекта, а из-за удаления типов сам `Ordered` не может ее выполнить. Поэтому определять `equals` вам придется самим, даже если вы примешиваете трейт `Ordered`. Как справиться с этой задачей, мы расскажем в главе 30.

12.5. Трейты как наращиваемые модификации

Основное применение трейтов — превращение «тонкого» интерфейса в «толстый» — вы уже видели. Перейдем теперь ко второму по значимости способу использования трейтов — предоставлению классам наращиваемых модификаций. Трейты дают возможность *изменять* методы класса, позволяя вам *наращивать* их друг на друге.

Рассмотрим в качестве примера наращиваемые модификации применительно к очереди целых чисел. В очереди будут две операции: `put`, помещающая целые числа в очередь, и `get`, извлекающая их из очереди. Очереди работают по принципу «первым пришел, первым ушел», поэтому целые числа извлекаются из очереди в том же порядке, в котором были туда помещены.

Располагая классом, реализующим такую очередь, можно определить трейты для выполнения следующих модификаций:

- удваивание — удваиваются все целые числа, помещенные в очередь;
- увеличение на единицу — увеличиваются все целые числа, помещенные в очередь;
- фильтрация — из очереди отфильтровываются все отрицательные целые числа.

Эти три трейта представляют *модификации*, поскольку модифицируют поведение соответствующего класса очереди, а не определяют полный класс очереди. Все три трейта также являются *наращиваемыми*. Можно выбрать любые из трех трейтов, примешать их в класс и получить новый класс, обладающий всеми выбранными модификациями.

В листинге 12.6 показан абстрактный класс `IntQueue`. В `IntQueue` имеются метод `put`, добавляющий к очереди новые целые числа, и метод `get`, возвращающий

целые числа и удаляющий их из очереди. Основная реализация `IntQueue`, которая использует `ArrayBuffer`, показана в листинге 12.7.

Листинг 12.6. Абстрактный класс IntQueue

```
abstract class IntQueue {
  def get(): Int
  def put(x: Int): Unit
}
```

Листинг 12.7. Реализация класса BasicIntQueue с использованием ArrayBuffer

```
import scala.collection.mutable.ArrayBuffer

class BasicIntQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) = { buf += x }
}
```

В классе `BasicIntQueue` имеется приватное поле, содержащее буфер в виде массива. Метод `get` удаляет запись с одного конца буфера, а метод `put` добавляет элементы к другому его концу. Пример использования данной реализации выглядит следующим образом:

```
scala> val queue = new BasicIntQueue
queue: BasicIntQueue = BasicIntQueue@23164256

scala> queue.put(10)

scala> queue.put(20)

scala> queue.get()
res9: Int = 10

scala> queue.get()
res10: Int = 20
```

Пока все вроде бы в порядке. Теперь посмотрим на использование трейтов для модификации данного поведения. В листинге 12.8 показан трейт, который удваивает целые числа по мере их помещения в очередь. Трейт `Doubling` имеет две интересные особенности. Первая заключается в том, что в нем объявляется суперкласс `IntQueue`. Это объявление означает, что трейт может примешиваться только в класс, который также расширяет `IntQueue`. То есть `Doubling` можно примешивать в `BasicIntQueue`, но не в `Rational`.

Листинг 12.8. Трейт наращиваемых модификаций Doubling

```
trait Doubling extends IntQueue {
  abstract override def put(x: Int) = { super.put(2 * x) }
}
```

Вторая интересная особенность заключается в том, что у трейта имеется вызов `super` в отношении метода, объявленного абстрактным. Для обычных классов такие вызовы применять запрещено, поскольку во время выполнения они гарантированно дадут сбой. Но для трейта такой вызов может действительно пройти успешно. В трейте вызовы `super` динамически связаны, поэтому вызов `super` в трейте `Doubling` будет работать при условии, что трейт примешан *после* другого трейта или класса, в котором дается конкретное определение метода.

Трейтам, которые реализуют наращиваемые модификации, зачастую нужен именно такой порядок. Чтобы сообщить компилятору, что это делается намеренно, подобные методы следует помечать модификаторами `abstract override`. Это сочетание модификаторов позволительно только для членов трейтов, но не классов, и означает, что трейт должен быть примешан в некий класс, имеющий конкретное определение рассматриваемого метода.

Вроде бы простой трейт, а сколько всего в нем происходит! Применение трейта выглядит следующим образом:

```
scala> class MyQueue extends BasicIntQueue with Doubling
defined class MyQueue

scala> val queue = new MyQueue
queue: MyQueue = MyQueue@44bbf788

scala> queue.put(10)

scala> queue.get()
res12: Int = 20
```

В первой строке этого сеанса работы с интерпретатором определяется класс `MyQueue`, который расширяет класс `BasicIntQueueand`, примешивая в него трейт `Doubling`. Мы помещаем в очередь число 10, но в результате примешивания трейта `Doubling` оно удваивается. При извлечении целого числа из очереди оно уже имеет значение 20.

Обратите внимание: в `MyQueue` не определяется никакой новый код — просто объявляется класс и примешивается трейт. В подобной ситуации вместо определения именованного класса код `BasicIntQueue with Doubling` может быть предоставлен непосредственно с ключевым словом `new`. Работа такого кода показана в листинге 12.9.

Листинг 12.9. Примешивание трейта при создании экземпляра с помощью ключевого слова new

```
scala> val queue = new BasicIntQueue with Doubling
queue: BasicIntQueue with Doubling = $anon$1@141f05bf

scala> queue.put(10)

scala> queue.get()
res14: Int = 20
```

Чтобы посмотреть, как нарастить модификации, нужно определить еще два модифицирующих трейта: `Incrementing` и `Filtering`. Реализация этих трейтов показана в листинге 12.10.

Листинг 12.10. Трейты наращиваемых модификаций Incrementing и Filtering

```
trait Incrementing extends IntQueue {
  abstract override def put(x: Int) = { super.put(x + 1) }
}
trait Filtering extends IntQueue {
  abstract override def put(x: Int) = {
    if (x >= 0) super.put(x)
  }
}
```

Теперь, располагая модифицирующими трейтами, можно выбрать, какой из них вам понадобится для той или иной очереди. Например, ниже показана очередь, в которой не только отфильтровываются отрицательные числа, но и ко всем сохраняемым числам прибавляется единица:

```
scala> val queue = (new BasicIntQueue
           with Incrementing with Filtering)
queue: BasicIntQueue with Incrementing with Filtering...

scala> queue.put(-1); queue.put(0); queue.put(1)

scala> queue.get()
res16: Int = 1

scala> queue.get()
res17: Int = 2
```

Порядок примешивания играет существенную роль[1]. Конкретные правила даны в следующем разделе, но, грубо говоря, трейт, находящийся правее, вступает в силу первым. Когда метод вызывается в отношении экземпляра класса с примешанными трейтами, первым вызывается тот метод, который определен в самом правом трейте. Если этот метод выполняет вызов `super`, то вызывается метод, который определен в следующем трейте левее данного трейта, и т. д. В предыдущем примере сначала вызывается метод `put` трейта `Filtering`, следовательно, все начинается с того, что он удаляет отрицательные целые числа. Вторым вызывается метод `put` трейта `Incrementing`, следовательно, к оставшимся целым числам прибавляется единица.

Если расположить трейты в обратном порядке, то сначала к целым числам будет прибавляться единица и только *потом* те целые числа, которые все же останутся отрицательными, будут удалены:

```
scala> val queue = (new BasicIntQueue
           with Filtering with Incrementing)
queue: BasicIntQueue with Filtering with Incrementing...
```

[1] Поскольку трейт примешивается к классу, его можно называть также примесью.

```
scala> queue.put(-1); queue.put(0); queue.put(1)

scala> queue.get()
res19: Int = 0

scala> queue.get()
res20: Int = 1

scala> queue.get()
res21: Int = 2
```

В общем, код, создаваемый в данном стиле, открывает перед вами широкие возможности для проявления гибкости. Примешивая эти три трейта в разных сочетаниях и разном порядке следования, можно определить 16 различных классов. Весьма впечатляющая гибкость для столь незначительного объема кода, поэтому постарайтесь не проглядеть возможности организации кода в целях получения наращиваемых модификаций.

12.6. Почему не используется множественное наследование

Трейты позволяют наследовать из множества похожих на классы конструкций, но имеют весьма важные отличия от множественного наследования, присутствующего во многих языках программирования. Одно из отличий, интерпретация `super`, играет особенно важную роль. При использовании множественного наследования метод, вызванный с помощью `super`, может быть определен прямо там, где появляется этот вызов. При использовании трейтов вызываемый метод определяется путем *линеаризации*, то есть выстраивания в ряд классов и трейтов, примешанных в класс. Это то самое рассмотренное в предыдущем разделе отличие, которое позволяет выполнять наращивание модификаций.

Прежде чем рассмотреть линеаризацию, немного отвлечемся на то, как наращиваемые модификации выполняются в языке с традиционным множественным наследованием. Представим следующий код, однако на этот раз интерпретируемый не как примешивание трейтов, а как множественное наследование:

```
// Мысленный эксперимент с множественным наследованием
val q = new BasicIntQueue with Incrementing with Doubling
q.put(42) // Который из методов put будет вызван?
```

Сразу возникает вопрос: который из методов `put` будет задействован в этом вызове? Возможно, вступят в силу правила, согласно которым победу одержит самый последний суперкласс. В таком случае будет вызван метод из `Doubling`. В данном методе будет удвоен его аргумент, сделан вызов `super.put`, и на этом все. Не произойдет никакого увеличения на единицу! Кроме того, если бы действовало правило, при котором побеждал бы первый суперкласс, то в получающейся очереди

целые числа увеличивались бы на единицу, но не удваивались. То есть не срабатывало бы никакое упорядочение.

Можно подумать и о том, как предоставить программистам возможность точно указывать при использовании вызова `super`, из какого именно суперкласса им нужен метод. Представим, к примеру, что есть следующий Scala-подобный код, в котором в `super` фигурирует точное указание на то, что метод вызывается как из `Incrementing`, так и из `Doubling`:

```
// Мысленный эксперимент с множественным наследованием
trait MyQueue extends BasicIntQueue
    with Incrementing with Doubling {

  def put(x: Int) = {
    Incrementing.super.put(x) // (на самом деле это не Scala)
    Doubling.super.put(x)
  }
}
```

Такой подход породит новые проблемы, самой малой из которых будет многословие. Получится, что метод `put` базового класса будет вызван *дважды*: один раз со значением, увеличенным на единицу, и один раз с удвоенным значением, — но никогда не будет вызван с увеличенным на единицу и удвоенным значением.

При использовании множественного наследования данная задача просто не имеет правильного решения. Придется опять возвращаться к проектированию и реорганизовывать код. В отличие от этого, с решением на основе применения трейтов в Scala все предельно понятно. Вы просто примешиваете трейты `Incrementing` и `Doubling`, и имеющееся в Scala особое правило, которое касается применения `super` в трейтах, позволяет добиться всего, чего вы хотели. Нечто здесь очевидно отличается от традиционного множественного наследования, но что именно?

Согласно уже данной подсказке ответом будет линеаризация. Когда с помощью ключевого слова `new` создается экземпляр класса, Scala берет класс со всеми его унаследованными классами и трейтами и располагает их в едином *линейном порядке*. Затем при любом вызове `super` внутри одного из таких классов вызывается тот метод, который идет следующим по порядку. Если во всех методах, кроме последнего, присутствует вызов `super`, то получается наращивание.

Описание конкретного порядка линеаризации дается в спецификации языка. Он сложноват, но вам нужно знать лишь главное: при любой линеаризации класс всегда следует впереди *всех* своих суперклассов и примешанных трейтов. Таким образом, при написании метода, содержащего вызов `super`, этот метод изменяет поведение суперкласса и примешанных трейтов, а не наоборот.

ПРИМЕЧАНИЕ

Далее в разделе описываются подробности линеаризации. Вы можете пропустить этот материал, если сейчас он вам неинтересен.

Главные свойства выполняющейся в Scala линеаризации показаны в следующем примере. Предположим, у вас есть класс `Cat` (Кот) — наследник класса `Animal` (Животное) — и два супертрейта: `Furry` (Пушистый) и `FourLegged` (Четырехлапый). Трейт `FourLegged` расширяет еще один трейт — `HasLegs` (С лапами):

```
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with FourLegged
```

Иерархия наследования и линеаризация класса `Cat` показаны на рис. 12.1. Наследование изображено с помощью традиционной нотации UML[1]: стрелки, на концах которых белые треугольники, обозначают наследование, указывая на супертип. Стрелки с затемненными не треугольными концами показывают линеаризацию. Они указывают направление, в котором будут разрешаться вызовы `super`.

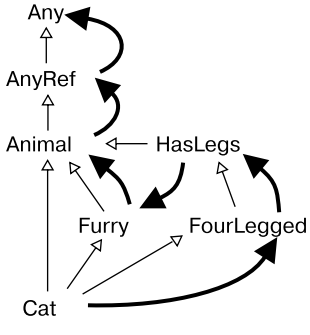


Рис. 12.1. Иерархия наследования и линеаризации класса Cat

Линеаризация `Cat` вычисляется от конца к началу следующим образом. Последней часть линеаризации `Cat` — линеаризация его суперкласса `Animal`. Эта линеаризация копируется без каких-либо изменений. (Линеаризация каждого из этих типов показана в табл. 12.1.) Поскольку класс `Animal` не расширяет явным образом какой-либо суперкласс и в него не примешаны никакие супертрейты, то по умолчанию он расширяет класс `AnyRef`, который расширяет класс `Any`. Поэтому линеаризация класса `Animal` имеет следующий вид:

`Animal → AnyRef → Any`

Предпоследней является линеаризация первой примеси, трейта `Furry`, но все классы, которые уже присутствуют в линеаризации класса `Animal`, теперь не учитываются, поэтому каждый класс в линеаризации `Cat` появляется только раз. Результат выглядит следующим образом:

`Furry → Animal → AnyRef → Any`

---

[1] *Rumbaugh, J., Jacobson I., Booch G.* The Unified Modeling Language Reference Manual. 2nd Ed. — Addison-Wesley, 2004.

Всему этому предшествует линеаризация `FourLegged`, в которой также не учитываются любые классы, которые уже были скопированы в линеаризациях суперкласса или первого примешанного трейта:

```
FourLegged → HasLegs → Furry → Animal → AnyRef → Any
```

И наконец, первым в линеаризации класса `Cat` фигурирует сам этот класс:

```
Cat → FourLegged → HasLegs → Furry → Animal → AnyRef → Any
```

Когда любой из этих классов и трейтов вызывает метод через вызов `super`, вызываться будет первая реализация, которая в линеаризации расположена справа от него.

Таблица 12.1. Линеаризация типов в иерархии класса Cat

Тип	Линеаризация
Animal	Animal, AnyRef, Any
Furry	Furry, Animal, AnyRef, Any
FourLegged	FourLegged, HasLegs, Animal, AnyRef, Any
HasLegs	HasLegs, Animal, AnyRef, Any
Cat	Cat, FourLegged, HasLegs, Furry, Animal, AnyRef, Any

12.7. Так все же с трейтами или без?

При реализации набора повторно используемого поведения придется решать, к чему прибегнуть: трейту или абстрактному классу. Золотого правила не существует, но в этом разделе содержатся несколько полезных рекомендаций, к которым стоит прислушаться.

- **Если поведение не будет повторно использовано**, то создавайте конкретный класс, поскольку в нем вообще нет повторно используемого поведения.
- **Если поведение может быть повторно использовано различными неродственными классами**, то создайте трейт. Только трейты могут примешиваться в различные части иерархии классов.
- **Если хотите наследовать поведение в коде Java**, то задействуйте абстрактный класс. В Java не существует близкого аналога трейтам, поэтому унаследовать что-либо из трейта в классе Java вряд ли получится. А вот наследование из класса Scala ничем не отличается от наследования из класса Java. В качестве единственного исключения следует отметить, что трейт Scala, имеющий исключительно абстрактные элементы, преобразуется непосредственно в интерфейс Java. Поэтому можно спокойно определять такие трейты, даже притом, что наследование из них предполагается использовать в коде Java. Более подробную информацию о совместной работе кода на Java с кодом на Scala можно найти в главе 31.

- **Если планируется распространение кода в скомпилированном виде** и ожидается, что сторонние группы разработчиков станут создавать классы, что-либо наследующие из него, то следует предпочесть абстрактные классы. Проблема в том, что при появлении в трейте нового элемента или удалении из него старого элемента все классы, являющиеся его наследниками, должны быть перекомпилированы, даже если в них не было никаких изменений. Если сторонние клиенты будут только вызывать поведение, а не наследовать его, то с использованием трейта не будет никаких проблем.
- **Если вы не пришли к решению** даже после того, как взвесили все ранее предоставленные рекомендации, то начните с создания трейта. Позже вы всегда сможете его изменить, а в целом использование трейта предоставляет вам больше возможностей.

Резюме

В этой главе мы показали работу трейтов и порядок их применения в нескольких часто встречающихся идиомах. Вы увидели, что трейты похожи на множественное наследование. Но благодаря тому, что в трейтах вызовы `super` интерпретируются с помощью линеаризации, удается не только избавиться от некоторых трудностей традиционного множественного наследования, но и воспользоваться наращиванием модификаций поведения программы. Вдобавок мы рассмотрели трейт `Ordered` и изучили порядок создания собственных расширяющих трейтов.

А теперь, усвоив все эти аспекты, нам предстоит вернуться немного назад и посмотреть на трейты в целом с другого ракурса. Трейты не просто поддерживают средства выражения, рассмотренные в данной главе, — они являются базовыми блоками кода, допускающими их повторное использование с помощью механизма наследования. Благодаря этому многие опытные программисты, которые работают на Scala, на ранних стадиях реализации начинают с трейтов. Любой трейт не может охватить всю концепцию, ограничиваясь лишь ее фрагментом. По мере того как конструкция приобретает все более четкие очертания, фрагменты путем примешивания трейтов могут объединяться в более полноценные концепции.

13 Пакеты и импорты

В ходе работы над программой, особенно объемной, важно свести к минимуму *сцепление*, то есть степень взаимозависимости различных частей программы. Низкое сцепление (low coupling) сокращает риск того, что незначительное, казалось бы, безобидное изменение в одной части программы вызовет разрушительные последствия для другой части. Один из способов сведения сцепления к минимуму — написание программы в модульном стиле. Программа разбивается на несколько меньших по размеру модулей, у каждого из которых есть внутренняя и внешняя части. При работе над внутренней частью модуля, то есть над его *реализацией*, нужно согласовывать действия только с другими программистами, работающими над созданием того же самого модуля. И лишь в случае необходимости внести изменения в его внешнюю часть, то есть в *интерфейс*, приходится координировать свои действия с разработчиками других модулей.

В этой главе мы покажем конструкции, помогающие программировать в модульном стиле. Мы рассмотрим вопросы помещения кода в пакеты, создания имен, видимых при импортировании, и управления видимостью определений с помощью модификаторов доступа. Эти конструкции сходны по духу с конструкциями, имеющимися в Java, за исключением некоторых различий, которые, как правило, выражаются в их более последовательном характере. Поэтому данную главу стоит прочитать даже тем, кто хорошо разбирается в Java.

13.1. Помещение кода в пакеты

Код Scala размещается в глобальной иерархии пакетов платформы Java. Все показанные до сих пор в этой книге примеры кода размещались в *безымянных* пакетах. Поместить код в именованные пакеты в Scala можно двумя способами. Первый — поместить содержимое всего файла в пакет, указав директиву `package` в самом начале файла (листинг 13.1).

Листинг 13.1. Помещение в пакет всего содержимого файла

```
package bobsrockets.navigation
class Navigator
```

Указание директивы `package` в листинге 13.1 приводит к тому, что класс `Navigator` помещается в пакет по имени `bobsrockets.navigation`. По-видимому, это программные средства навигации, разработанные корпорацией Bob's Rockets.

ПРИМЕЧАНИЕ

Поскольку код Scala — часть экосистемы Java, то тем пакетам Scala, которые выпускаются открытыми, рекомендуется соответствовать принятому в Java соглашению по присваиванию имени с обратным порядком следования доменных имен. Поэтому наиболее подходящим именем пакета для класса Navigator может быть com.bobsrockets.navigation. Но в данной главе мы отбросим com., чтобы было легче разобраться в примерах.

Другой способ, позволяющий в Scala помещать код в пакеты, больше похож на использование пространств имен C#. За директивой `package` следует раздел в фигурных скобках, в котором содержатся определения, помещаемые в пакет. Такой синтаксис называется *пакетированием*. Показанное в листинге 13.2 пакетирование имеет тот же эффект, что и код в листинге 13.1 (см. выше).

Листинг 13.2. Длинная форма простого объявления пакетирования

```
package bobsrockets.navigation {
  class Navigator
}
```

Кроме того, для таких простых примеров можно воспользоваться «синтаксическим сахаром», показанным в листинге 13.1 (см. выше). Но лучше все же реализовать один из вариантов более универсальной системы записи, позволяющей разместить разные части файла в разных пакетах. Например, в тот же самый файл с исходным кодом можно включить тесты классов, но, как показано в листинге 13.3, поместить тесты в другой пакет.

Листинг 13.3. Несколько пакетов в одном и том же файле

```
package bobsrockets {
  package navigation {

    // в пакете bobsrockets.navigation
    class Navigator

    package tests {

      // в пакете bobsrockets.navigation.tests
      class NavigatorSuite
    }
  }
}
```

13.2. Краткая форма доступа к родственному коду

Представление кода в виде иерархии пакетов не только помогает просматривать код, но и сообщает компилятору, что части кода в одном и том же пакете как-то связаны между собой. В Scala эта родственность позволяет при доступе к коду, находящемуся в одном и том же пакете, применять краткие имена.

В листинге 13.4 приведены три примера. В первом, согласно ожиданиям, к классу можно обращаться из его собственного пакета, не указывая префикс. Именно поэтому `new StarMap` проходит компиляцию. Класс `StarMap` находится в том же самом пакете по имени `bobsrockets.navigation`, что и выражение `new`, которое к нему обращается, поэтому указывать префикс в виде имени пакета не нужно.

Листинг 13.4. Краткая форма обращения к классам и пакетам

```
package bobsrockets {
  package navigation {
    class Navigator {
      // Указывать bobsrockets.navigation.StarMap не нужно
      val map = new StarMap
    }
    class StarMap
  }
  class Ship {
    // Указывать bobsrockets.navigation.Navigator не нужно
    val nav = new navigation.Navigator
  }
  package fleets {
    class Fleet {
      // Указывать bobsrockets.Ship не нужно
      def addShip() = { new Ship }
    }
  }
}
```

Во втором примере обращение к самому пакету может производиться из того же пакета, в котором он находится, без указания префикса. В листинге 13.4 показано, как создается экземпляр класса `Navigator`. Выражение `new` появляется в пакете `bobsrockets`, который содержится в пакете `bobsrockets.navigation`. Поэтому обращение к последнему можно указывать просто как `navigation`.

В третьем примере показано, что при использовании синтаксиса пакетирования с применением фигурных скобок все имена, доступные в пространстве имен вне пакета, доступны также и внутри него. Это обстоятельство позволяет в листинге 13.4 в `addShip()` создать новый экземпляр класса, воспользовавшись выражением `new Ship`. Метод определен внутри двух пакетов: внешнего `bobsrockets` и внутреннего `bobsrockets.fleets`. Поскольку к объекту `Ship` доступ можно получить во внешнем пакете, то из `addShip()` можно воспользоваться ссылкой на него.

Следует заметить, что такая форма доступа применима только в том случае, если пакеты вложены друг в друга явным образом. Если в каждый файл будет помещаться лишь один пакет, то, как и в Java, доступны будут только те имена, которые

определены в текущем пакете. В листинге 13.5 пакет `bobsrockets.fleets` был перемещен на самый верхний уровень. Он больше не заключен в пакет `bobsrockets`, поэтому имена из `bobsrockets` в его пространстве имен отсутствуют. В результате использование выражения `new Ship` вызовет ошибку компиляции.

Листинг 13.5. Обозначения, заключенные в пакеты, недоступны автоматически

```
package bobsrockets {
  class Ship
}
package bobsrockets.fleets {
  class Fleet {
    // Не пройдет компиляцию! Ship вне области видимости.
    def addShip() = { new Ship }
  }
}
```

Если обозначение вложенности пакетов с использованием фигурных скобок приводит к неудобному для вас сдвигу кода вправо, то можно воспользоваться несколькими указаниями директивы `package` без фигурных скобок[1]. Например, в показанном далее коде класс `Fleet` определяется в двух вложенных пакетах `bobsrockets` и `fleets` точно так же, как это было сделано в листинге 13.4 (см. выше):

```
package bobsrockets
package fleets
class Fleet {
  // Указывать bobsrockets.Ship не нужно
  def addShip() = { new Ship }
}
```

Важно знать еще об одной, последней особенности. Иногда в области видимости оказывается слишком много всего и имена пакетов скрывают друг друга. В листинге 13.6 пространство имен класса `MissionControl` включает три отдельных пакета по имени `launch`! Один пакет `launch` находится в `bobsrockets.navigation`, один — в `bobsrockets` и еще один — на верхнем уровне. А как тогда ссылаться на `Booster1`, `Booster2` и `Booster3`?

Листинг 13.6. Доступ к скрытым именам пакетов

```
// в файле launch.scala
package launch {
  class Booster3
}

// в файле bobsrockets.scala
package bobsrockets {
  package navigation {
    package launch {
      class Booster1
    }
    class MissionControl {
```

[1] Этот стиль, в котором используются несколько директив package без фигурных скобок, называется объявлением цепочки пакетов.

```
      val booster1 = new launch.Booster1
      val booster2 = new bobsrockets.launch.Booster2
      val booster3 = new _root_.launch.Booster3
    }
  }
  package launch {
    class Booster2
  }
}
```

Проще всего обратиться к первому из них. Ссылка на само имя `launch` приведет вас к пакету `bobsrockets.navigation.launch`, поскольку этот пакет `launch` определен в ближайшей области видимости. Поэтому к первому из `booster`-классов можно обратиться просто как к `launch.Booster1`. Ссылка на второй подобный класс также указывается без каких-либо особенных приемов. Можно указать `bobsrockets.launch.Booster2` и не оставить ни малейших сомнений о том, к какому из трех классов происходит обращение. Открытым остается лишь вопрос по поводу третьего класса `booster`: как обратиться к `Booster3` при условии, что пакет на самом верхнем уровне перекрывается вложенными пакетами?

Чтобы помочь справиться с подобной ситуацией, Scala предоставляет имя пакета `_root_`, являющегося внешним по отношению к любым другим создаваемым пользователем пакетам. Иначе говоря, каждый пакет верхнего уровня, который может быть создан, рассматривается как члены пакета `_root_`. Например, и `launch`, и `bobsrockets` в листинге 13.6 (см. выше) являются членами пакета `_root_`. В результате этого `_root_.launch` позволяет обратиться к пакету `launch` самого верхнего уровня, а `_root_.launch.Booster3` обозначает внешний класс `booster`.

13.3. Импортирование кода

В Scala пакеты и их члены могут импортироваться с использованием директивы `import`. Затем ко всему, что было импортировано, можно получить доступ, указав простое имя, такое как `File`, не используя такое развернутое имя, как `java.io.File`. Рассмотрим, к примеру, код, показанный в листинге 13.7.

Листинг 13.7. Превосходные фрукты от Боба, готовые к импорту

```
package bobsdelights

abstract class Fruit(
  val name: String,
  val color: String
)

object Fruits {
  object Apple extends Fruit("apple", "red")
  object Orange extends Fruit("orange", "orange")
  object Pear extends Fruit("pear", "yellowish")
  val menu = List(Apple, Orange, Pear)
}
```

Указание директивы `import` делает члены пакета или объект доступными по их именам, исключая необходимость ставить перед ними префикс с именем пакета или объекта. Рассмотрим ряд простых примеров:

```
// простая форма доступа к Fruit
import bobsdelights.Fruit

// простая форма доступа ко всем членам bobsdelights
import bobsdelights._

// простая форма доступа ко всем членам Fruits
import bobsdelights.Fruits._
```

Первый пример относится к импортированию отдельно взятого Java-типа, а во втором показан Java-импорт *до востребования* (on-demand). Единственное отличие состоит в том, что импорт до востребования в Scala записывается с замыкающим знаком подчеркивания (`_`) вместо знака звездочки (`*`). (Поскольку `*` является в Scala допустимым идентификатором!) Третья из показанных директив `import` относится к Java-импорту статических полей класса.

Эти три директивы `import` дают представление о том, что можно делать с помощью импортирования, но в Scala импортирование носит более универсальный характер. В частности, оно может быть где угодно, а не только в начале компилируемого модуля. К тому же при импортировании можно ссылаться на произвольные значения. Например, возможен импорт, показанный в листинге 13.8.

Листинг 13.8. Импортирование членов обычного объекта (не одиночки)

```
def showFruit(fruit: Fruit) = {
  import fruit._
  println(name + "s are " + color)
}
```

Метод `showFruit` импортирует все члены его параметра `fruit`, относящегося к типу `Fruit`. Следующая инструкция `println` может непосредственно ссылаться на `name` и `color`. Эти две ссылки — эквиваленты ссылок `fruit.name` и `fruit.color`. Такой синтаксис пригодится, в частности, при использовании объектов в качестве модулей. Соответствующее описание будет дано в главе 29.

Гибкость импортирования в Scala

Директива `import` работает в Scala намного более гибко, чем в Java. Эта гибкость характеризуется тремя принципиальными отличиями. Импорт кода в Scala:

- может появляться где угодно;
- позволяет, помимо пакетов, ссылаться на объекты (одиночки или обычные);
- позволяет изменять имена или скрывать некоторые из импортированных членов.

Еще один из факторов гибкости импорта кода в Scala заключается в возможности импортировать пакеты как таковые, без их непакетированного наполнения.

Смысл в этом будет только в том случае, если предполагается, что в пакете заключены другие пакеты. Например, в листинге 13.9 импортируется пакет `java.util.regex`. Благодаря этому `regex` можно использовать с указанием его простого имени. Чтобы обратиться к объекту-одиночке `Pattern` из пакета `java.util.regex`, можно, как показано в данном листинге, просто воспользоваться идентификатором `regex.Pattern`.

Листинг 13.9. Импортирование имени пакета

```
import java.util.regex

class AStarB {
  // обращение к java.util.regex.Pattern
  val pat = regex.Pattern.compile("a*b")
}
```

При импортировании кода Scala позволяет также переименовывать или скрывать члены. Для этого *импорт заключается* в фигурные скобки с указанием перед получившимся в результате блоком того объекта, из которого импортируются его члены. Рассмотрим несколько примеров:

```
import Fruits.{Apple, Orange}
```

Здесь из объекта `Fruits` импортируются только его члены `Apple` и `Orange`.

```
import Fruits.{Apple => McIntosh, Orange}
```

Здесь из объекта `Fruits` импортируются два члена, `Apple` и `Orange`. Но объект `Apple` переименовывается в `McIntosh`, поэтому к нему можно обращаться либо как `Fruits.Apple`, либо как `McIntosh`. Директива переименования всегда имеет вид <*исходное_имя*> `=>` <*новое_имя*>.

```
import java.sql.{Date => SDate}
```

Здесь под именем `SDate` импортируется класс данных SQL, чтобы можно было в то же время импортировать обычный класс для работы с датами Java просто как `Date`.

```
import java.{sql => S}
```

Здесь под именем `S` импортируется пакет `java.sql`, чтобы можно было воспользоваться кодом вида `S.Date`.

```
import Fruits.{_}
```

Здесь импортируются все члены объекта `Fruits`. Это означает то же самое, что и `import Fruits._`.

```
import Fruits.{Apple => McIntosh, _}
```

Здесь импортируются все члены объекта `Fruits`, но `Apple` переименовывается в `McIntosh`.

```
import Fruits.{Pear => _, _}
```

Здесь импортируются все члены объекта `Fruits`, *за исключением* `Pear`. Директива вида <*исходное_имя*> `=> _` исключает <*исходное_имя*> из импортируемых имен.

В определенном смысле переименование чего-либо в _ говорит о полном сокрытии переименованного члена. Это помогает избегать неоднозначностей. Предположим, имеется два пакета, `Fruits` и `Notebooks`, и в каждом из них определен класс `Apple`. Если нужно получить только ноутбук под названием Apple, а не фрукт, то можно воспользоваться двумя импортами по запросу:

```
import Notebooks._
import Fruits.{Apple => _, _}
```

Будут импортированы все члены `Notebooks` и все члены `Fruits`, за исключением `Apple`.

Эти примеры демонстрируют поразительную гибкость, которую предлагает Scala в вопросах избирательного импортирования членов, возможно, даже под другими именами. Таким образом, директива `import` может состоять из следующих селекторов:

- простого имени *x*, которое включается в набор импортируемых имен;
- директивы переименования *x* `=>` *y*. Член по имени *x* будет виден под именем *y*;
- директивы сокрытия *x* `=> _`. Имя *x* исключается из набора импортируемых имен;
- элемента *«поймать все»* (catch-all) `_`. Импортируются все члены, за исключением тех, которые были упомянуты в предыдущей директиве. Если указан элемент «поймать все», то в списке селекторов импортирования он должен стоять последним.

Самые простые `import`-директивы, показанные в начале данного раздела, могут рассматриваться как специальные сокращения директив `import` с селекторами. Например, `import p._` — эквивалент `import p.{_}`, а `import p.n` — эквивалент `import p.{n}`.

13.4. Неявное импортирование

Scala неявно добавляет импортируемый код в каждую программу. По сути, происходит то, что произошло бы при добавлении в самое начало каждого исходного файла с расширением `.scala` следующих трех директив `import`:

```
import java.lang._ // все из пакета java.lang
import scala._     // все из пакета scala
import Predef._    // все из пакета Predef
```

В пакете `java.lang` содержатся стандартные классы Java. Он всегда неявно импортируется в исходные файлы Scala[1]. Неявное импортирование `java.lang` позволяет вам, например, использовать вместо `java.lang.Thread` просто идентификатор `Thread`.

[1] Изначально имелась также реализация Scala на платформе .NET., где вместо этого импортировалось пространство имен System, .NET-аналог пакета java.lang.

Теперь уже вряд ли приходится сомневаться в том, что в пакете `scala` находится стандартная библиотека Scala, в которой содержатся многие самые востребованные классы и объекты. Поскольку пакет `scala` импортируется неявно, то можно, к примеру, вместо `scala.List` указать просто `List`.

В объекте `Predef` содержится множество определений типов, методов и неявных преобразований, которые обычно используются в программах на Scala. Например, `Predef` импортируется неявно, поэтому можно вместо `Predef.assert` задействовать просто идентификатор `assert`.

Эти три директивы `import` трактуются особым образом, позволяющим тому импорту, который указан позже, перекрывать указанный ранее. К примеру, класс `StringBuilder` определен как в пакете `scala`, так и, начиная с версии Java 1.5, в пакете `java.lang`. Импорт `scala` перекрывает импорт `java.lang`, поэтому простое имя `StringBuilder` будет ссылаться на `scala.StringBuilder`, а не на `java.lang.StringBuilder`.

13.5. Модификаторы доступа

Члены пакетов, классов или объектов могут быть помечены модификаторами доступа `private` и `protected`. Они ограничивают доступ к членам, позволяя обращаться к ним только из определенных областей кода. Трактовка модификаторов доступа Scala примерно соответствует принятой в Java, но при этом имеет ряд весьма важных отличий, которые рассматриваются в данном разделе.

Приватные члены

Приватные члены трактуются в Scala точно так же, как и в Java. Член с пометкой `private` виден только внутри класса или объекта, в котором содержится его определение. В Scala это правило распространяется и на внутренние классы. Данная трактовка более последовательна, но отличается от принятой в Java. Рассмотрим пример, показанный в листинге 13.10.

Листинг 13.10. Отличие приватного доступа в Scala от такого же доступа в Java

```
class Outer {
  class Inner {
    private def f() = { println("f") }
    class InnerMost {
      f() // вполне допустимо
    }
  }
  (new Inner).f() // ошибка: доступ к f отсутствует
}
```

В Scala обращение `(new Inner).f()` недопустимо, поскольку приватное объявление `f` сделано в классе `Inner`, а попытка обращения делается не из данного класса. В отличие от этого первое обращение к `f` в классе `InnerMost` вполне допустимо,

поскольку содержится в теле класса `Inner`. В Java допустимы оба обращения, так как в данном языке разрешается обращение из внешнего класса к приватным членам его внутренних классов.

Защищенные члены

Доступ к защищенным членам в Scala также менее свободен, чем в Java. В Scala обратиться к защищенному члену можно только из подклассов того класса, в котором был определен этот член. В Java обращение возможно и из других классов того же самого пакета. В Scala есть еще один способ достижения того же самого эффекта[1], поэтому модификатор `protected` можно оставить без изменений. Защищенные виды доступа показаны в листинге 13.11.

Листинг 13.11. Отличие защищенного доступа в Scala от такого же доступа в Java

```
package p {
  class Super {
    protected def f() = { println("f") }
  }
  class Sub extends Super {
    f()
  }
  class Other {
    (new Super).f() // ошибка: доступ к f отсутствует
  }
}
```

В данном листинге обращение к `f` в классе `Sub` вполне допустимо, поскольку объявление `f` было сделано с модификатором `protected` в `Super`, а `Sub` — подкласс `Super`. В отличие от этого обращение к `f` в `Other` недопустимо, поскольку `Other` не является наследником `Super`. В Java последнее обращение все равно будет разрешено, так как `Other` находится в том же самом пакете, что и `Sub`.

Публичные члены

В Scala нет явного модификатора для публичных членов: любой член, не помеченный как `private` или `protected`, является публичным. К публичным членам можно обращаться откуда угодно.

Область защиты

Модификаторы доступа в Scala могут дополняться спецификаторами. Модификатор вида `private[`*X*`]` или `protected[`*X*`]` означает, что доступ закрыт или защищен вплоть до *X*, где *X* определяет некий внешний пакет, класс или объект-одиночку.

[1] Используя спецификаторы, рассматриваемые ниже, в подразделе «Область защиты».

Специфицированные модификаторы доступа позволяют весьма четко обозначить границы управления видимостью. В частности, они позволяют выразить понятия доступности, имеющиеся в Java, такие как приватность пакета, защищенность пакета или закрытость вплоть до самого внешнего класса, которые невозможно выразить напрямую с помощью простых модификаторов, используемых в Scala. Но, помимо этого, они позволяют выразить правила доступности, которые не могут быть выражены в Java.

В листинге 13.12 представлен пример с использованием множества спецификаторов доступа. Здесь класс `Navigator` помечен как `private[bobsrockets]`. Это значит, он имеет область видимости, охватывающую все классы и объекты, которые содержатся в пакете `bobsrockets`. В частности, доступ к `Navigator` разрешен в объекте `Vehicle`, поскольку `Vehicle` содержится в пакете `launch`, который, в свою очередь, содержится в пакете `bobsrockets`. В то же время весь код, находящийся за пределами пакета `bobsrockets`, не может получить доступ к классу `Navigator`.

Листинг 13.12. Придание гибкости областям защиты с помощью спецификаторов доступа

```
package bobsrockets

package navigation {
  private[bobsrockets] class Navigator {
    protected[navigation] def useStarChart() = {}
    class LegOfJourney {
      private[Navigator] val distance = 100
    }
    private[this] var speed = 200
  }
}
package launch {
  import navigation._
  object Vehicle {
    private[launch] val guide = new Navigator
  }
}
```

Этот прием особенно полезен при разработке крупных проектов, содержащих несколько пакетов. Он позволяет определять элементы, видимость которых распространяется на несколько подчиненных пакетов проекта, оставляя их невидимыми для клиентов, являющихся внешними по отношению к данному проекту. Применить подобный прием в Java не представляется возможным. Там, сразу после перехода границы своего пакета, определение становится видимым повсеместно.

Разумеется, действие спецификатора `private` может распространяться и на непосредственно окружающий пакет. В листинге 13.12 (см. выше) показан пример модификатора доступа `guide` в объекте `Vehicle`. Такой модификатор доступа эквивалентен имеющемуся в Java доступу, ограниченному пределами одного пакета.

Все спецификаторы также могут применяться к модификатору `protected` со значениями, аналогичными тем, с которыми они применяются к модификатору

`private`. То есть модификатор `protected[`*X*`]` в классе `C` позволяет получить доступ к определению с подобной пометкой во всех подклассах `C`, а также во внешнем пакете, классе или объекте с названием *X*. Например, метод `useStarChart` в приведенном выше листинге 13.12 доступен из всех подклассов `Navigator`, а также из всего кода, содержащегося во внешнем пакете `navigation`. В результате получается точное соответствие значению модификатора `protected` в Java.

Спецификаторы модификатора `private` могут также ссылаться на окружающий (внешний) класс или объект. Например, показанная в листинге 13.12 (см. выше) переменная `distance` в классе `LegOfJourney` имеет пометку `private[Navigator]`, следовательно, видима из любого места в классе `Navigator`. Тем самым ей придаются такие же возможности видимости, как и приватным членам внутренних классов в Java. Модификатор `private[`*C*`]`, где *C* — самый внешний класс, аналогичен простому модификатору `private` в Java.

И наконец, в Scala имеется также модификатор доступа, ограничивающий еще больше, чем `private`. Определение с пометкой `private[this]` доступно только внутри того же самого объекта, в котором содержится это определение. Такое определение называется *приватным внутри объекта* (object-private). Например, в листинге 13.12 (см. выше) приватным внутри объекта является определение `speed` в классе `Navigator`. Это значит, любое обращение должно быть сделано внутри не только класса `Navigator`, но и именно этого экземпляра класса `Navigator`. Таким образом, внутри класса `Navigator` будут вполне допустимы обращения `speed` и `this.speed`.

Но следующее обращение не будет позволено, даже если появится внутри класса `Navigator`:

```
val other = new Navigator
other.speed // эта строка кода не пройдет компиляцию
```

Элемент помечается модификатором `private[this]` с целью гарантировать, что он не будет виден из других объектов того же самого класса. Это может пригодиться для документирования. Иногда это также позволяет создавать более универсальные варианты аннотаций (подробности рассматриваются в разделе 19.7).

В качестве резюме в табл. 13.1 приведен список действий спецификаторов модификатора `private`. В каждой строке показан модификатор `private` со спецификатором и раскрыто его значение при применении в отношении переменной `distance`, объявленной в классе `LegOfJourney` в листинге 13.12 (см. выше).

Таблица 13.1. Действия спецификаторов private в отношении LegOfJourney.distance

Спецификатор	**Действие**
Без указания модификатора доступа	Открытый доступ
private[bobsrockets]	Доступ в пределах внешнего пакета
private[navigation]	Аналог имеющейся в Java видимости в пределах пакета
private[Navigator]	Аналог имеющегося в Java модификатора private
private[LegOfJourney]	Аналог имеющегося в Scala модификатора private
private[this]	Доступ только из того же самого объекта

Видимость и объекты-компаньоны

В Java статические члены и члены экземпляра принадлежат одному и тому же классу, поэтому модификаторы доступа применяются к ним одинаково. Вы уже видели, что в Scala статических членов нет, вместо них может быть объект-компаньон, содержащий члены, существующие в единственном экземпляре. Например, в листинге 13.13 объект `Rocket` — компаньон класса `Rocket`.

Листинг 13.13. Обращение к приватным членам класса- и объекта-компаньона

```
class Rocket {
  import Rocket.fuel
  private def canGoHomeAgain = fuel > 20
}

object Rocket {
  private def fuel = 10
  def chooseStrategy(rocket: Rocket) = {
    if (rocket.canGoHomeAgain)
      goHome()
    else
      pickAStar()
  }
  def goHome() = {}
  def pickAStar() = {}
}
```

Что касается приватного или защищенного доступа, то в правилах доступа, действующих в Scala, объектам- и классам-компаньонам даются особые привилегии. Класс делится всеми своими правами доступа со своим объектом-компаньоном, и *наоборот*. В частности, объект может обращаться ко всем приватным членам своего класса-компаньона точно так же, как класс может обращаться ко всем приватным членам своего объекта-компаньона.

Например, в показанном выше листинге 13.13 класс `Rocket` может обращаться к методу `fuel`, который объявлен приватным в объекте `Rocket`. Аналогично этому объект `Rocket` может обращаться к приватному методу `canGoHomeAgain` в классе `Rocket`.

Одно из исключений, которое нарушает аналогию между Scala и Java, касается защищенных статических членов. Защищенный статический член Java-класса `C` может быть доступен во всех подклассах `C`. В отличие от этого в наличии защищенного члена в объекте-компаньоне нет никакого смысла, поскольку у объектов-одиночек нет никаких подклассов.

13.6. Объекты пакетов

До сих пор единственным встречающимся вам кодом, добавляемым к пакетам, были классы, трейты и самостоятельные объекты. Они, несомненно, являются наиболее распространенными определениями, помещаемыми на самом верхнем уровне пакета. Но Scala не ограничивает вас только этим перечнем — любые виды

определений, которые можно помещать внутри класса, могут присутствовать и на верхнем уровне пакета. На самый верхний уровень пакета можно смело помещать любой вспомогательный метод, который хотелось бы иметь в области видимости всего пакета.

Для этого поместите определение в *объект пакета*. В каждом пакете разрешается иметь один объект пакета. Любые определения, помещенные в объекте пакета, считаются членами самого пакета.

Пример показан в листинге 13.14. В файле `package.scala` содержится объект пакета для пакета `bobsdelights`. Синтаксически объект пакета очень похож на одно из ранее показанных в данной главе пакетирований в фигурных скобках. Единственное отличие заключается в том, что в него включено ключевое слово `object`. Это *объект* пакета, а не сам *пакет*. Содержимое, заключенное в фигурные скобки, может включать любые нужные вам определения. В данном случае объект пакета включает вспомогательный метод `showFruit` из листинга 13.8 (см. выше).

Листинг 13.14. Объект пакета

```
// в файле bobsdelights/package.scala
package object bobsdelights {
  def showFruit(fruit: Fruit) = {
    import fruit._
    println(name + "s are " + color)
  }
}

// в файле PrintMenu.scala
package printmenu
import bobsdelights.Fruits
import bobsdelights.showFruit

object PrintMenu {
  def main(args: Array[String]) = {
    for (fruit <- Fruits.menu) {
      showFruit(fruit)
    }
  }
}
```

При наличии этого определения любой другой код в любом пакете может импортировать метод точно так же, как мог бы импортировать класс. Например, в данном листинге показан самостоятельный объект `PrintMenu`, который находится в другом пакете. `PrintMenu` может импортировать вспомогательный метод `showFruit` аналогично тому, как импортировал бы класс `Fruit`.

Забегая вперед, следует отметить, что есть и другие примеры использования объектов пакетов для тех разновидностей определений, которые еще не были вам показаны. Объекты пакетов часто используются для содержания псевдонимов типов, предназначенных для всего пакета (см. главу 20) и неявных преобразований (см. главу 21). В пакете верхнего уровня `scala` имеется объект пакета, чьи определения доступны всему коду Scala. Объекты пакетов компилируются в фай-

лы классов с именами `package.class`, размещаемыми в каталоге `package`, который они дополняют. Им будет полезно придерживаться того же соглашения, которое действует в отношении исходных файлов. Таким образом, исходный код объекта пакета `bobsdelights` из листинга 13.14 было бы разумно поместить в файл `package.scala`, размещаемый в каталоге `bobsdelights`.

Резюме

В данной главе мы показали основные конструкции, предназначенные для разбиения программы на пакеты. Благодаря этому вы имеете простую и полезную разновидность модульности и можете работать с весьма большими объемами кода, не допуская взимного влияния различных его частей. Существующая в Scala система по духу аналогична пакетированию, используемому в Java, но с некоторыми различиями: в Scala проявляется более последовательный или же более универсальный подход.

Забегая вперед: в главе 29 описана более гибкая система модулей, разбиваемая на пакеты. Данный подход позволяет разбивать код на несколько пространств имен и вдобавок дает модулям возможность выполнить взаимное наследование и параметризацию. В следующей главе мы переключимся на утверждения и модульное тестирование.

14 Утверждения и тесты

Утверждения и тесты — два важных способа проверки правильности поведения созданных вами программных средств. В данной главе мы покажем несколько вариантов для их создания и запуска, доступных в Scala.

14.1. Утверждения

Утверждения в Scala создаются в виде вызовов предопределенного метода `assert`[1]. Выражение `assert(condition)` выдает ошибку `AssertionError`, если *условие* `condition` не соблюдается. Существует также версия `assert`, использующая два аргумента: выражение `assert(condition, explanation)` тестирует *условие*. При его не соблюдении оно выдает ошибку `AssertionError`, в сообщении о которой содержится заданное *объяснение* `explanation`. Тип объяснения — `Any`, поэтому в качестве объяснения может передаваться любой объект. Чтобы поместить в `AssertionError` строковое объяснение, метод `assert` будет вызывать в отношении этого объекта метод `toString`. Например, в метод по имени `above` класса `Element` и показанный в листинге 10.13 (см. выше) `assert` можно поместить после вызовов `widen`, чтобы убедиться в одинаковой ширине расширенных элементов. Этот вариант показан в листинге 14.1.

Листинг 14.1. Использование утверждения

```
def above(that: Element): Element = {
  val this1 = this widen that.width
  val that1 = that widen this.width
  assert(this1.width == that1.width)
  elem(this1.contents ++ that1.contents)
}
```

Выполнить ту же задачу можно и по-другому: проверить ширину в конце метода `widen`, непосредственно перед возвращением значения. Этого можно добиться,

[1] Метод assert определен в объекте-одиночке Predef, элементы которого автоматически импортируются в каждый исходный файл программы на языке Scala.

сохранив результат в val-переменной, выполняя утверждение применительно к результату с последующим указанием val-переменной, чтобы результат возвращался в том случае, если утверждение было успешно подтверждено. Но, как показано в листинге 14.2, это можно сделать более выразительно, воспользовавшись довольно удобным методом ensuring из объекта-одиночки Predef.

Листинг 14.2. Использование метода ensuring для утверждения результата выполнения функции

```
private def widen(w: Int): Element =
  if (w <= width)
    this
  else {
    val left = elem(' ', (w - width) / 2, height)
    var right = elem(' ', w - width - left.width, height)
    left beside this beside right
  } ensuring (w <= _.width)
```

Благодаря неявному преобразованию метод ensuring может использоваться с любым результирующим типом. В данном коде все выглядит так, словно ensuring вызван в отношении результата выполнения метода widen, имеющего тип Element. Однако на самом деле ensuring вызван в отношении типа, в который Element был автоматически преобразован. Метод ensuring получает один аргумент, функцию-предикат, которая берет результирующий тип, возвращает булево значение и передает результат предикату. Если предикат возвращает true, то метод ensuring возвращает результат, в противном случае метод выдаст ошибку AssertionError.

В данном примере предикат имеет вид w <= _.width. Знак подчеркивания является заместителем для одного аргумента, который передается предикату, а именно результата типа Element, получаемого от метода widen. Если ширина, переданная в виде w методу widen, меньше ширины результата типа Element или равна ей, то предикат будет вычислен в true и результатом ensuring будет объект типа Element, в отношении которого он был вызван. Данное выражение в методе widen последнее, поэтому его результат типа Element и будет значением, возвращаемым самим методом widen.

Утверждения могут включаться и выключаться с помощью флагов командной строки JVM -ea и -da. Когда флаги включены, каждое утверждение служит небольшим тестом, использующим фактические данные, вычисленные при выполнении программы. Далее в главе мы сфокусируемся на написании внешних тестов, которые предоставляют собственные тестовые данные и выполняются независимо от приложения.

14.2. Тестирование в Scala

Scala содержит много опций для тестирования, начиная с хорошо известного инструментария Java, такого как JUnit и TestNG, и заканчивая инструментарием, написанным на Scala, например ScalaTest, specs2 и ScalaCheck. В оставшейся части главы мы дадим краткий обзор этого инструментария. Начнем со ScalaTest.

ScalaTest — наиболее гибкая среда тестирования в Scala, это средство можно легко настроить на решение различных задач. Гибкость ScalaTest означает, что команды могут использовать тот стиль тестирования, который более полно отвечает их потребностям. Например, для команд, знакомых с JUnit, наиболее знакомым и удобным станет стиль `AnyFunSuite`. Пример показан в листинге 14.3.

Листинг 14.3. Написание тестов с использованием AnyFunSuite

```
import org.scalatest.FunSuite
import Element.elem

class ElementSuite extends FunSuite {

  test("elem result should have passed width") {
    val ele = elem('x', 2, 3)
    assert(ele.width == 2)
  }
}
```

Центральная концепция в ScalaTest — *набор*, то есть коллекция тестов. *Тестом* может являться любой код с именем, который может быть запущен и завершен успешно или неуспешно, отложен или отменен. Центральный блок композиции в ScalaTest — трейт `Suite`. В нем объявляются методы жизненного цикла, определяющие исходный способ запуска тестов, который можно переопределить под нужные способы написания и запуска тестов.

В ScalaTest предлагаются *стилевые трейты*, расширяющие `Suite` и переопределяющие методы жизненного цикла для поддержания различных стилей тестирования. В этой среде также предоставляются *примешиваемые трейты*, которые переопределяют методы жизненного цикла таким образом, чтобы те отвечали конкретным потребностям тестирования. Классы тестов определяются сочетанием стиля `Suite` и примешиваемых трейтов, а тестовые наборы — путем составления экземпляров `Suite`.

Пример стиля тестирования — `AnyFunSuite`, расширенный тестовым классом, показанным в листинге 14.3 (см. выше). Слово Fun в `AnyFunSuite` означает функцию, а `test` является определенным в `AnyFunSuite` методом, который вызывается первичным конструктором `ElementSuite`. Вы указываете название теста в виде строки в круглых скобках, а сам код теста помещаете в фигурные скобки. Код теста является функцией, передаваемой методу `test` в виде параметра по имени, которая регистрирует его для последующего выполнения.

Среда ScalaTest интегрирована в широко используемые инструментальные средства сборки, такие как sbt и Maven, и в IDE, например IntelliJIDEA и Eclipse. `Suite` можно запустить и непосредственно через приложение ScalaTest Runner или из интерпретатора Scala, просто вызвав в отношении него метод `execute`. Соответствующий пример выглядит так:

```
scala> (new ElementSuite).execute()
ElementSuite:
- elem result should have passed width
```

Все стили ScalaTest, включая `AnyFunSuite`, предназначены для стимуляции написания специализированных тестов с осмысленными названиями. Кроме того, все стили создают вывод, похожий на спецификацию, которая может облегчить общение между заинтересованными сторонами. Выбранным вами стилем предписывается только то, как будут выглядеть объявления ваших тестов. Все остальное в ScalaTest работает одинаково, независимо от выбранного стиля[1].

14.3. Информативные отчеты об ошибках

В тесте, показанном в листинге 14.3 (см. выше), предпринимается попытка создать элемент шириной, равной 2, и высказывается утверждение, что ширина получившегося элемента действительно равна 2. Если утверждение не подтвердится, то отчет об ошибке будет включать имя файла и номер строки с неоправдавшимся утверждением, а также информативное сообщение об ошибке:

```
scala> val width = 3
width: Int = 3

scala> assert(width == 2)
org.scalatest.exceptions.TestFailedException:
    3 did not equal 2
```

Чтобы обеспечить содержательные сообщения об ошибках при неверных утверждениях, ScalaTest в ходе компиляции анализирует выражения, переданные каждому вызову утверждения. Если вы хотите увидеть более подробную информацию о неверных утверждениях, то можете воспользоваться имеющимся в ScalaTest средством Diagrams, сообщения об ошибках которого показывают схему выражения, переданного утверждению `assert`:

```
scala> assert(List(1, 2, 3).contains(4))
org.scalatest.exceptions.TestFailedException:

  assert(List(1, 2, 3).contains(4))
         |    |  |  |  |        |
         |    1  2  3  false    4
         List(1, 2, 3)
```

Имеющиеся в ScalaTest методы `assert` не делают разницы в сообщениях об ошибках между фактическим и ожидаемым результатами. Они просто показывают, что левый операнд не равен правому, или показывают значения на схеме. Если нужно подчеркнуть различия между фактическим и ожидаемым результатами, то можно воспользоваться имеющимся в ScalaTest альтернативным методом `assertResult`:

```
assertResult(2) {
  ele.width
}
```

[1] Более подробную информацию о ScalaTest можно найти на сайте www.scalatest.org.

С помощью данного выражения показывается, что от кода в фигурных скобках ожидается результат 2. Если в результате выполнения этого кода получится 3, то в отчете об ошибке тестирования будет показано сообщение `Expected 2, but got 3` (Ожидалось 2, но получено 3).

При необходимости проверить, выдает ли метод ожидаемое исключение, можно воспользоваться имеющимся в ScalaTest методом `assertThrows`:

```
assertThrows[IllegalArgumentException] {
  elem('x', -2, 3)
}
```

Если код в фигурных скобках выдает не то исключение, которое ожидалось, или вообще не выдает его, то метод `assertThrows` тут же завершит свою работу и выдаст исключение `TestFailedException`. А отчет об ошибке будет содержать сообщение с полезной для вас информацией:

```
Expected IllegalArgumentException to be thrown,
  but NegativeArraySizeException was thrown.
```

Но если код завершится внезапной генерацией экземпляра переданного класса исключения, то метод `assertThrows` выполнится нормально. При необходимости провести дальнейшее исследование ожидаемого исключения можно вместо `assertThrows` воспользоваться методом перехвата `intercept`. Он работает аналогично методу `asassertThrows`, за исключением того, что при генерации ожидаемого исключения `intercept` возвращает это исключение:

```
val caught =
  intercept[ArithmeticException] {
    1 / 0
  }

assert(caught.getMessage == "/ by zero")
```

Короче говоря, имеющиеся в ScalaTest утверждения стараются выдать полезные сообщения об ошибках, способные помочь вам диагностировать и устранить проблемы в коде.

14.4. Использование тестов в качестве спецификаций

В стиле тестирования при *разработке через реализацию поведения* (behavior-driven development, BDD) основной упор делается на написание легко воспринимаемых человеком спецификаций расширенного поведения кода и сопутствующих тестов, которые проверяют, свойственно ли коду такое поведение. В ScalaTest включены несколько трейтов, содействующих этому стилю тестирования. Пример использования одного такого трейта, `AnyFlatSpec`, показан в листинге 14.4.

Листинг 14.4. Спецификация и тестирование поведения с помощью AnyFlatSpec

```
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should.Matchers
import Element.elem

class ElementSpec extends AnyFlatSpec with Matchers {
  "A UniformElement" should
      "have a width equal to the passed value" in {
    val ele = elem('x', 2, 3)
    ele.width should be (2)
  }

  it should "have a height equal to the passed value" in {
    val ele = elem('x', 2, 3)
    ele.height should be (3)
  }

  it should "throw an IAE if passed a negative width" in {
    an [IllegalArgumentException] should be thrownBy {
      elem('x', -2, 3)
    }
  }
}
```

При использовании `AnyFlatSpec` тесты создаются в виде *директив спецификации*. Сначала в виде строки пишется название тестируемого *субъекта* (`"A UniformElement"` в листинге 14.4), затем `should`, или `must`, или `can` (что означает «обязан», или «должен», или «может» соответственно), потом строка, обозначающая характер поведения, требуемого от субъекта, а затем ключевое слово `in`. В фигурных скобках, стоящих после `in`, пишется код, тестирующий указанное поведение. В последующих директивах, чтобы сослаться на самый последний субъект, можно написать `it`. При выполнении `AnyFlatSpec` этот трейт будет запускать каждую директиву спецификации в виде теста ScalaTest. `AnyFlatSpec` (и другие специфицирующие трейты, которые есть в ScalaTest) генерирует вывод, который при запуске читается как спецификация. Например, вот на что будет похож вывод, если запустить `ElementSpec` из листинга 14.4 в интерпретаторе:

```
scala> (new ElementSpec).execute()
A UniformElement
- should have a width equal to the passed value
- should have a height equal to the passed value
- should throw an IAE if passed a negative width
```

В листинге 14.4 также показан имеющийся в ScalaTest предметно-ориентированный язык (domain-specific language, DSL) *выявления соответствий*. Примешиванием трейта `Matchers` можно создавать утверждения, которые при чтении больше похожи на естественный язык. Имеющийся в ScalaTest DSL-язык предоставляет множество средств выявления соответствий, кроме этого, позволяет определять новые предоставленные пользователем средства выявления соответствий, содержащие сообщения об ошибках. Такие средства, показанные в листинге 14.4, включают

синтаксис `should be` и `an [...] should be thrownBy { ...}`. Как вариант, если предпочтение отдается глаголу *must*, а не *should*, то можно примешать `MustMatchers`. Например, примешивание `MustMatchers` позволит вам создавать следующие выражения:

```
result must be >= 0
map must contain key 'c'
```

Если последнее утверждение не подтвердится, то будет показано сообщение об ошибке следующего вида:

```
Map('a' -> 1, 'b' -> 2) did not contain key 'c'
```

Среда тестирования specs2 — средство с открытым кодом, написанное на Scala Эриком Торребо (Eric Torreborre), — также поддерживает BDD-стиль тестирования, но с другим синтаксисом. Например, specs2 можно использовать для создания теста, показанного в листинге 14.5.

Листинг 14.5. Спецификация и тестирование поведения с использованием среды specs2

```
import org.specs2._
import Element.elem

object ElementSpecification extends Specification {
  "A UniformElement" should {
    "have a width equal to the passed value" in {
      val ele = elem('x', 2, 3)
      ele.width must be_==(2)
    }
    "have a height equal to the passed value" in {
      val ele = elem('x', 2, 3)
      ele.height must be_==(3)
    }
    "throw an IAE if passed a negative width" in {
      elem('x', -2, 3) must
        throwA[IllegalArgumentException]
    }
  }
}
```

В specs2, как и в ScalaTest, существует DSL-язык выявления соответствий. Некоторые примеры работы средств выявления соответствий, имеющихся в specs2, показаны в листинге 14.5 в строках, содержащих `must be_==` и `must throwA`[1]. Среду specs2 можно использовать в автономном режиме, но она также интегрируется со ScalaTest и JUnit, поэтому specs2-тесты можно запускать и с этими инструментами.

Одной из основных идей BDD является то, что тесты могут помогать обмениваться мнениями людям, принимающим решения о характере поведения программных средств, людям, разрабатывающим программные средства, и людям, определяющим степень завершенности и работоспособности программных средств. В этом ключе могут применяться любые стили, имеющиеся в ScalaTest или specs2,

[1] Программное средство specs2 можно загрузить с сайта specs2.org.

однако в ScalaTest есть специально разработанный для этого стиль `AnyFeatureSpec`. Пример его использования показан в листинге 14.6.

Листинг 14.6. Использование тестов для содействия обмену мнениями среди всех заинтересованных сторон

```
import org.scalatest._
import org.scalatest.featurespec.AnyFeatureSpec

class TVSetSpec extends AnyFeatureSpec with GivenWhenThen {

  Feature("TV power button") {
    Scenario("User presses power button when TV is off") {
      Given("a TV set that is switched off")
      When("the power button is pressed")
      Then("the TV should switch on")
      pending
    }
  }
}
```

Стиль `AnyFeatureSpec` разработан для того, чтобы направить в нужное русло обсуждений предназначение программных средств: вам следует выявить специфические *требования*, а затем дать им точное определение в виде *скриптов*. Сосредоточиться на переговорах об особенностях отдельно взятых скриптов помогают методы `Given`, `When` и `Then`, предоставляемые трейтом `GivenWhenThen`. Вызов `pending` в самом конце показывает: и тест, и само поведение не реализованы, имеется лишь спецификация. Как только будут реализованы все тесты и конкретные действия, тесты будут пройдены и требования можно будет посчитать выполненными.

14.5. Тестирование на основе свойств

Еще одним полезным средством тестирования для Scala является ScalaCheck — среда с открытым кодом, созданная Рикардом Нильсоном (Rickard Nilsson). Она позволяет указывать свойства, которыми должен обладать тестируемый код. Для каждого свойства ScalaCheck создает данные и выдает утверждения, проверяющие наличие тех или иных свойств. Пример использования ScalaCheck из ScalaTest `AnyWordSpec`, в который примешан трейт `ScalaCheckPropertyChecks`, показан в листинге 14.7.

Листинг 14.7. Тестирование на основе свойств с помощью ScalaCheck

```
import org.scalatest.wordspec.AnyWordSpec
import org.scalatestplus.scalacheck.ScalaCheckPropertyChecks
import org.scalatest.matchers.must.Matchers._
import Element.elem

class ElementSpec extends AnyWordSpec
        with ScalaCheckPropertyChecks {
  "elem result" must {
```

```
    "have passed width" in {
      forAll { (w: Int) =>
        whenever (w > 0) {
          elem('x', w % 100, 3).width must equal (w % 100)
        }
      }
    }
  }
}
```

`AnyWordSpec` — класс стиля, имеющийся в ScalaTest. Трейт `ScalaCheckPropertyChecks` предоставляет несколько методов `forAll`, позволяющих смешивать тесты на основе проверки наличия свойств с традиционными тестами на основе утверждений или выявления соответствий. В данном примере проверяется наличие свойства, которым должен обладать фабричный метод `elem`. Свойства ScalaCheck выражены в виде значений функций, получающих в качестве параметров данные, необходимые для утверждений о наличии свойств. Эти данные будут генерироваться ScalaCheck. В свойстве, показанном в листинге 14.7, данными является целое число `w`, которое представляет ширину. Внутри тела функции показан следующий код:

```
whenever (w > 0) {
  elem('x', w % 100, 3).width must equal (w % 100)
}
```

Директива `whenever` указывает на то, что при каждом вычислении левостороннего выражения в `true` правостороннее также должно быть `true`. Таким образом, в данном случае выражение в блоке должно быть `true` всякий раз, когда `w` больше нуля. А правостороннее будет выдавать `true`, если ширина, переданная фабричному методу `elem`, будет равна ширине объекта `Element`, возвращенного фабричным методом.

При таком небольшом объеме кода ScalaCheck сгенерирует несколько пробных значений, проверяя каждое из них в поисках значения, для которого свойство не соблюдается. Если свойство соблюдается для каждого значения, испытанного с помощью ScalaCheck, то тест будет пройден. В противном случае он будет завершен с генерацией исключения `TestFailedException`, которое содержит информацию, включающую значение, вызвавшее сбой.

14.6. Подготовка и проведение тестов

Во всех средах, упомянутых в текущей главе, имеется механизм для подготовки и проведения тестов. В данном разделе будет дан краткий обзор того подхода, который используется в ScalaTest. Чтобы получить подробное описание любой из этих сред, нужно обратиться к их документации.

Подготовка больших наборов тестов в ScalaTest проводится путем вложения `Suite`-наборов в `Suite`-наборы. При выполнении `Suite`-набор запустит не только свои тесты, но и все тесты вложенных в него `Suite`-наборов. Вложенные `Suite`-

наборы, в свою очередь, выполнят тесты вложенных в них `Suite`-наборов и т. д. Таким образом, большой набор тестов будет представлен деревом `Suite`-объектов. При выполнении в этом дереве корневого `Suite`-объекта будут выполнены все имеющиеся в нем `Suite`-объекты.

Наборы можно вкладывать вручную или автоматически. Чтобы вложить их вручную, вам нужно либо переопределить метод `nestedSuites` в своих `Suite`-классах, либо передать предназначенные для вложения `Suite`-объекты в конструктор класса `Suites`, который для этих целей предоставляет ScalaTest. Для автоматического вложения имена пакетов передаются имеющемуся в ScalaTest средству Runner, которое определит `Suite`-объекты автоматически, вложит их ниже корневого `Suite` и выполнит корневой `Suite`.

Имеющееся в ScalaTest приложение Runner можно вызвать из командной строки или через такие средства сборки, как sbt, maven или ant. Проще всего вызвать Runner в командной строке через приложение `theorg.scalatest.run`. Оно ожидает указания полного имени тестового класса. Например, для запуска тестового класса, показанного выше, в листинге 14.6, его нужно скомпилировать с помощью такой команды:

```
$ scalac -cp scalatest-app.jar:scalaxml.-jar TVSetSpec.scala
```

Затем его можно будет запустить, воспользовавшись командой:

```
$ scala -cp scalatest-app.jar:scalaxml.-jar org.scalatest.run TVSetSpec
```

Используя флаг `-cp`, вы помещаете JAR-файлы `scalatest-app` и `scala-xml` в путь класса. (При скачивании в имена JAR-файлов будут включены встроенные номера версий.) Следующий элемент, `org.scalatest.run`, является полным именем приложения. Scala запустит это приложение и передаст ему все остальные элементы командной строки в качестве аргументов. Аргумент `TVSetSpec` указывает на выполняемый набор. Результат показан на рис. 14.1.

Рис. 14.1. Вывод, полученный при запуске org.scalatest.run

Резюме

В этой главе мы показали примеры примешивания утверждений непосредственно в рабочий код, а также способы их внешней записи в тестах. Вы увидели, что программисты, работающие на Scala, могут воспользоваться преимуществами таких популярных средств тестирования от сообщества Java, как JUnit и TestNG, и более новыми средствами, разработанными исключительно для Scala, например ScalaTest, ScalaCheck и specs2. И утверждения, встроенные в код, и внешние тесты могут помочь повысить качество ваших программных продуктов. Понимая важность этих технологий, мы представили их в данной главе, немного отклонившись от учебного материала по Scala. А в следующей главе вернемся к учебнику по языку и рассмотрим весьма полезный аспект Scala — сопоставление с образцом.

15 Case-классы и сопоставление с образцом

В данной главе вводятся понятия *case-классов* и *сопоставления с образцом* (pattern matching) — парной конструкции, способствующей созданию обычных, неинкапсулированных структур данных. Особенно полезны эти две конструкции при работе с древовидными рекурсивными данными.

Если вам уже приходилось программировать на функциональном языке, то, возможно, с сопоставлением с образцом вы уже знакомы. А вот понятие `case`-классов будет для вас новым. `Case`-классы в Scala позволяют применять сопоставление с образцом к объектам и не требуют большого объема шаблонного кода. По сути, чтобы приспособить отдельно взятый класс к сопоставлению с образцом, нужно лишь добавить к нему ключевое слово `case`.

Эту главу мы начнем с простого примера `case`-классов и сопоставления с образцом. Затем дадим обзор всех видов поддерживаемых образцов (или паттернов), рассмотрим роль *запечатанных классов* (sealed classes), тип `Option` и покажем некоторые неочевидные места в языке, где используется сопоставление с образцом. В заключение покажем более объемный и приближенный к реальному использованию пример сопоставления с образцом.

15.1. Простой пример

Прежде чем вникать во все правила и нюансы сопоставления с образцом, есть смысл рассмотреть простой пример, дающий общее представление. Допустим, нужно написать библиотеку, которая работает с арифметическими выражениями и, возможно, является частью разрабатываемого предметно-ориентированного языка.

Первым шагом к решению этой задачи будет определение входных данных. Чтобы ничего не усложнять, сконцентрируемся на арифметических выражениях, состоящих из переменных, чисел и унарных и бинарных операций. Все это выражается иерархией классов Scala, показанной в листинге 15.1.

Листинг 15.1. Определение case-классов

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
    left: Expr, right: Expr) extends Expr
```

Эта иерархия включает абстрактный базовый класс `Expr` с четырьмя подклассами, по одному для каждого вида рассматриваемых выражений[1]. Тела всех пяти классов пусты. Как уже упоминалось, при желании в Scala пустые тела классов в фигурные скобки можно не заключать, следовательно, определение `class C` аналогично определению `class C {}`.

Case-классы

Еще одна особенность объявлений в листинге 15.1 (см. выше), которая заслуживает внимания, — наличие у каждого подкласса модификатора `case`. Классы с таким модификатором называются `case`-классами. Использование этого модификатора заставляет компилятор Scala добавлять к вашему классу некоторые синтаксические удобства.

Первое синтаксическое удобство заключается в том, что к классу добавляется фабричный метод с именем данного класса. Это означает, к примеру, что для создания `var`-объекта можно применить код `Var("x")`, не используя несколько более длинный вариант `new Var("x")`:

```
scala> val v = Var("x")
v: Var = Var(x)
```

Особенно полезны фабричные методы благодаря их вложенности. Теперь код не загроможден ключевыми словами `new`, поэтому структуру выражения можно воспринять с одного взгляда:

```
scala> val op = BinOp("+", Number(1), v)
op: BinOp = BinOp(+,Number(1.0),Var(x))
```

Второе синтаксическое удобство заключается в том, что все аргументы в списке параметров `case`-класса автоматически получают префикс `val`, то есть сохраняются в качестве полей:

```
scala> v.name
res0: String = x

scala> op.left
res1: Expr = Number(1.0)
```

[1] Вместо абстрактного класса можно было бы смоделировать корень этой иерархии классов в виде трейта. Моделирование в виде абстрактного класса может оказаться чуть-чуть эффективнее.

Третье удобство состоит в том, что компилятор добавляет к вашему классу «естественную» реализацию методов `toString`, `hashCode` и `equals`. Они будут заниматься подготовкой данных к выводу, их хешированием и сравнением всего дерева, состоящего из класса, и (рекурсивно) всех его аргументов. Поскольку метод `==` в Scala всегда передает полномочия методу `equals`, то это значит, что элементы `case`-классов всегда сравниваются структурно:

```
scala> println(op)
BinOp(+,Number(1.0),Var(x))

scala> op.right == Var("x")
res3: Boolean = true
```

И наконец, чтобы создать измененные копии, компилятор добавляет к вашему классу метод `copy`. Он пригодится для создания нового экземпляра класса, аналогичного другому экземпляру, за исключением того, что будет отличаться одним или двумя атрибутами. Метод работает за счет использования именованных параметров и параметров по умолчанию (см. раздел 8.8). Применение именованных параметров позволяет указать требуемые изменения. А для любого неуказанного параметра используется значение из старого объекта. Посмотрим в качестве примера, как можно создать операцию, похожую на `op` во всем, кроме того, что будет изменен параметр `operator`:

```
scala> op.copy(operator = "-")
res4: BinOp = BinOp(-,Number(1.0),Var(x))
```

Все эти соглашения в качестве небольшого бонуса придают вашей работе массу удобств. Нужно просто указать модификатор `case`, и ваши классы и объекты приобретут гораздо более оснащенный вид. Они станут больше за счет создания дополнительных методов и неявного добавления поля для каждого параметра конструктора. Но самым большим преимуществом `case`-классов является то, что они поддерживают сопоставления с образцом.

Сопоставление с образцом

Предположим, нужно упростить арифметические выражения только что представленных видов. Существует множество возможных правил упрощения. В качестве иллюстрации подойдут три следующих правила:

```
UnOp("-", UnOp("-", e))  => e // двойное отрицание
BinOp("+", e, Number(0)) => e // прибавление нуля
BinOp("*", e, Number(1)) => e // умножение на единицу
```

Как показано в листинге 15.2, чтобы в Scala сформировать ядро функции упрощения с помощью сопоставления с образцом, эти правила можно взять практически в неизменном виде. Показанную в листинге функцию `simplifyTop` можно использовать следующим образом:

```
scala> simplifyTop(UnOp("-", UnOp("-", Var("x"))))
res4: Expr = Var(x)
```

Листинг 15.2. Функция simplifyTop, выполняющая сопоставление с образцом

```
def simplifyTop(expr: Expr): Expr = expr match {
  case UnOp("-", UnOp("-", e))  => e // двойное отрицание
  case BinOp("+", e, Number(0)) => e // прибавление нуля
  case BinOp("*", e, Number(1)) => e // умножение на единицу
  case _ => expr
}
```

Правая часть `simplifyTop` состоит из выражения `match`, которое соответствует `switch` в Java, но записывается после выражения выбора. Иными словами, оно выглядит как:

выбор `match { ` *альтернативы* `}`

вместо:

`switch (`*выбор*`) { `*альтернативы*` }`

Сопоставление с образцом включает последовательность *альтернатив*, каждый из которых начинается с ключевого слова `case`. Каждая альтернатива состоит из *паттерна* и одного или нескольких выражений, которые будут вычислены при соответствии паттерну. Обозначение стрелки `=>` отделяет паттерн от выражений.

Выражение `match` вычисляется проверкой соответствия каждого из паттернов в порядке их написания. Выбирается первый же соответствующий паттерн, а также выбирается и выполняется та часть, которая следует за обозначением стрелки.

Паттерн-константа вида `+` или `1` соответствует значениям, равным константе в случае применения метода `==`. *Паттерн-переменная* вида `e` соответствует любому значению. Затем переменная ссылается на это же значение в правой части условия `case`. Обратите внимание: в данном примере первые три альтернативы вычисляются в `e`, то есть в переменную, связанную внутри соответствующего паттерна. *Подстановочный паттерн* (`_`) также соответствует любому значению, но без представления имени переменной для ссылки на это значение. Стоит отметить, как в листинге 15.2 (см. выше) выражение `match` заканчивается условием `case`, которое применяется при отсутствии соответствующих паттернов и не предполагает никаких действий с выражением. Вместо этого получается просто выражение `expr`, в отношении которого и выполняется сопоставление с образцом.

Паттерн-конструктор выглядит как `UnOp("-", e)`. Он соответствует всем значениям типа `UnOp`, первый аргумент которых соответствует `"-"`, а второй — `e`. Обратите внимание: аргументы конструктора сами являются паттернами. Это позволяет составлять многоуровневые паттерны, используя краткую форму записи.

Примером может послужить следующий паттерн:

```
UnOp("-", UnOp("-", e))
```

Представьте попытку реализовать такую же функциональную возможность с помощью шаблона проектирования visitor![1] Практически так же трудно предста-

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020.

вить реализацию такой же функциональной возможности в виде длинной последовательности инструкций, проверок соответствия типам и явного приведения типов.

Сравнение match со switch

Выражения `match` могут быть представлены в качестве общих случаев `switch`-выражений в стиле Java. В Java `switch`-выражение может быть вполне естественно представлено в виде `match`-выражения, где каждый паттерн является константой, а последний паттерн может быть подстановочным (который представлен в `switch`-выражении вариантом, используемым при отсутствии других соответствий).

И тем не менее следует учитывать три различия. Во-первых, `match` является *выражением* языка Scala, то есть всегда вычисляется в значение. Во-вторых, применяемые в Scala выражения альтернатив никогда не «выпадают» в следующий вариант. В-третьих, если не найдено соответствие ни одному из паттернов, то выдается исключение `MatchError`. Следовательно, вам придется всегда обеспечивать охват всех возможных вариантов, даже если это будет означать вариант по умолчанию, в котором не делается ничего.

Пример показан в листинге 15.3. Второй необходим, поскольку без него выражение `match` выдаст исключение `MatchError` для любого `expr`-аргумента, не являющегося `BinOp`. В данном примере для этого второго варианта не указан никакой код, поэтому при его срабатывании ничего не произойдет. Результатом в любом случае будет `unit`-значение `()`, которое также будет результатом вычисления всего выражения `match`.

Листинг 15.3. Сопоставление с образцом с пустым вариантом по умолчанию

```
expr match {
  case BinOp(op, left, right) =>
    println(s"$expr является бинарной операцией")
  case _ =>
}
```

15.2. Разновидности паттернов

В предыдущем примере мы кратко описали некоторые разновидности паттернов. А теперь потратим немного времени на более подробное изучение каждого из них.

Синтаксис паттернов довольно прост, поэтому не стоит особо переживать из-за него. Все паттерны выглядят точно так же, как и соответствующие им выражения. Например, если взять иерархию из листинга 15.1 (см. выше), то паттерн `Var(x)` соответствует любому выражению, содержащему переменную, с привязкой `x` к имени переменной. Будучи использованным в качестве выражения, `Var(x)` с точно таким же синтаксисом воссоздает эквивалентный объект, предполагая, что идентификатор `x` уже привязан к имени переменной. В синтаксисе паттернов все прозрачно, поэтому главное, на что следует обратить внимание, — это какого вида паттерны можно применять.

Подстановочные паттерны

Подстановочный паттерн (_) соответствует абсолютно любому объекту. Вы уже видели, как он используется в качестве общего паттерна, выявляющего все оставшиеся альтернативы:

```
expr match {
  case BinOp(op, left, right) =>
    println(s"$expr является бинарной операцией")
  case _ => // обработка общего варианта
}
```

Кроме того, подстановочные паттерны могут использоваться для игнорирования тех частей объекта, которые не представляют для вас интереса. Например, в предыдущем примере нас не интересует, что представляют собой элементы бинарной операции, в нем лишь проверяется, является ли она бинарной. Поэтому, как показано в листинге 15.4, для элементов `BinOp` в коде также могут использоваться подстановочные паттерны.

Листинг 15.4. Сопоставление с образцом с подстановочными паттернами

```
expr match {
  case BinOp(_, _, _) => println(s"$expr является бинарной операцией")
  case _ => println("Это что-то другое")
}
```

Паттерны-константы

Паттерн-константа соответствует только самому себе. В качестве константы может использоваться любой литерал. Например, паттернами-константами являются `5`, `true` и `"hello"`. В качестве константы может использоваться и любой `val`- или объект-одиночка. Так, объект-одиночка `Nil` является паттерном, соответствующим только пустому списку. Некоторые примеры паттернов-констант показаны в листинге 15.5.

Листинг 15.5. Сопоставление с образцом с использованием паттернов-констант

```
def describe(x: Any) = x match {
  case 5 => "пять"
  case true => "правда"
  case "hello" => "привет!"
  case Nil => "пустой список"
  case _ => "что-то другое"
}
```

А вот как сопоставление с образцом, показанное в данном листинге, выглядит в действии:

```
scala> describe(5)
res6: String = пять
```

```
scala> describe(true)
res7: String = правда

scala> describe("hello")
res8: String = привет!

scala> describe(Nil)
res9: String = пустой список

scala> describe(List(1,2,3))
res10: String = что-то другое
```

Патерны-переменные

Паттерн-переменная соответствует любому объекту точно так же, как подстановочный паттерн, но в отличие от него Scala привязывает переменную к объекту. Затем с помощью этой переменной можно в дальнейшем воздействовать на объект. Например, в листинге 15.6 показано сопоставление с образцом, имеющее специальный вариант для нуля и общий вариант для всех остальных значений. В общем варианте используется паттерн-переменная, и потому у него есть имя для переменной, независимо от того, что это на самом деле.

Листинг 15.6. Сопоставление с образцом с использованием паттерна-переменной

```
expr match {
  case 0 => "нуль"
  case somethingElse => "не нуль: " + somethingElse
}
```

Переменная или константа?

У паттернов-констант могут быть символические имена. Вы уже это видели, когда в качестве образца использовался `Nil`. А вот похожий пример, где в сопоставлении с образцом задействуются константы `E` (2,71828...) и `Pi` (3,14159...):

```
scala> import math.{E, Pi}
import math.{E, Pi}

scala> E match {
         case Pi => "математический казус? Pi = " + Pi
         case _ => "OK"
       }
res11: String = OK
```

Как и ожидалось, значение `E` не равно значению `Pi`, поэтому вариант «математический казус» не выбирается.

А как компилятор Scala распознает, что `Pi` — это константа, импортированная из `scala.math`, а не переменная, обозначающая само значение селектора? Во избежание путаницы в Scala действует простое лексическое правило: обычное имя,

начинающееся с буквы в нижнем регистре, считается переменной паттерна, а все другие ссылки считаются константами. Чтобы заметить разницу, создайте для `pi` псевдоним с первой буквой, указанной в нижнем регистре, и попробуйте в работе следующий код:

```
scala> val pi = math.Pi
pi: Double = 3.141592653589793

scala> E match {
         case pi => "математический казус? Pi = " + pi
       }
res12: String = математический казус? Pi = 2.718281828459045
```

Здесь компилятор даже не позволит вам добавить вариант по умолчанию. Поскольку `pi` — паттерн-переменная, то будет соответствовать всем вводимым данным, поэтому до следующих вариантов дело просто не дойдет:

```
scala> E match {
         case pi => "математический казус? Pi = " + pi
         case _ => "OK"
       }
         case pi => "математический казус? Pi = " + pi
              ^
On line 2: warning: patterns after a variable pattern cannot match (SLS 8.1.1)
```

Но при необходимости для паттерна-константы можно задействовать имя, начинающееся с буквы в нижнем регистре; для этого придется воспользоваться одним из двух приемов. Если константа является полем какого-нибудь объекта, то перед ней можно поставить префикс-классификатор. Например, `pi` — паттерн-переменная, а `this.pi` или `obj.pi` — константы, несмотря на то что их имена начинаются с букв в нижнем регистре. Если это не сработает (поскольку, скажем, `pi` — локальная переменная), то как вариант можно будет заключить имя переменной в обратные кавычки. Например, `` `pi` `` будет опять восприниматься как константа, а не как переменная:

```
scala> E match {
         case `pi` => "математический казус? Pi = " + pi
         case _ => "OK"
       }
res14: String = OK
```

Как вы, наверное, заметили, использование для идентификаторов в Scala синтаксиса с обратными кавычками во избежание в коде необычных обстоятельств преследует две цели. Здесь показано, что этот синтаксис может применяться для рассмотрения идентификатора с именем, начинающимся с буквы в нижнем регистре, в качестве константы при сопоставлении с образцом. Ранее, в разделе 6.10, было показано, что этот синтаксис может использоваться также для трактовки ключевого слова в качестве обычного идентификатора. Например, в выражении `writingThread.` `` `yield` `` `()` слово `yield` трактуется как идентификатор, а не ключевое слово.

Паттерны-конструкторы

Реальная эффективность сопоставления с образцом проявляется именно в конструкторах. Паттерн-конструктор выглядит как `BinOp("+", e, Number(0))`. Он состоит из имени (`BinOp`), после которого в круглых скобках стоят несколько образцов: `"+"`, `e` и `Number(0)`. При условии, что имя обозначает `case`-класс, такой паттерн показывает следующее: сначала проверяется принадлежность элемента к названному `case`-классу, а затем соответствие параметров конструктора объекта предоставленным дополнительным паттернам.

Эти дополнительные паттерны означают, что в паттернах Scala поддерживаются *глубкие сопоставления* (deep matches). Такой паттерн проверяет не только предоставленный объект верхнего уровня, но и его содержимое на соответствие следующим паттернам. Дополнительные паттерны сами по себе могут быть паттернами-конструкторами, поэтому их можно использовать для проверки объекта произвольной глубины. Например, паттерн, показанный в листинге 15.7, проверяет, что объект верхнего уровня относится к типу `BinOp`, третьим параметром его конструктора является число `Number` и значение поля этого числа — `0`. Весь паттерн умещается в одну строку кода, хотя выполняет проверку на глубину в три уровня.

Листинг 15.7. Сопоставление с образцом с использованием паттерна-конструктора

```
expr match {
  case BinOp("+", e, Number(0)) => println("глубокое соответствие")
  case _ =>
}
```

Паттерны-последовательности

По аналогии с сопоставлением с `case`-классами можно сопоставлять с такими типами последовательностей, как `List` или `Array`. Допустимо воспользоваться тем же синтаксисом, но теперь в паттерне вы можете указать любое количество элементов. В листинге 15.8 показан паттерн для проверки того факта, что трехэлементный список начинается с нуля.

Листинг 15.8. Паттерн-последовательность фиксированной длины

```
expr match {
  case List(0, _, _) => println("соответствие найдено")
  case _ =>

}
```

Если нужно сопоставить с последовательностью, не указывая ее длину, то в качестве последнего элемента паттерна-последовательности можно указать образец `_*`. Он имеет весьма забавный вид и соответствует любому количеству элементов внутри последовательности, включая нуль элементов. В листинге 15.9 показан пример, соответствующий любому списку, который начинается с нуля, независимо от длины этого списка.

Листинг 15.9. Паттерн-последовательность произвольной длины

```
expr match {
  case List(0, _*) => println("соответствие найдено ")
  case _ =>
}
```

Паттерны-кортежи

Можно выполнять и сопоставление с кортежами. Паттерн вида `(a, b, c)` соответствует произвольному трехэлементному кортежу. Пример показан в листинге 15.10.

Листинг 15.10. Сопоставление с образцом с использованием паттерна-кортежа

```
def tupleDemo(expr: Any) =
  expr match {
    case (a, b, c) => println("соответствует " + a + b + c)
    case _ =>
  }
```

Если загрузить показанный в листинге 15.10 метод `tupleDemo` в интерпретатор и передать ему кортеж из трех элементов, то получится следующая картина:

```
scala> tupleDemo(("a ", 3, "-tuple"))
соответствует a 3-tuple
```

Типизированные паттерны

Типизированный паттерн (typed pattern) можно использовать в качестве удобного заменителя для проверок типов и приведения типов. Пример показан в листинге 15.11.

Листинг 15.11. Сопоставление с образцом с использованием типизированных паттернов

```
def generalSize(x: Any) = x match {
  case s: String => s.length
  case m: Map[_, _] => m.size
  case _ => -1
}
```

А вот несколько примеров использования `generalSize` в интерпретаторе Scala:

```
scala> generalSize("abc")
res16: Int = 3

scala> generalSize(Map(1 -> 'a', 2 -> 'b'))
res17: Int = 2

scala> generalSize(math.Pi)
res18: Int = -1
```

Метод `generalSize` возвращает размер или длину объектов различных типов. Типом его аргумента является `Any`, поэтому им может быть любое значение. Если

в качестве типа аргумента выступает `String`, то метод возвращает длину строки. Образец `s: String` является типизированным паттерном и соответствует каждому (ненулевому) экземпляру класса `String`. Затем на эту строку ссылается паттерн-переменная `s`.

Заметьте: даже притом, что `s` и `x` ссылаются на одно и то же значение, типом `x` является `Any`, а типом `s` — `String`. Поэтому в альтернативном выражении, соответствующем паттерну, можно воспользоваться кодом `s.length`, но нельзя — кодом `x.length`, поскольку в типе `Any` отсутствует член `length`.

Эквивалентный, но более многословный способ достичь такого же результата сопоставления с типизированным образцом — использовать проверку типа с его последующим приведением. В Scala для этого применяется не такой синтаксис, как в Java. К примеру, чтобы проверить, относится ли выражение `expr` к типу `String`, используется такой код:

```
expr.isInstanceOf[String]
```

Для приведения того же выражения к типу `String` используется код:

```
expr.asInstanceOf[String]
```

Применяя проверку и приведение типа, можно переписать первый вариант предыдущего `match`-выражения, получив код, показанный в листинге 15.12.

Листинг 15.12. Использование isInstanceOf и asInstanceOf (плохой стиль)

```
if (x.isInstanceOf[String]) {
  val s = x.asInstanceOf[String]
  s.length
} else ...
```

Операторы `isInstanceOf` и `asInstanceOf` считаются предопределенными методами класса `Any`, получающими параметр типа в квадратных скобках. Фактически `x.asInstanceOf[String]` — частный случай вызова метода с явно заданным параметром типа `String`.

Как вы уже заметили, написание проверок и приведений типов в Scala страдает излишним многословием. Сделано это намеренно, поскольку подобная практика не приветствуется. Как правило, лучше воспользоваться сопоставлением с типизированным образцом. В частности, подобный подход будет оправдан, если нужно выполнить две операции: проверку типа и его приведение, так как обе они будут сведены к единственному сопоставлению.

Второй вариант `match`-выражения в листинге 15.11 содержит типизированный паттерн `m: Map[_, _]`. Он соответствует любому значению, являющемуся отображением каких-либо произвольных типов ключа и значения, и позволяет `m` ссылаться на это значение. Поэтому `m.size` имеет правильный тип и возвращает размер отображения. Знаки подчеркивания в типизированном паттерне[1] подобны таким же знакам в подстановочных паттернах. Вместо них можно указывать переменные типа с символами в нижнем регистре.

[1] В типизированном паттерне m: Map[_, _], часть "Map[_, _]" называется паттерном типа.

Затирание типов

А можно ли также проводить проверку на отображение с конкретными типами элементов? Это пригодилось бы, скажем, для проверки того, является ли заданное значение отображением типа `Int` на тип `Int`. Попробуем:

```
scala> def isIntIntMap(x: Any) = x match {
         case m: Map[Int, Int] => true
         case _ => false
       }
         case m: Map[Int, Int] => true
                 ^
On line 2: warning: non-variable type argument Int in type pattern scala.
collection.immutable.Map[Int,Int] (the underlying of Map[Int,Int]) is
unchecked since it is eliminated by erasure
```

В Scala точно так же, как и в Java, используется модель *затирания* обобщенных типов. Это значит, в ходе выполнения программы никакая информация об аргументах типов не сохраняется. Следовательно, способов определить в ходе выполнения программы, создавался ли заданный `Map`-объект с двумя `Int`-аргументами, а не с аргументами других типов, не существует. Система может лишь определить, что значение является отображением (`Map`) неких произвольных параметров типа. Убедиться в таком поведении можно, применив `isIntIntMap` к различным экземплярам класса `Map`:

```
scala> isIntIntMap(Map(1 -> 1))
res19: Boolean = true

scala> isIntIntMap(Map("abc" -> "abc"))
res20: Boolean = true
```

Первое применение возвращает `true`, что выглядит вполне корректно, но второе тоже возвращает `true`, и это может оказаться сюрпризом. Чтобы оповестить вас о возможном непонятном поведении программы в ходе ее выполнения, компилятор выдает предупреждение о том, что не контролирует это поведение, похожее на показанные ранее.

Единственное исключение из правила затирания — массивы, поскольку в Java, а также в Scala они обрабатываются особым образом. Тип элемента массива сохраняется вместе со значением массива, поэтому к нему можно применить сопоставление с образцом. Пример выглядит так:

```
scala> def isStringArray(x: Any) = x match {
         case a: Array[String] => "yes"
         case _ => "no"
       }
isStringArray: (x: Any)String

scala> val as = Array("abc")
as: Array[String] = Array(abc)
```

```
scala> isStringArray(as)
res21: String = yes

scala> val ai = Array(1, 2, 3)
ai: Array[Int] = Array(1, 2, 3)

scala> isStringArray(ai)
res22: String = no
```

Привязка переменной

Кроме использования отдельно взятого паттерна-переменной можно также добавить переменную к любому другому паттерну. Нужно просто указать имя переменной, знак «собачка» (`@`), а затем паттерн. Это даст вам паттерн с привязанной переменной, то есть паттерн для выполнения обычного сопоставления с образцом с возможностью в случае совпадения присвоить переменной соответствующий объект, как и при использовании обычного паттерна-переменной.

В качестве примера в листинге 15.13 показано сопоставление с образцом — поиск операции получения абсолютного значения, применяемой в строке дважды. Такое выражение можно упростить, однократно получив абсолютное значение.

Листинг 15.13. Паттерн с привязкой переменной (посредством использования знака @)

```
expr match {
  case UnOp("abs", e @ UnOp("abs", _)) => e
  case _ =>
}
```

Пример, показанный в данном листинге, включает паттерн с привязкой переменной, где в качестве переменной выступает `e`, а в качестве паттерна — `UnOp("abs", _)`. Если будет найдено соответствие всему паттерну, то часть, которая соответствует `UnOp("abs", _)`, станет доступна как значение переменной `e`. Результатом варианта будет просто `e`, поскольку у `e` такое же значение, что и у `expr`, но с меньшим на единицу количеством операций получения абсолютного значения.

15.3. Ограждение образца

Иногда синтаксическое сопоставление с образцом является недостаточно точным. Предположим, перед вами стоит задача сформулировать правило упрощения, заменяющее выражение сложения с двумя одинаковыми операндами, такое как `e + e`, умножением на 2, например `e * 2`. На языке деревьев `Expr` выражение вида:

```
BinOp("+", Var("x"), Var("x"))
```

этим правилом будет превращено в следующее:

```
BinOp("*", Var("x"), Number(2))
```

Правило можно попробовать выразить следующим образом:

```
scala> def simplifyAdd(e: Expr) = e match {
         case BinOp("+", x, x) => BinOp("*", x, Number(2))
         case _ => e
       }
         case BinOp("+", x, x) => BinOp("*", x, Number(2))
                                  ^
On line 2: error: x is already defined as value x
```

Попытка будет неудачной, поскольку в Scala паттерны должны быть *линейными*: паттерн-переменная может появляться в образце только один раз. Но, как показано в листинге 15.14, соответствие можно переформулировать с помощью *ограничителя паттернов* (pattern guard).

Листинг 15.14. Сопоставление с образцом с применением ограждения паттернов

```
scala> def simplifyAdd(e: Expr) = e match {
         case BinOp("+", x, y) if x == y =>
           BinOp("*", x, Number(2))
         case _ => e
       }
simplifyAdd: (e: Expr)Expr
```

Ограждение паттерна указывается после образца и начинается с ключевого слова `if`. В качестве ограждения может использоваться произвольное булево выражение, которое обычно ссылается на переменные в образце. При наличии ограждения паттернов соответствие считается найденным, только если ограждение вычисляется в `true`. Таким образом, первый вариант показанного ранее кода соответствует только бинарным операциям, имеющим два одинаковых операнда.

А вот как выглядят некоторые другие огражденные паттерны:

```
// соответствует только положительным целым числам
case n: Int if 0 < n => ...

// соответствует только строкам, начинающимся с буквы 'a'
case s: String if s(0) == 'a' => ...
```

15.4. Наложение паттернов

Паттерны применяются в порядке их указания. Версия метода `simplify`, показанная в листинге 15.15, представляет собой пример, в котором порядок следования вариантов имеет значение.

Листинг 15.15. Выражение сопоставления, в котором порядок следования вариантов имеет значение

```
def simplifyAll(expr: Expr): Expr = expr match {
  case UnOp("-", UnOp("-", e)) =>
    simplifyAll(e) // '-' является своей собственной обратной величиной
  case BinOp("+", e, Number(0)) =>
    simplifyAll(e) // '0' — нейтральный элемент для '+'
```

```
  case BinOp("*", e, Number(1)) =>
    simplifyAll(e) // '1' — нейтральный элемент для '*'
  case UnOp(op, e) =>
    UnOp(op, simplifyAll(e))
  case BinOp(op, l, r) =>
    BinOp(op, simplifyAll(l), simplifyAll(r))
  case _ => expr
}
```

Версия метода `simplify`, показанная в данном листинге, станет применять правила упрощения в любом месте выражения, а не только в его верхней части, как это сделала бы версия `simplifyTop`. Данную версию можно вывести из версии `simplifyTop`, добавив два дополнительных варианта для обычных унарных и бинарных выражений (четвертый и пятый варианты в листинге).

В четвертом варианте используется паттерн `UnOp(op, e)`, который соответствует любой унарной операции. Оператор и операнд унарной операции могут быть какими угодно. Они привязаны к паттернам-переменным `op` и `e` соответственно. Альтернативой в данном варианте будет рекурсивное применение `simplifyAll` к операнду `e` с последующим перестроением той же самой унарной операции с (возможно) упрощенным операндом. Пятый вариант для `BinOp` аналогичен четвертому: он является вариантом «поймать все» для произвольных бинарных операций, который рекурсивно применяет метод упрощения к своим двум операндам.

Важным обстоятельством в этом примере является то, что варианты «поймать все» следуют *после* более конкретизированных правил упрощения. Если расположить их в другом порядке, то вариант «поймать все» будет запущен вместо более конкретизированных правил. Во многих случаях компилятор будет жаловаться на такие попытки. Например, вот как выглядит выражение `match`, которое не пройдет компиляцию, поскольку первый вариант будет соответствовать всему тому, чему будет соответствовать второй вариант:

```
scala> def simplifyBad(expr: Expr): Expr = expr match {
         case UnOp(op, e) => UnOp(op, simplifyBad(e))
         case UnOp("-", UnOp("-", e)) => e
       }
         case UnOp("-", UnOp("-", e)) => e
                                         ^
On line 3: warning: unreachable code
simplifyBad: (expr: Expr)Expr
```

15.5. Запечатанные классы

При написании сопоставления с образцом нужно удостовериться в том, что охвачены все возможные варианты. Иногда это можно сделать, добавив в конец `match` вариант по умолчанию, но данный способ применим, только когда есть вполне определенное поведение по умолчанию. А что делать, если его нет? Как узнать, что охвачены все варианты и нет опасности упустить что-либо?

Чтобы определить пропущенные в выражении `match` комбинации паттернов, можно обратиться за помощью к компилятору Scala. Для этого компилятор должен иметь возможность сообщить обо всех потенциальных вариантах. По сути, в Scala это сделать нереально, поскольку классы могут быть определены в любое время и в произвольных блоках компиляции. Например, ничто не помешает вам добавить к иерархии класса `Expr` пятый `case`-класс не в том блоке компиляции, в котором определены четыре других `case`-класса, а в другом.

Альтернативой этому может стать превращение суперкласса ваших `case`-классов в *запечатанный* класс. У такого запечатанного класса не может быть никаких дополнительных подклассов, кроме тех, которые определены в том же самом файле. Особую пользу из этого можно извлечь при сопоставлении с образцом, поскольку запечатанность класса будет означать, что беспокоиться придется только по поводу тех подклассов, о которых вам уже известно. Более того, будет улучшена поддержка со стороны компилятора. При сопоставлении с образцом `case`-классам, являющимся наследниками запечатанного класса, компилятор в предупреждении отметит пропущенные комбинации паттернов.

Если создается иерархия классов, предназначенная для сопоставления с образцом, то нужно предусмотреть ее запечатанность. Чтобы это сделать, просто поставьте перед классом на вершине иерархии ключевое слово `sealed`. Программисты, использующие вашу иерархию классов, при сопоставлении с образцом будут чувствовать себя уверенно. Таким образом, ключевое слово `sealed` зачастую выступает лицензией на сопоставление с образцом. Пример, в котором `Expr` превращается в запечатанный класс, показан в листинге 15.16.

Листинг 15.16. Запечатанная иерархия case-классов

```
sealed abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
    left: Expr, right: Expr) extends Expr
```

А теперь определим сопоставление с образцом, в котором пропущены некоторые возможные варианты:

```
def describe(e: Expr): String = e match {
  case Number(_) => "число"
  case Var(_) => "переменная"
}
```

В результате будет получено следующее предупреждение компилятора:

```
warning: match is not exhaustive!
missing combination           UnOp
missing combination          BinOp
```

Оно сообщает о существовании риска генерации вашим кодом исключения `MatchError`, поскольку некоторые возможные паттерны (`UnOp`, `BinOp`) не обрабатываются. Предупреждение указывает на потенциальный источник сбоя в ходе

выполнения программы, поэтому обычно хорошо помогает при корректировке ее кода.

Но порой можно столкнуться с ситуацией, в которой компилятор при выдаче предупреждений проявляет излишнюю дотошность. Например, из контекста может быть известно, что показанный ранее метод `describe` будет применяться только к выражениям типа `Number` или `Var`, следовательно, исключение `MatchError` не станет генерироваться. Чтобы избавиться от предупреждения, к методу можно добавить третий вариант по умолчанию:

```
def describe(e: Expr): String = e match {
  case Number(_) => "число"
  case Var(_) => "переменная"
  case _ => throw new RuntimeException // Не должно произойти
}
```

Решение вполне работоспособное, однако не идеальное. Вряд ли вас обрадует принуждение добавить код, который никогда не будет выполнен (по вашему мнению), лишь для того, чтобы успокоить компилятор.

Более экономной альтернативой станет добавление к селектору выражения сопоставления с образцом аннотации `@unchecked`. Делается это следующим образом:

```
def describe(e: Expr): String = (e: @unchecked) match {
  case Number(_) => "число"
  case Var(_) => "переменная"
}
```

Аннотации рассматриваются в главе 27. В общем, аннотации можно добавлять к выражению точно так же, как это делается при добавлении типа: нужно после выражения поставить двоеточие, знак «собачка» и указать название аннотации. Например, в данном случае к переменной `e` добавляется аннотация `@unchecked`, для чего используется код `e: @unchecked`. Аннотация `@unchecked` имеет особое значение для сопоставления с образцом. Если выражение селектора поиска содержит данную аннотацию, то исчерпывающая проверка последующих паттернов будет подавлена.

15.6. Тип Option

Для необязательных значений в Scala имеется стандартный тип `Option`. Значение такого типа может быть двух видов: это либо `Some(`x`)`, где x — реальное значение, либо `None`-объект, который представляет отсутствующее значение.

Необязательные значения выдаются некоторыми стандартными операциями над коллекциями Scala. Например, метод `get` из Scala-класса `Map` производит `Some(`*значение*`)`, если найдено *значение*, соответствующее заданному ключу, или `None`, если заданный ключ не определен в `Map`-объекте. Пример выглядит так:

```
scala> val capitals =
         Map("France" -> "Paris", "Japan" -> "Tokyo")
```

```
capitals: scala.collection.immutable.Map[String,String] =
Map(France -> Paris, Japan -> Tokyo)

scala> capitals get "France"
res23: Option[String] = Some(Paris)

scala> capitals get "North Pole"
res24: Option[String] = None
```

Самый распространенный способ разобрать необязательные значения — использовать сопоставление с образцом, например:

```
scala> def show(x: Option[String]) = x match {
         case Some(s) => s
         case None => "?"
       }
show: (x: Option[String])String

scala> show(capitals get "Japan")
res25: String = Tokyo

scala> show(capitals get "France")
res26: String = Paris

scala> show(capitals get "North Pole")
res27: String = ?
```

Тип `Option` применяется в программах на языке Scala довольно часто. Его использование можно сравнить с доминирующей в Java идиомой `null`, показывающей отсутствие значения. Например, метод `get` из `java.util.HashMap` возвращает либо значение, сохраненное в `HashMap`, либо `null`, если значение не было найдено. В Java такой подход работает, но, применяя его, легко допустить ошибку, поскольку на практике довольно трудно отследить, каким переменным в программе разрешено иметь значение `null`.

В случае, когда переменной разрешено иметь значение `null`, вы должны вспомнить о ее проверке на наличие этого значения при каждом использовании. Если забыть выполнить эту проверку, то появится вероятность генерации в ходе выполнения программы исключений `NullPointerException`. Подобные исключения могут генерироваться довольно редко, поэтому с выявлением ошибки при тестировании могут возникнуть затруднения. В Scala такой подход вообще не сработает, поскольку этот язык позволяет сохранять типы значений в хеш-отображениях, а `null` не является допустимым элементом для типов значений. Например, `HashMap[Int, Int]` не может вернуть `null`, чтобы обозначить отсутствие элемента.

Вместо этого в Scala для указания значения, которое может отсутствовать, применяется тип `Option`. По сравнению с подходом, используемым в Java, подход Scala к работе с необязательными значениями имеет ряд преимуществ. Во-первых, тем, кто читает код, намного понятнее, что переменная, типом которой является `Option[String]`, — необязательная переменная `String`, а не переменная типа `String`, которая иногда может иметь значение `null`. Во-вторых, и это важнее всего, рассмо-

тренные ранее ошибки программирования, связанные с использованием переменной, которая может иметь значение `null`, без предварительной проверки ее на `null` превращаются в Scala в ошибку типа. Если переменная имеет тип `Option[String]`, то при попытке ее использования в качестве строки ваша программа на Scala не пройдет компиляцию.

15.7. Паттерны везде

Паттерны можно использовать не только в отдельно взятых `match`-выражениях, но и во многих других местах программы на языке Scala. Рассмотрим несколько подобных мест применения паттернов.

Паттерны в определениях переменных

При определении `val`- или `var`-переменной вместо простых идентификаторов можно использовать паттерны. Например, можно, как показано в листинге 15.17, разобрать кортеж и присвоить каждую его часть собственной переменной.

Листинг 15.17. Определение нескольких переменных с помощью одного присваивания

```
scala> val myTuple = (123, "abc")
myTuple: (Int, String) = (123,abc)

scala> val (number, string) = myTuple
number: Int = 123
string: String = abc
```

Особенно полезной эта конструкция может быть при работе с `case`-классами. Если точно известен `case`-класс, с которым ведется работа, то вы можете разобрать его с помощью паттерна. Пример выглядит следующим образом:

```
scala> val exp = new BinOp("*", Number(5), Number(1))
exp: BinOp = BinOp(*,Number(5.0),Number(1.0))

scala> val BinOp(op, left, right) = exp
op: String = *
left: Expr = Number(5.0)
right: Expr = Number(1.0)
```

Последовательности вариантов в качестве частично примененных функций

Последовательность вариантов (то есть альтернатив), заключенную в фигурные скобки, можно задействовать везде, где может использоваться функциональный литерал. По сути, последовательность вариантов и есть функциональный литерал,

только более универсальный. Вместо единственной точки входа и списка параметров последовательность вариантов имеет несколько точек входа, каждой из которых присущ собственный список параметров. Каждый вариант является точкой входа в функцию, а параметры указываются с помощью паттерна. Тело каждой точки входа — правосторонняя часть варианта.

Простой пример выглядит следующим образом:

```
val withDefault: Option[Int] => Int = {
  case Some(x) => x
  case None => 0
}
```

В теле этой функции имеются два варианта. Первый соответствует `Some` и возвращает число, находящееся внутри `Some`. Второй соответствует `None` и возвращает стандартное значение нуль. А вот как используется данная функция:

```
scala> withDefault(Some(10))
res28: Int = 10

scala> withDefault(None)
res29: Int = 0
```

Такая возможность особенно полезна для библиотеки акторов Akka, поскольку позволяет определить ее метод `receive` в виде серии вариантов:

```
var sum = 0

def receive = {

case Data(byte) =>
  sum += byte

case GetChecksum(requester) =>
  val checksum = ~(sum & 0xFF) + 1
  requester ! checksum
}
```

Кроме того, стоит упомянуть еще одно общее правило: последовательность вариантов дает вам *частично* примененную функцию. Если применить такую функцию в отношении не поддерживаемого ею значения, то она сгенерирует исключение времени выполнения. Например, ниже показана частично примененная функция, которая возвращает второй элемент списка, состоящего из целых чисел:

```
val second: List[Int] => Int = {
  case x :: y :: _ => y
}
```

При компиляции этого кода компилятор вполне резонно выведет предупреждение о том, что сопоставление с образцом не охватывает все возможные варианты:

```
<console>:17: warning: match is not exhaustive!
missing combination            Nil
```

Функция справится со своей задачей, если ей передать список, состоящий из трех элементов, но не станет работать при передаче пустого списка:

```
scala> second(List(5, 6, 7))
res24: Int = 6

scala> second(List())
scala.MatchError: List()
      at $anonfun$1.apply(<console>:17)
      at $anonfun$1.apply(<console>:17)
```

Если нужно проверить, определена ли частично примененная функция, то сначала следует сообщить компилятору: вы знаете, что работаете с частично примененными функциями. Тип `List[Int] => Int` включает все функции, получающие из целочисленных списков целочисленные значения независимо от того, частично они применяются или нет. Тип, который включает только *частично* примененные функции, которые получают из целочисленных списков целочисленные значения, записывается в виде `PartialFunction[List[Int],Int]`. Ниже представлен еще один вариант функции `second`, определенной с типом частично примененной функции:

```
val second: PartialFunction[List[Int],Int] = {
  case x :: y :: _ => y
}
```

У частично примененных функций есть метод `isDefinedAt`, который может использоваться для тестирования того, определена ли функция в отношении конкретного значения. В данном случае функция определена для любого списка, состоящего по крайней мере из двух элементов:

```
scala> second.isDefinedAt(List(5,6,7))
res30: Boolean = true

scala> second.isDefinedAt(List())
res31: Boolean = false
```

Типичным образчиком частично примененной функции может послужить функциональный литерал сопоставления с образцом, подобный представленному в предыдущем примере. Фактически такое выражение преобразуется компилятором Scala в частично примененную функцию с помощью двойного преобразования паттернов: один раз для реализации реальной функции, а второй — для проверки того, определена ли функция.

Например, функциональный литерал `{ case x :: y :: _ => y }` преобразуется в следующее значение частично примененной функции:

```
new PartialFunction[List[Int], Int] {
  def apply(xs: List[Int]) = xs match {
    case x :: y :: _ => y
  }
  def isDefinedAt(xs: List[Int]) = xs match {
    case x :: y :: _ => true
    case _ => false
  }
}
```

Это преобразование осуществляется в том случае, когда в качестве объявляемого типа функционального литерала выступает `PartialFunction`. Если объявляемый тип — просто `Function1` или не указан, функциональный литерал вместо этого преобразуется в *полноценную функцию*.

Вообще-то полноценными функциями нужно пробовать пользоваться везде, где только можно, поскольку использование частично примененных функций допускает возникновение ошибок времени выполнения, устранить которые компилятор вам не может помочь. Но иногда частично примененные функции приносят реальную пользу. Вам следует позаботиться о том, чтобы этим функциям не было предоставлено необрабатываемое значение. Как вариант, вы можете задействовать фреймворк, который допускает использование частично примененных функций и поэтому всегда перед вызовом функции выполняет проверку функцией `isDefinedAt`. Последнее проиллюстрировано приведенным ранее примером метода `receive`, где результатом выступает частично примененная функция с определением, данным в точности для тех сообщений, которые нужно обработать вызывающему коду.

Паттерны в выражениях for

Паттерны, как показано в листинге 15.18, можно использовать также в выражениях `for`. Это выражение извлекает все пары «ключ — значение» из отображения `capitals` (столицы). Каждая пара соответствует паттерну `(country, city)` (страна, город), который определяет две переменные: `country` и `city`.

Листинг 15.18. Выражение for с паттерном-кортежем

```
scala> for ((country, city) <- capitals)
         println("Столицей " + country + " является " + city)
Столицей France является Paris
Столицей Japan является Tokyo
```

Паттерн пар, показанный в данном листинге, интересен, поскольку сопоставление с ним никогда не даст сбой. Конечно, `capitals` выдает последовательность пар, следовательно, можно быть уверенными, что каждая сгенерированная пара может соответствовать паттерну пар.

Но с равной долей вероятности возможно, что паттерн не будет соответствовать сгенерированному значению. Именно такой случай показан в листинге 15.19.

Листинг 15.19. Отбор элементов списка, соответствующих паттерну

```
scala> val results = List(Some("apple"), None,
           Some("orange"))
results: List[Option[String]] = List(Some(apple), None,
Some(orange))

scala> for (Some(fruit) <- results) println(fruit)
apple
orange
```

В этом примере показано, что сгенерированные значения, не соответствующие паттерну, отбрасываются. Так, второй элемент `None` в получившемся списке не соответствует паттерну `Some(fruit)`, поэтому отсутствует в выводимой информации.

15.8. Большой пример

После изучения различных форм паттернов может быть интересно посмотреть на их применение в более существенном примере. Предлагаемая задача заключается в написании класса, форматирующего выражения, который выводит арифметическое выражение в двумерной разметке. Такое выражение деления, как `x / (x + 1)`, должно быть выведено вертикально — с числителем, показанным над знаменателем:

```
  x
-----
x + 1
```

В качестве еще одного примера ниже в двумерной разметке показано выражение `((a / (b * c) + 1 / n) / 3)`:

```
  a     1
----- + -
b * c   n
---------
    3
```

Исходя из этих примеров, можно прийти к выводу, что манипулированием разметкой должен заняться класс — назовем его `ExprFormatter`, поэтому имеет смысл задействовать библиотеку разметки, разработанную в главе 10. Мы также используем семейство `case`-классов `Expr`, ранее уже встречавшееся в данной главе, и поместим в именованные пакеты как библиотеку разметки из главы 10, так и средство форматирования выражений. Полный код этого примера будет показан в листингах 15.20 и 15.21.

Сначала полезно будет сосредоточиться на горизонтальной разметке. Структурированное выражение:

```
BinOp("+",
      BinOp("*",
            BinOp("+", Var("x"), Var("y")),
            Var("z")),
      Number(1))
```

должно привести к выводу `(x + y) * z + 1`. Обратите внимание на обязательность круглых скобок вокруг выражения `x + y` и их необязательность вокруг выражения `(x + y) * z`. Чтобы разметка получилась максимально разборчивой, следует стремиться к отказу от избыточных круглых скобок и обеспечить наличие всех обязательных.

Чтобы узнать, куда ставить круглые скобки, код должен быть в курсе относительной приоритетности каждого оператора; как следствие, неплохо было бы сначала отрегулировать именно этот вопрос. Относительную приоритетность можно выразить непосредственно в виде литерала отображения следующей формы:

```
Map(
  "|" -> 0, "||" -> 0,
  "&" -> 1, "&&" -> 1, ...
)
```

Конечно, вам потребуется предварительно выполнить определенный объем вычислительной работы для назначения приоритетов. Удобнее будет просто определить группы операторов с нарастающим уровнем приоритета, а затем, исходя из этого, вычислить приоритет каждого оператора. Соответствующий код показан в листинге 15.20.

Листинг 15.20. Верхняя половина средства форматирования выражений

```
package org.stairwaybook.expr
import org.stairwaybook.layout.Element.elem

sealed abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
    left: Expr, right: Expr) extends Expr

class ExprFormatter {

  // Содержит операторы в группах с нарастающей степенью приоритетности
  private val opGroups =
    Array(
      Set("|", "||"),
      Set("&", "&&"),
      Set("^"),
      Set("==", "!="),
      Set("<", "<=", ">", ">="),
      Set("+", "-"),
      Set("*", "%")
    )

  // Отображение операторов на их степень приоритетности
  private val precedence = {
    val assocs =
      for {
        i <- 0 until opGroups.length
        op <- opGroups(i)
      } yield op -> i
    assocs.toMap
  }
```

```
private val unaryPrecedence = opGroups.length
private val fractionPrecedence = -1

// продолжение в листинге 15.21...
```

Переменная `precedence` — отображение операторов на уровень их приоритета, представленный целыми числами, начинающимися с нуля. Приоритет вычисляется с использованием выражения `for` с двумя генераторами. Первый выдает каждый индекс `i` массива `opGroups`, второй — каждый оператор `op`, находящийся в `opGroups(i)`. Для каждого такого оператора выражение `for` выдает привязку оператора `op` к его индексу `i`. В результате приоритетность оператора берется из его относительной позиции в массиве.

Привязки записываются с помощью инфиксной стрелки, например `op -> i`. До сих пор привязки были показаны только как часть конструкций отображений, но они имеют значение и сами по себе. Фактически привязка `op -> i` есть не что иное, как пара `(op, i)`.

Теперь, зафиксировав уровень приоритета всех бинарных операторов, за исключением `/`, имеет смысл обобщить данную концепцию, охватив также унарные операторы. Уровень приоритета унарного оператора выше уровня приоритета любого бинарного оператора. Поэтому для переменной `unaryPrecedence`, показанной в листинге 15.20, устанавливается значение длины массива `opGroups` на единицу большее, чем уровень приоритета операторов `*` и `%`. Уровень приоритета оператора деления рассматривается не так, как параметр других операторов, поскольку для дробей используется вертикальная разметка. Но, конечно же, было бы удобно присвоить оператору деления специальное значение уровня приоритета `-1`, поэтому переменная `fractionPrecedence` будет инициализирована значением `-1`, как было показано в листинге 15.20 (см. выше).

После такой подготовительной работы можно приступать к написанию основного метода `format`. Этот метод получает два аргумента: выражение `e`, имеющее тип `Expr`, и уровень приоритета `enclPrec` того оператора, который непосредственно заключен в данное выражение. (Если в выражении нет никакого оператора, то значение `enclPrec` должно быть нулевым.) Метод выдает элемент разметки, представленный в виде двумерного массива символов. В листинге 15.21 показана остальная часть класса `ExprFormatter`, включающая три метода. Первый, `stripDot`, — вспомогательный. Далее идет приватный метод `format`, выполняющий основную работу по форматированию выражений. Последний метод, который также называется `format`, представляет собой единственный публичный метод в библиотеке, получающий выражение для форматирования. Приватный метод `format` проделывает свою работу, выполняя сопоставление с образцом по разновидностям выражения. У выражения `match` есть пять вариантов, каждый из которых будет рассмотрен отдельно.

Листинг 15.21. Нижняя половина средства форматирования выражений

```
// ...продолжение, начало в листинге 15.20
import org.stairwaybook.layout.Element
```

```
private def format(e: Expr, enclPrec: Int): Element =

  e match {

    case Var(name) =>
      elem(name)

    case Number(num) =>
      def stripDot(s: String) =
        if (s endsWith ".0") s.substring(0, s.length - 2)
        else s
      elem(stripDot(num.toString))

    case UnOp(op, arg) =>
      elem(op) beside format(arg, unaryPrecedence)

    case BinOp("/", left, right) =>
      val top = format(left, fractionPrecedence)
      val bot = format(right, fractionPrecedence)
      val line = elem('-', top.width max bot.width, 1)
      val frac = top above line above bot
      if (enclPrec != fractionPrecedence) frac
      else elem(" ") beside frac beside elem(" ")

    case BinOp(op, left, right) =>
      val opPrec = precedence(op)
      val l = format(left, opPrec)
      val r = format(right, opPrec + 1)
      val oper = l beside elem(" " + op + " ") beside r
      if (enclPrec <= opPrec) oper
      else elem("(") beside oper beside elem(")")
  }

  def format(e: Expr): Element = format(e, 0)
}
```

Первый вариант имеет следующий вид:

```
case Var(name) =>
  elem(name)
```

Если выражение является переменной, то результатом станет элемент, сформированный из имени переменной.

Второй вариант выглядит так:

```
case Number(num) =>
  def stripDot(s: String) =
    if (s endsWith ".0") s.substring(0, s.length - 2)
    else s
  elem(stripDot(num.toString))
```

Если выражение является числом, то результатом станет элемент, сформированный из значения числа. Функция `stripDot` очистит изображение числа с плавающей точкой, удалив из строки любой суффикс вида `".0"`.

А вот как выглядит третий вариант:

```
case UnOp(op, arg) =>
  elem(op) beside format(arg, unaryPrecedence)
```

Если выражение представляет собой унарную операцию `UnOp(op, arg)`, то результат будет сформирован из операции `op` и результата форматирования аргумента `arg` с самым высоким из возможных уровнем приоритета, имеющимся в данном окружении[1]. Это означает, что, если аргумент `arg` является бинарной операцией (но не делением), то всегда будет отображаться в круглых скобках.

Четвертый вариант представлен следующим кодом:

```
case BinOp("/", left, right) =>
  val top = format(left, fractionPrecedence)
  val bot = format(right, fractionPrecedence)
  val line = elem('-', top.width max bot.width, 1)
  val frac = top above line above bot
  if (enclPrec != fractionPrecedence) frac
  else elem(" ") beside frac beside elem(" ")
```

Если выражение имеет вид дроби, то промежуточный результат `frac` формируется путем помещения отформатированных операндов `left` и `right` друг над другом с разделительным элементом в виде горизонтальной линии. Ширина горизонтальной линии равна максимальной ширине отформатированных операндов. Промежуточный результат становится окончательным, если только дробь сама по себе не появляется в виде аргумента еще одной функции. В последнем случае по обе стороны `frac` добавляется по пробелу. Чтобы понять, зачем это делается, рассмотрим выражение `(a / b) / c`.

Без коррекции по ширине при форматировании этого выражения получится следующая картинка:

```
a
-
b
-
c
```

Вполне очевидна проблема с разметкой: непонятно, где именно находится дробная черта верхнего уровня. Показанное ранее выражение может означать либо `(a / b) / c`, либо `a / (b / c)`. Чтобы устранить неоднозначность, с обеих сторон разметки вложенной дроби `a / b` нужно добавить пробелы.

Тогда разметка станет однозначной:

```
 a
 -
 b
---
 c
```

[1] Значение unaryPrecedence является самым высоким приоритетом из возможных, поскольку ему было присвоено значение, на единицу превышающее значения приоритета операторов * и %.

Пятый и последний вариант выглядит следующим образом:

```
case BinOp(op, left, right) =>
  val opPrec = precedence(op)
  val l = format(left, opPrec)
  val r = format(right, opPrec + 1)
  val oper = l beside elem(" " + op + " ") beside r
  if (enclPrec <= opPrec) oper
  else elem("(") beside oper beside elem(")")
```

Этот вариант применяется ко всем остальным бинарным операциям. Он указан после варианта, который начинается со следующего кода:

```
case BinOp("/", left, right) => ...
```

поэтому понятно, что оператор `op` в паттерне `BinOp(op, left, right)` не может быть оператором деления. Чтобы форматировать такую бинарную операцию, нужно сначала отформатировать его операнды `left` и `right`. В качестве параметра уровня приоритета для форматирования левого операнда используется `opPrec` оператора `op`, а для правого операнда берется уровень на единицу больше. Вдобавок такая схема обеспечивает правильную ассоциативность, выраженную круглыми скобками.

Например, операция:

```
BinOp("-", Var("a"), BinOp("-", Var("b"), Var("c")))
```

будет вполне корректно выражена с применением круглых скобок в виде `a - (b - c)`. Затем с помощью выстраивания в линию операндов `left` и `right`, разделенных оператором, формируется промежуточный результат `oper`. Если уровень приоритета текущего оператора ниже уровня приоритета оператора, заключенного в скобки, то `oper` помещается между круглыми скобками, в противном случае возвращается в исходном виде.

На этом разработка приватной функции `format` завершена. Остается только публичный метод `format`, который позволяет программистам, создающим клиентский код, форматировать высокоуровневые выражения, не прибегая к передаче аргумента, содержащего уровень приоритета. Демонстрационная программа, с помощью которой проверяется `ExprFormatter`, показана в листинге 15.22.

Листинг 15.22. Приложение, выполняющее вывод отформатированных выражений

```
import org.stairwaybook.expr._

object Express extends App {

  val f = new ExprFormatter

  val e1 = BinOp("*", BinOp("/", Number(1), Number(2)),
                      BinOp("+", Var("x"), Number(1)))

  val e2 = BinOp("+", BinOp("/", Var("x"), Number(2)),
                      BinOp("/", Number(1.5), Var("x")))
```

```
  val e3 = BinOp("/", e1, e2)

  def show(e: Expr) = s"${println(f.format(e))}\n\n"

  for (e <- Array(e1, e2, e3)) show(e)
}
```

Обратите внимание: приложение вполне работоспособно даже при отсутствии в нем определения метода `main`, поскольку оно является наследником трейта `App`. Запустить программу `Express` можно командой:

```
scala Express
```

При этом будет получен следующий вывод:

```
    1
    - * (x + 1)
    2

  x   1.5
  - + ---
  2    x

1
- * (x + 1)
2
-----------
  x   1.5
  - + ---
  2    x
```

Резюме

В этой главе мы представили подробную информацию о `case`-классах и сопоставлении с образцом. Их применение позволяет получить преимущества в процессе использования ряда лаконичных идиом, которые обычно недоступны в объектно-ориентированных языках. Но реализованное в Scala сопоставление с образцом не ограничивается тем, что было показано в данной главе. Если нужно задействовать сопоставление с образцом, но при этом нежелательно открывать доступ к вашим классам, как делается в `case`-классах, то можно обратиться к *экстракторам*, рассматриваемым в главе 26. В следующей главе мы переключим свое внимание на списки.

16 Работа со списками

Вероятно, наиболее востребованные структуры данных в программах на Scala — списки. В этой главе мы подробно разъясним, что это такое. В ней мы представим много общих операций, которые могут выполняться над списками. Кроме того, раскроем некоторые наиболее важные принципы проектирования программ, работающих со списками.

16.1. Литералы списков

Списки уже попадались в предыдущих главах, следовательно, вам известно, что список, содержащий элементы `'a'`, `'b'` и `'c'`, записывается как `List('a', 'b', 'c')`. А вот другие примеры:

```
val fruit = List("apples", "oranges", "pears")
val nums = List(1, 2, 3, 4)
val diag3 =
  List(
    List(1, 0, 0),
    List(0, 1, 0),
    List(0, 0, 1)
  )
val empty = List()
```

Списки очень похожи на массивы, но имеют два важных отличия. Во-первых, списки являются неизменяемой структурой данных, то есть их элементы нельзя изменить путем присваивания. Во-вторых, списки имеют рекурсивную структуру (имеется в виду *связанный список*)[1], а у массивов она линейная.

[1] Графическое представление структуры списка типа List показано на рис. 22.2.

16.2. Тип List

Как и массивы, списки *однородны*: у всех элементов списка один и тот же тип. Тип списка, имеющего элементы типа `T`, записывается как `List[T]`. Например, далее показаны те же четыре списка с явным указанием типов:

```
val fruit: List[String] = List("apples", "oranges", "pears")
val nums: List[Int] = List(1, 2, 3, 4)
val diag3: List[List[Int]] =
  List(
    List(1, 0, 0),
    List(0, 1, 0),
    List(0, 0, 1)
  )
val empty: List[Nothing] = List()
```

В Scala тип списка обладает *ковариантностью*. Этот значит, для каждой пары типов `S` и `T`, если `S` — подтип `T`, то `List[S]` — подтип `List[T]`. Например, `List[String]` — подтип `List[Object]`. И это вполне естественно, поскольку каждый список строк также может рассматриваться как список объектов[1].

Обратите внимание: типом пустого списка является `List[Nothing]`. В разделе 11.3 уже было показано, что `Nothing` — это низший тип в иерархии классов Scala. Он является подтипом любого другого имеющегося в Scala типа. Списки ковариантны, отсюда следует, что `List[Nothing]` — подтип `List[T]` для любого типа `T`. Следовательно, пустой списочный объект, имеющий тип `List[Nothing]`, также может рассматриваться в качестве объекта любого другого списочного типа, имеющего вид `List[T]`. Именно поэтому вполне допустимо написать такой код:

```
// List() также относится к типу List[String]!
val xs: List[String] = List()
```

16.3. Создание списков

Все списки создаются из двух основных строительных блоков: `Nil` и `::` (произносится как «конс», от слова «конструировать»). `Nil` представляет собой пустой список. Инфиксный оператор конструирования `::` обозначает расширение списка с начала. То есть запись `x :: xs` представляет собой список, первым элементом которого является `x`, за которым следует список `xs` (его элементы). Следовательно, предыдущие списочные значения можно определить так:

```
val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
val nums  = 1 :: (2 :: (3 :: (4 :: Nil)))
```

[1] Более подробно ковариантность и другие разновидности вариантности рассмотрены в главе 19.

```
val diag3 = (1 :: (0 :: (0 :: Nil))) ::
            (0 :: (1 :: (0 :: Nil))) ::
            (0 :: (0 :: (1 :: Nil))) :: Nil
val empty = Nil
```

Фактически предыдущие определения `fruit`, `nums`, `diag3` и `empty`, выраженные в виде `List(...)`, всего лишь оболочки, которые разворачиваются в эти определения. Например, применение `List(1, 2, 3)` приводит к созданию списка `1 :: (2 :: (3 :: Nil))`.

То, что операция `::` заканчивается двоеточием, означает ее правую ассоциативность: `A :: B :: C` интерпретируется как `A :: (B :: C)`. Поэтому круглые скобки в предыдущих определениях можно отбросить. Например:

```
val nums = 1 :: 2 :: 3 :: 4 :: Nil
```

будет эквивалентом предыдущего определения `nums`.

16.4. Основные операции над списками

Все действия со списками можно свести к трем основным операциям:

- `head` возвращает первый элемент списка;
- `tail` возвращает список, состоящий из всех элементов, за исключением первого;
- `isEmpty` возвращает `true`, если список пуст.

Эти операции определены как методы класса `List`. Некоторые примеры их использования показаны в табл. 16.1. Методы `head` и `tail` определены только для непустых списков. Будучи примененными к пустому списку, они генерируют исключение:

```
scala> Nil.head
java.util.NoSuchElementException: head of empty list
```

Таблица 16.1. Основные операции над списками

Что используется	Что этот метод делает
empty.isEmpty	Возвращает true
fruit.isEmpty	Возвращает false
fruit.head	Возвращает "apples"
fruit.tail.head	Возвращает "oranges"
diag3.head	Возвращает List(1, 0, 0)

В качестве примера того, как можно обрабатывать список, рассмотрим сортировку элементов списка чисел в возрастающем порядке. Один из простых способов выполнить эту задачу заключается в *сортировке вставками*, которая работает так:

для сортировки непустого списка `x :: xs` сортируется остаток `xs` и первый элемент `x` вставляется в нужную позицию результата.

Сортировка пустого списка выдает пустой список. Выраженный в виде кода Scala, алгоритм сортировки вставками выглядит следующим образом:

```
def isort(xs: List[Int]): List[Int] =
  if (xs.isEmpty) Nil
  else insert(xs.head, isort(xs.tail))

def insert(x: Int, xs: List[Int]): List[Int] =
  if (xs.isEmpty || x <= xs.head) x :: xs
  else xs.head :: insert(x, xs.tail)
```

16.5. Паттерны-списки

Разбирать списки можно и с помощью сопоставления с образцом. Паттерны-списки по порядку следования соответствуют выражениям списков. Используя паттерн вида `List(...)`, можно либо сопоставить все элементы списка, либо разобрать список поэлементно, применив паттерны, составленные из оператора `::` и константы `Nil`.

Пример использования первой разновидности паттерна выглядит следующим образом:

```
scala> val List(a, b, c) = fruit
a: String = apples
b: String = oranges
c: String = pears
```

Паттерн `List(a, b, c)` соответствует спискам длиной три элемента и привязывает эти три элемента к паттернам-переменным `a`, `b` и `c`. Если количество элементов заранее не известно, то лучше вместо этого сопоставлять с помощью оператора `::`. Например, паттерн `a :: b :: rest` соответствует спискам длиной два и более элемента:

```
scala> val a :: b :: rest = fruit
a: String = apples
b: String = oranges
rest: List[String] = List(pears)
```

О сопоставлении с образцом объектов класса List

Если провести беглый обзор возможных форм паттернов, рассмотренных в главе 15, то выяснится, что ничего похожего ни на `List(...)`, ни на `::` в определенных там разновидностях нет. Фактически `List(...)` — экземпляр определенного в библиотеке паттерна-*экстрактора*. Такие паттерны будут рассматриваться в главе 26. Конспаттерн `x :: xs` — особый случай паттерна инфиксной операции. В качестве выражения инфиксная операция выступает эквивалентом вызова метода. Для паттернов

действуют иные правила: в качестве паттерна такая инфиксная операция, как `p op q`, является эквивалентом `op(p, q)`. То есть инфиксный оператор `op` рассматривается в качестве паттерна-конструктора. В частности, такой конс-паттерн, как `x :: xs`, рассматривается как `::(x, xs)`.

Это обстоятельство подсказывает, что должен быть класс по имени `::`, соответствующий паттерну-конструктору. Разумеется, такой класс существует — он называется `scala.::`, и это именно тот класс, который создает непустые списки. Следовательно, `::` в Scala фигурирует дважды: как имя класса в пакете `scala` и как метод класса `List`. Задача метода `::` — создать экземпляр класса `scala.::`. Более подробно реализация класса `List` рассматривается в главе 22.

Извлечение части списков с помощью паттернов — альтернатива использованию основных методов `head`, `tail` и `isEmpty`. Например, в коде ниже снова применена сортировка вставками, на этот раз записанная с сопоставлением с образцом:

```
def isort(xs: List[Int]): List[Int] = xs match {
  case List()   => List()
  case x :: xs1 => insert(x, isort(xs1))
}

def insert(x: Int, xs: List[Int]): List[Int] = xs match {
  case List()  => List(x)
  case y :: ys => if (x <= y) x :: xs
                  else y :: insert(x, ys)
}
```

Зачастую применение к спискам сопоставления с образцом оказывается более понятным, чем их декомпозиция с помощью методов, поэтому данное сопоставление должно стать частью вашего инструментария для обработки списков.

Вот и все, что нужно знать о списках в Scala, чтобы их правильно применять. Но существует также множество методов, которые вбирают в себя наиболее распространенные схемы проведения операций со списками. Эти методы делают программы обработки списков более лаконичными и зачастую более понятными. В следующих двух разделах мы представим наиболее важные методы, определенные в классе `List`.

16.6. Методы первого порядка класса List

В этом разделе мы рассмотрим большинство определенных в классе `List` методов *первого порядка*. Метод первого порядка не получает в качестве аргументов никаких функций. Мы также представим несколько рекомендованных приемов структурирования программ, работающих со списками, на двух примерах.

Конкатенация двух списков

Операцией, похожей на `::`, является конкатенация списков, записываемая в виде `:::`. В отличие от операции `::` операция `:::` получает в качестве операндов два списка. Результатом выполнения кода `xs ::: ys` выступает новый список, содержащий все элементы списка `xs`, за которыми следуют все элементы списка `ys`.

Рассмотрим несколько примеров:

```
scala> List(1, 2) ::: List(3, 4, 5)
res0: List[Int] = List(1, 2, 3, 4, 5)

scala> List() ::: List(1, 2, 3)
res1: List[Int] = List(1, 2, 3)

scala> List(1, 2, 3) ::: List(4)
res2: List[Int] = List(1, 2, 3, 4)
```

Как и конс-оператор, конкатенация списков правоассоциативна. Такое выражение, как:

```
xs ::: ys ::: zs
```

интерпретируется следующим образом:

```
xs ::: (ys ::: zs)
```

Принцип «разделяй и властвуй»

Конкатенация (`:::`) реализована в виде метода класса `List`. Можно было бы также реализовать конкатенацию «вручную», используя сопоставление с образцом для списков. Будет поучительно попробовать сделать это самостоятельно, поскольку таким образом можно проследить общий путь реализации алгоритмов с помощью списков. Сначала мы остановимся на сигнатуре метода конкатенации, который назовем `append`. Чтобы не создавать большой путаницы, предположим, что `append` определен за пределами класса `List`, поэтому будет получать в качестве параметров два конкатенируемых списка. Оба они должны быть согласованы по типу их элементов, но сам тип может быть произвольным. Все это можно обеспечить, задав `append` параметр типа[1], представляющего тип элементов двух исходных списков:

```
def append[T](xs: List[T], ys: List[T]): List[T]
```

Чтобы спроектировать реализацию `append`, имеет смысл вспомнить принцип «разделяй и властвуй» для программ, работающих с рекурсивными структурами

[1] Более подробно параметры типов будут рассмотрены в главе 19.

данных, такими как списки. Многие алгоритмы для работы со списками сначала разбивают исходный список на простые блоки, используя сопоставление с образцом. Это часть принципа, которая называется *«разделяй»*. Затем конструируется результат для каждого варианта. Если результатом является непустой список, то некоторые из его частей можно сконструировать с помощью рекурсивных вызовов того же самого алгоритма. В этом будет заключаться часть принципа, называемая *«властвуй»*.

Чтобы применить этот принцип к реализации метода `append`, сначала стоит ответить на вопрос: какой именно список нужно сопоставлять? Применительно к `append` данный вопрос не настолько прост, как для многих других методов, поскольку здесь имеется два варианта. Но следующая далее фаза «властвования» подсказывает: следует сконструировать список, состоящий из всех элементов обоих исходных списков. Списки конструируются из конца в начало, поэтому `ys` можно оставить нетронутым, а вот `xs` нужно разобрать и пристроить впереди `ys`. Таким образом, имеет смысл сконцентрироваться на `xs` как на источнике сопоставления с образцом. Самое частое сопоставление в отношении списков просто отличает пустой список от непустого. Итак, для метода `append` вырисовывается следующая схема:

```
def append[T](xs: List[T], ys: List[T]): List[T] =
  xs match {
    case List() => ???
    case x :: xs1 => ???
  }
```

Остается лишь заполнить два места, обозначенных `???`[1]. Первое такое место — альтернатива для случая, когда входной список `xs` пуст. В таком случае результатом конкатенации будет второй список:

```
case List() => ys
```

Второе место, оставленное незаполненным, — альтернатива для случая, когда входной список `xs` состоит из некоего `head`-элемента `x`, за которым следует остальная часть `xs1`. В таком случае результатом тоже будет непустой список. Чтобы сконструировать непустой список, нужно знать, какой должна быть его «голова» (head), а каким — «хвост» (tail). Вам известно, что первый элемент получающегося в результате списка — `x`. Что касается остальных элементов, то их можно вычислить, добавив второй список, `ys`, к оставшейся части первого списка, `xs1`.

Это завершает проектирование и дает:

```
def append[T](xs: List[T], ys: List[T]): List[T] =
  xs match {
    case List() => ys
    case x :: xs1 => x :: append(xs1, ys)
  }
```

[1] ??? — это метод, который генерирует ошибку scala.NotImplementedError и имеет результирующий тип Nothing. Он может применяться в качестве временной реализации в процессе разработки приложения.

Вычисление второй альтернативы — иллюстрация той части принципа, которая называется «властвуй»: сначала нужно продумать форму желаемого результата, затем вычислить отдельные части этой формы, используя где возможно рекурсивные вызовы алгоритма. И наконец, сконструировать из этих частей вывод.

Получение длины списка: length

Метод `length` вычисляет длину списка:

```
scala> List(1, 2, 3).length
res3: Int = 3
```

Определение длины списков, в отличие от массивов, — довольно затратная операция. Чтобы ее выполнить, нужно пройти по всему списку в поисках его конца, и на это затрачивается время, пропорциональное количеству элементов списка. Поэтому вряд ли имеет смысл заменять `xs.isEmpty` выражением `xs.length == 0`. Результаты будут получены одинаковые, но второе выражение будет выполняться медленнее, в частности, если список `xs` имеет бóльшую длину.

Обращение к концу списка: init и last

Вам уже известны основные операции `head` и `tail`, в результате выполнения которых извлекаются соответственно первый элемент списка и весь остальной список, за исключением первого элемента. У каждой из них есть обратная по смыслу операция: `last` возвращает последний элемент непустого списка, а `init` — список, состоящий из всех элементов, за исключением последнего:

```
scala> val abcde = List('a', 'b', 'c', 'd', 'e')
abcde: List[Char] = List(a, b, c, d, e)

scala> abcde.last
res4: Char = e

scala> abcde.init
res5: List[Char] = List(a, b, c, d)
```

Подобно методам `head` и `tail`, эти методы, примененные к пустому списку, генерируют исключение:

```
scala> List().init
java.lang.UnsupportedOperationException: Nil.init
      at scala.List.init(List.scala:544)
      at ...

scala> List().last
java.util.NoSuchElementException: Nil.last
      at scala.List.last(List.scala:563)
      at ...
```

В отличие от `head` и `tail`, на выполнение которых неизменно затрачивается одно и то же время, для вычисления результата методы `init` и `last` должны обойти весь список. То есть их выполнение требует времени, пропорционального длине списка.

> Неплохо было бы организовать ваши данные таким образом, чтобы основная часть обращений приходилась на головной, а не на последний элемент списка.

Реверсирование списков: reverse

Если в какой-то момент вычисления алгоритма требуется часто обращаться к концу списка, то порой более разумным будет сначала перестроить список в обратном порядке и работать уже с результатом. Получить такой список можно следующим образом:

```
scala> abcde.reverse
res6: List[Char] = List(e, d, c, b, a)
```

Как и все остальные операции со списками, `reverse` создает новый список, а не изменяет тот, с которым работает. Поскольку списки неизменяемы, сделать это все равно бы было невозможно. Чтобы убедиться в этом, проверьте, что исходный список `abcde` после операции `reverse` не изменился:

```
scala> abcde
res7: List[Char] = List(a, b, c, d, e)
```

Операции `reverse`, `init` и `last` подчиняются ряду законов, с помощью которых можно анализировать вычисления и упрощать программы.

1. Операция `reverse` является собственной инверсией:

   `xs.reverse.reverse` *равно* `xs`

2. Операция `reverse` превращает `init` в `tail`, а `last` в `head`, за исключением того, что все элементы стоят в обратном порядке:

   `xs.reverse.init` *равно* `xs.tail.reverse`
   `xs.reverse.tail` *равно* `xs.init.reverse`
   `xs.reverse.head` *равно* `xs.last`
   `xs.reverse.last` *равно* `xs.head`

Реверсирование можно реализовать, воспользовавшись конкатенацией (`:::`), как в следующем методе по имени `rev`:

```
def rev[T](xs: List[T]): List[T] = xs match {
  case List() => xs
  case x :: xs1 => rev(xs1) ::: List(x)
}
```

Но этот метод, вопреки предположениям, менее эффективен. Чтобы убедиться в высокой вычислительной сложности `rev`, представьте, будто список `xs` имеет

длину *n*. Обратите внимание: придется делать *n* рекурсивных вызовов `rev`. Каждый вызов, за исключением последнего, влечет за собой конкатенацию списков. На конкатенацию `xs ::: ys` затрачивается время, пропорциональное длине ее первого аргумента `xs`. Следовательно, общая вычислительная сложность `rev` выражается так:

```
n + (n - 1) + ... + 1 = (1 + n) * n / 2
```

Иными словами, `rev` имеет квадратичную вычислительную сложность по длине его входного аргумента. Если сравнить это обстоятельство со стандартным реверсированием изменяемого связного списка, имеющего линейную вычислительную сложность, то оно вызывает разочарование. Но данная реализация `rev` — не самая лучшая из возможных. Пример, который начинается на с. 312, позволит вам увидеть, как можно ускорить все это.

Префиксы и суффиксы: drop, take и splitAt

Операции `drop` и `take` обобщают `tail` и `init` в том смысле, что возвращают произвольные префиксы или суффиксы списка. Выражение `xs take` *n* возвращает первые *n* элементов списка `xs`. Если *n* больше `xs.length`, то возвращается весь список `xs`. Операция `xs drop` *n* возвращает все элементы списка `xs`, за исключением первых *n* элементов. Если *n* больше `xs.length`, то возвращается пустой список.

Операция `splitAt` разбивает список по заданному индексу, возвращая пару из двух списков[1]. Она определяется следующим равенством:

```
xs splitAt n равно (xs take n, xs drop n)
```

Но операция `splitAt` избегает двойного прохода по элементам списка. Примеры применения этих трех методов выглядят следующим образом:

```
scala> abcde take 2
res8: List[Char] = List(a, b)

scala> abcde drop 2
res9: List[Char] = List(c, d, e)

scala> abcde splitAt 2
res10: (List[Char], List[Char]) = (List(a, b),List(c, d, e))
```

Выбор элемента: apply и indices

Произвольный выбор элемента поддерживается методом `apply`, но эта операция менее востребована, чем аналогичная операция для массивов:

```
scala> abcde apply 2 // в Scala используется довольно редко
res11: Char = c
```

[1] Как уже упоминалось в разделе 10.12, понятие пары — неформальное название для Tuple2.

Что же касается всех остальных типов, то подразумевается, что `apply` вставляется в вызове метода, когда объект появляется в позиции функции. Поэтому показанную ранее строку кода можно сократить до следующей:

```
scala> abcde(2) // в Scala используется довольно редко
res12: Char = c
```

Одной из причин того, что выбор произвольного элемента менее популярен для списков, чем для массивов, является то, что на выполнение кода `xs(`*n*`)` затрачивается время, пропорциональное величине значения индекса *n*. Фактически метод `apply` определен сочетанием методов `drop` и `head`:

```
xs apply n равно (xs drop n).head
```

Из этого определения также становится понятно, что индексы списков, как и индексы массивов, задаются в диапазоне от 0 до длины списка минус 1. Метод `indices` возвращает список, состоящий из всех допустимых индексов заданного списка:

```
scala> abcde.indices
res13: scala.collection.immutable.Range
  = Range(0, 1, 2, 3, 4)
```

Линеаризация списка списков: flatten

Метод `flatten` принимает список списков и линеаризирует его в единый список:

```
scala> List(List(1, 2), List(3), List(), List(4, 5)).flatten
res14: List[Int] = List(1, 2, 3, 4, 5)
scala> fruit.map(_.toCharArray).flatten
res15: List[Char] = List(a, p, p, l, e, s, o, r, a, n, g, e,
s, p, e, a, r, s)
```

Он применим только к тем спискам, все элементы которых являются списками. Попытка линеаризации других списков выдаст ошибку компиляции:

```
scala> List(1, 2, 3).flatten
<console>:8: error: No implicit view available from Int =>
scala.collection.IterableOnce[B].
              List(1, 2, 3).flatten
                            ^
```

Объединение в пары: zip и unzip

Операция `zip` получает два списка и формирует список из пар их значений:

```
scala> abcde.indices zip abcde
res17: scala.collection.immutable.IndexedSeq[(Int, Char)] =
Vector((0,a), (1,b), (2,c), (3,d), (4,e))
```

Если списки разной длины, то все элементы без пары отбрасываются:

```
scala> val zipped = abcde zip List(1, 2, 3)
zipped: List[(Char, Int)] = List((a,1), (b,2), (c,3))
```

Особо полезен вариант объединения в пары списка с его индексами. Наиболее эффективно оно выполняется с помощью метода `zipWithIndex`, составляющего пары из каждого элемента списка и той позиции, в которой он появляется в этом списке.

```
scala> abcde.zipWithIndex
res18: List[(Char, Int)] = List((a,0), (b,1), (c,2), (d,3), (e,4))
```

Любой список кортежей можно превратить обратно в кортеж списков с помощью метода `unzip`:

```
scala> zipped.unzip
res19: (List[Char], List[Int])
  = (List(a, b, c),List(1, 2, 3))
```

Методы `zip` и `unzip` реализуют один из способов одновременной работы с несколькими списками. Более эффективный способ сделать то же самое показан в разделе 16.9.

Отображение списков: toString и mkString

Операция `toString` возвращает каноническое строковое представление списка:

```
scala> abcde.toString
res20: String = List(a, b, c, d, e)
```

Если требуется иное представление, то можно воспользоваться методом `mkString`. Операция `xs mkString (pre, sep, post)` задействует четыре операнда: отображаемый список `xs`, префиксную строку `pre`, отображаемую перед всеми элементами, строковый разделитель `sep`, отображаемый между последовательно выводимыми элементами, и постфиксную строку, отображаемую в конце.

Результатом операции будет следующая строка:

```
pre + xs(0) + sep + ...+ sep + xs(xs.length - 1) + post
```

У метода `mkString` имеются два перегруженных варианта, которые позволяют отбрасывать некоторые или все его аргументы. Первый вариант получает только строковый разделитель:

```
xs mkString sep равно xs mkString("", sep, "")
```

Второй вариант позволяет опустить все аргументы:

```
xs.mkString равно xs mkString ""
```

Рассмотрим несколько примеров:

```
scala> abcde.mkString("[", ",", "]")
res21: String = [a,b,c,d,e]
```

```
scala> abcde.mkString ""
res22: String = abcde

scala> abcde.mkString
res23: String = abcde

scala> abcde.mkString("List(", ", ", ")")
res24: String = List(a, b, c, d, e)
```

Есть также вариант `mkString`, называющийся `addString`; он не возвращает созданную строку в качестве результата, а добавляет ее к объекту `StringBuilder`[1]:

```
scala> val buf = new StringBuilder
buf: StringBuilder =

scala> abcde.addString(buf, "(", ";", ")")
res25: StringBuilder = (a;b;c;d;e)
```

Методы `mkString` и `addString` наследуются из супертрейта `Iterable` класса `List`, поэтому их можно применять ко всем другим коллекциям.

Преобразование списков: iterator, toArray, copyToArray

Чтобы выполнить преобразование данных между линейным миром массивов и рекурсивным миром списков, можно воспользоваться методом `toArray` в классе `List` и методом `toList` в классе `Array`:

```
scala> val arr = abcde.toArray
arr: Array[Char] = Array(a, b, c, d, e)

scala> arr.toList
res26: List[Char] = List(a, b, c, d, e)
```

Есть также метод `copyToArray`, который копирует элементы списка в последовательные позиции массива внутри некоего результирующего массива. Операция:

```
xs.copyToArray(arr, start)
```

копирует все элементы списка `xs` в массив `arr`, начиная с позиции `start`. Нужно обеспечить достаточную длину результирующего массива `arr`, чтобы в нем мог поместиться весь список. Рассмотрим пример:

```
scala> val arr2 = new Array[Int](10)
arr2: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

scala> List(1, 2, 3).copyToArray(arr2, 3)

scala> arr2
res28: Array[Int] = Array(0, 0, 0, 1, 2, 3, 0, 0, 0, 0)
```

[1] Имеется в виду класс scala.StringBuilder, а не java.lang.StringBuilder.

И наконец, если нужно получить доступ к элементам списка через итератор, то можно воспользоваться методом `iterator`:

```
scala> val it = abcde.iterator
it: Iterator[Char] = <iterator>

scala> it.next
res29: Char = a

scala> it.next
res30: Char = b
```

Пример: сортировка слиянием

Ранее представленная сортировка вставками записывается кратко, но эффективность ее невысока. Ее усредненная вычислительная сложность пропорциональна квадрату длины входного списка. Более эффективен алгоритм сортировки *слиянием*.

УСКОРЕННЫЙ РЕЖИМ ЧТЕНИЯ

> Этот пример — еще одна иллюстрация карринга и принципа «разделяй и властвуй». Кроме того, в нем говорится о вычислительной сложности алгоритма, что может оказаться полезным. Если же читать книгу нужно быстрее, то раздел 16.7 можно спокойно пропустить.

Сортировка слиянием работает следующим образом: если список имеет один элемент или не имеет никаких элементов, то он уже отсортирован, поэтому может быть возвращен в исходном виде. Более длинные списки разбиваются на два подсписка, в каждом из которых содержится около половины элементов исходного списка. Каждый подсписок сортируется с помощью рекурсивного вызова функции `sort`, затем два отсортированных списка объединяются в ходе операции слияния.

Чтобы выполнить обобщенную реализацию сортировки слиянием, вам придется оставить публичными тип элементов сортируемого списка и функцию, которая будет использоваться для сравнения элементов. Максимально обобщенную функцию можно получить, передав ей эти два элемента в качестве параметров. При этом получится реализация, показанная в листинге 16.1.

Листинг 16.1. Функция сортировки слиянием объектов List

```
def msort[T](less: (T, T) => Boolean)
    (xs: List[T]): List[T] = {

  def merge(xs: List[T], ys: List[T]): List[T] =
    (xs, ys) match {
      case (Nil, _) => ys
      case (_, Nil) => xs
      case (x :: xs1, y :: ys1) =>
```

```
        if (less(x, y)) x :: merge(xs1, ys)
        else y :: merge(xs, ys1)
    }

  val n = xs.length / 2
  if (n == 0) xs
  else {
    val (ys, zs) = xs splitAt n
    merge(msort(less)(ys), msort(less)(zs))
  }
}
```

Вычислительная сложность `msort` — $n \log(n)$, где n — длина входного списка. Чтобы понять причину происходящего, следует отметить: и разбиение списка на два подсписка, и слияние двух отсортированных списков, требует времени, которое пропорционально длине аргумента `list(s)`. Каждый рекурсивный вызов `msort` вполовину уменьшает количество элементов, используемых им в качестве входных данных. Поэтому производится примерно $\log(n)$ последовательных вызовов, выполняемых до тех пор, пока не будет достигнут базовый вариант для списков длиной 1. Но для более длинных списков каждый вызов порождает два последующих вызова. Если все это сложить вместе, получится, что на каждом уровне вызова $\log(n)$ каждый элемент исходных списков примет участие в одной операции разбиения и одной операции слияния.

Следовательно, каждый уровень вызова имеет общий уровень затрат, пропорциональный n. Поскольку число уровней вызова равно $\log(n)$, мы получаем общий уровень затрат, пропорциональный $n \log(n)$. Он не зависит от исходного распределения элементов в списке, следовательно, в наихудшем варианте будет таким же, как уровень затрат в усредненном. Это свойство делает сортировку слиянием весьма привлекательным алгоритмом для сортировки списков.

Пример использования `msort` выглядит следующим образом:

```
scala> msort((x: Int, y: Int) => x < y)(List(5, 7, 1, 3))
res31: List[Int] = List(1, 3, 5, 7)
```

Функция `msort` представляет собой классический образец каррингa, рассмотренный в разделе 9.3. Карринг упрощает специализацию функции для конкретных функций сравнения. Рассмотрим пример:

```
scala> val intSort = msort((x: Int, y: Int) => x < y) _
intSort: List[Int] => List[Int] = <function1>
```

Переменная `intSort` ссылается на функцию, которая получает список целочисленных значений и сортирует их в порядке следования чисел. Как объяснялось в разделе 8.6, знак подчеркивания означает недостающий список аргументов. В данном случае в качестве недостающего аргумента фигурирует сортируемый список. А вот другой пример, который демонстрирует способ возможного определения функции, выполняющей сортировку списка целочисленных значений в обратном порядке следования чисел:

```
scala> val reverseIntSort = msort((x: Int, y: Int) => x > y) _
reverseIntSort: (List[Int]) => List[Int] = <function>
```

Поскольку функция сравнения уже представлена с помощью карринга, при вызове функций `intSort` или `reverseIntSort` нужно будет только предоставить сортируемый список. Рассмотрим несколько примеров:

```
scala> val mixedInts = List(4, 1, 9, 0, 5, 8, 3, 6, 2, 7)
mixedInts: List[Int] = List(4, 1, 9, 0, 5, 8, 3, 6, 2, 7)

scala> intSort(mixedInts)
res0: List[Int] = List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> reverseIntSort(mixedInts)
res1: List[Int] = List(9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
```

16.7. Методы высшего порядка класса List

У многих операций над списками схожая структура. Раз за разом используются несколько схем. К подобным примерам можно отнести какое-либо преобразование каждого элемента списка; проверку того, что свойство соблюдается для всех элементов списка; извлечение из списка элементов, удовлетворяющих неким критериям; или объединение элементов списка с помощью того или иного оператора. В Java подобные схемы будут, как правило, созданы идиоматическими комбинациями циклов `for` или `while`. В Scala они могут быть выражены более коротко и непосредственно за счет использования операторов высшего порядка[1], которые реализуются в виде методов, определенных в классе `List`. Этим операторам высшего порядка и посвящен данный раздел.

Отображения списков: map, flatMap и foreach

Операция `xs map f` получает в качестве операндов список `xs` типа `List[T]` и функцию `f` типа `T => U`. Она возвращает список, получающийся в результате применения `f` к каждому элементу списка `xs`, например:

```
scala> List(1, 2, 3) map (_ + 1)
res32: List[Int] = List(2, 3, 4)

scala> val words = List("the", "quick", "brown", "fox")
words: List[String] = List(the, quick, brown, fox)

scala> words map (_.length)
res33: List[Int] = List(3, 5, 5, 3)

scala> words map (_.toList.reverse.mkString)
res34: List[String] = List(eht, kciuq, nworb, xof)
```

[1] Под операторами высшего порядка понимаются функции высшего порядка, используемые в системе записи операторов. Как упоминалось в разделе 9.1, функция является функцией высшего порядка, если получает в качестве параметров одну функцию и более.

Оператор `flatMap` похож на `map`, но в качестве правого операнда получает функцию, возвращающую список элементов. Он применяет функцию к каждому элементу списка и возвращает конкатенацию всех результатов выполнения функции. Разница между `map` и `flatMap` показана в следующем примере:

```
scala> words map (_.toList)
res35: List[List[Char]] = List(List(t, h, e), List(q, u, i,
    c, k), List(b, r, o, w, n), List(f, o, x))

scala> words flatMap (_.toList)
res36: List[Char] = List(t, h, e, q, u, i, c, k, b, r, o, w,
    n, f, o, x)
```

Как видите, там, где `map` возвращает список списков, `flatMap` возвращает единый список, в котором все элементы списков сконкатенированы.

Различия и взаимодействие методов `map` и `flatMap` показаны также в следующем выражении, с помощью которого создается список всех пар `(i, j)`, отвечающих условию $1 \le j < i < 5$:

```
scala> List.range(1, 5) flatMap (
         i => List.range(1, i) map (j => (i, j))
       )
res37: List[(Int, Int)] = List((2,1), (3,1), (3,2), (4,1),
    (4,2), (4,3))
```

Метод `List.range` является вспомогательным, создающим список из всех целых чисел в некотором диапазоне. В этом примере он используется дважды в целях создания списков целых чисел: в первый раз — списка целых чисел от 1 (включительно) до 5 (не включительно), во второй — списка целых чисел от 1 до `i` для каждого значения `i`, взятого из первого списка. Метод `map` в данном выражении создает список кортежей `(i, j)`, где `j < i`. Внешний метод `flatMap` в этом примере создает данный список для каждого `i` между 1 и 5, а затем конкатенирует все результаты. По-другому этот же список может быть создан с помощью выражения `for`:

```
for (i <- List.range(1, 5); j <- List.range(1, i)) yield (i, j)
```

Взаимодействие выражений `for` и операций со списками более подробно будет рассмотрено в главе 23.

Третья `map`-подобная операция — `foreach`. Но, в отличие от `map` и `flatMap`, она получает в качестве правого операнда процедуру (функцию, результирующим типом которой является `Unit`). Она просто применяет процедуру к каждому элементу списка. А сам результат операции также имеет тип `Unit`, то есть никакого результирующего списка не будет. В качестве примера рассмотрим краткий способ суммирования всех чисел списка:

```
scala> var sum = 0
sum: Int = 0

scala> List(1, 2, 3, 4, 5) foreach (sum += _)

scala> sum
res39: Int = 15
```

Фильтрация списков: filter, partition, find, takeWhile, dropWhile и span

Операция `xs filter p` получает в качестве операндов список `xs` типа `List[T]` и функцию-предикат `p` типа `T => Boolean`. Эта операция выдает список всех элементов `x` из списка `xs`, для которых `p(x)` вычисляется в `true`, например:

```
scala> List(1, 2, 3, 4, 5) filter (_ % 2 == 0)
res40: List[Int] = List(2, 4)

scala> words filter (_.length == 3)
res41: List[String] = List(the, fox)
```

Метод `partition` похож на метод `filter`, но возвращает пару списков. Один список содержит все элементы, для которых предикат вычисляется в `true`, а другой — все элементы, для которых предикат вычисляется в `false`. Он определяется равенством:

```
xs partition p равно (xs filter p, xs filter (!p(_)))
```

Пример его работы выглядит следующим образом:

```
scala> List(1, 2, 3, 4, 5) partition (_ % 2 == 0)
res42: (List[Int], List[Int]) = (List(2, 4),List(1, 3, 5))
```

Метод `find` тоже похож на метод `filter`, но возвращает только первый элемент, который удовлетворяет условию заданного предиката, а не все такие элементы. Операция `xs find p` получает в качестве операндов список `xs` и предикат `p`. Она возвращает `Option`. Если в списке `xs` есть элемент `x`, для которого `p(x)` вычисляется в `true`, то возвращается `Some(x)`. В противном случае `p` вычисляется в `false` для всех элементов и возвращается `None`. Вот несколько примеров работы этого метода:

```
scala> List(1, 2, 3, 4, 5) find (_ % 2 == 0)
res43: Option[Int] = Some(2)

scala> List(1, 2, 3, 4, 5) find (_ <= 0)
res44: Option[Int] = None
```

Операторы `takeWhile` и `dropWhile` также получают в качестве правого операнда предикат. Операция `xs takeWhile p` получает самый длинный префикс списка `xs`, в котором каждый элемент удовлетворяет условию предиката `p`. Аналогично этому операция `xs dropWhile p` удаляет самый длинный префикс из списка `xs`, в котором каждый элемент удовлетворяет условию предиката `p`. Ряд примеров использования этих методов выглядит следующим образом:

```
scala> List(1, 2, 3, -4, 5) takeWhile (_ > 0)
res45: List[Int] = List(1, 2, 3)

scala> words dropWhile (_ startsWith "t")
res46: List[String] = List(quick, brown, fox)
```

Метод `span` объединяет `takeWhile` и `dropWhile` в одну операцию точно так же, как метод `splitAt` объединяет `stake` и `drop`. Он возвращает пару из двух списков, определяемых следующим равенством:

```
xs span p равно (xs takeWhile p, xs dropWhile p)
```

Как и `splitAt`, метод `span` избегает двойного прохода элементов списка:

```
scala> List(1, 2, 3, -4, 5) span (_ > 0)
res47: (List[Int], List[Int]) = (List(1, 2, 3),List(-4, 5))
```

Применение предикатов к спискам: forall и exists

Операция `xs forall p` получает в качестве аргументов список `xs` и предикат `p`. Она возвращает результат `true`, если все элементы списка удовлетворяют условию предиката `p`. Напротив, операция `xs exists p` возвращает `true`, если в `xs` есть хотя бы один элемент, удовлетворяющий условию предиката `p`. Например, чтобы определить наличие в матрице, представленной списком списков, строки, состоящей только из нулевых элементов, можно применить следующий код:

```
scala> def hasZeroRow(m: List[List[Int]]) =
         m exists (row => row forall (_ == 0))
hasZeroRow: (m: List[List[Int]])Boolean

scala> hasZeroRow(diag3)
res48: Boolean = false
```

Свертка списков: foldLeft и foldRight

Еще один распространенный вид операции объединяет элементы списка с помощью оператора, например:

```
sum(List(a, b, c)) равно 0 + a + b + c
```

Это особый случай операции свертки:

```
scala> def sum(xs: List[Int]): Int = xs.foldLeft(0)(_ + _)
sum: (xs: List[Int])Int
```

Аналогично этому:

```
product(List(a, b, c)) равно 1 * a * b * c
```

представляет собой особый случай этой операции свертки:

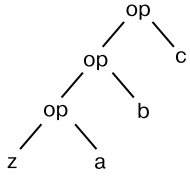
```
scala> def product(xs: List[Int]): Int = xs.foldLeft(1)(_ * _)
product: (xs: List[Int])Int
```

Операция левой свертки `xs.foldLeft(z)(op)` задействует три объекта: начальное значение `z`, список `xs` и бинарную операцию `op`. Результат свертки — применение `op`

между последовательно извлекаемыми элементами списка, где в качестве префикса выступает значение `z`, например:

`List(a, b, c).foldLeft(z)(op)` *равно* `op(op(op(z, a), b), c)`

Или в графическом представлении:



Вот еще один пример, иллюстрирующий использование операции `foldLeft`. Чтобы объединить все слова в списке из строковых значений с пробелами между ними и пробелом в самом начале списка, можно задействовать следующий код:

```
scala> words.foldLeft("")(_ + " " + _)
res49: String = " the quick brown fox"
```

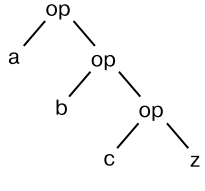
Этот код выдаст лишний пробел в самом начале. Избавиться от него можно с помощью слегка видоизмененного варианта кода:

```
scala> words.tail.foldLeft(words.head)(_ + " " + _)
res50: String = the quick brown fox
```

Операция `foldLeft` создает деревья операций с уклоном влево. По аналогии с этим оператор `foldRight` создает деревья операций с уклоном вправо, например:

`List(a, b, c).foldRight(z)(op)` *равно* `op(a, op(b, op(c, z)))`

Или в графическом представлении:



Для ассоциативных операций левая и правая свертки абсолютно эквивалентны, но эффективность их применения может существенно различаться. Рассмотрим, к примеру, операцию, соответствующую методу `flatten`, которая конкатенирует все элементы в список списков. Она может быть реализована с помощью либо левой, либо правой свертки:

```
def flattenLeft[T](xss: List[List[T]]) =
    xss.foldLeft(List[T]())(_ ::: _)

def flattenRight[T](xss: List[List[T]]) =
    xss.foldRight(List[T]())(_ ::: _)
```

Поскольку конкатенация списков `xs ::: ys` занимает время, пропорциональное длине его первого аргумента `xs`, то реализация в понятиях правой свертки в `flattenRight` более эффективна, чем реализация с применением левой свертки в `flattenLeft`. Дело в том, что `flattenLeft(xss)` копирует первый элемент списка `xss.head` $n - 1$ раз, где n — длина списка `xss`.

Учтите, что обе версии `flatten` требуют аннотации типа в отношении пустого списка, который является начальным значением свертки. Это связано с ограничениями в имеющемся в Scala механизме вывода типов, который не в состоянии автоматически вывести правильный тип списка. При попытке игнорировать аннотацию будет выдано такое сообщение об ошибке:

```
scala> def flattenRight[T](xss: List[List[T]]) =
           xss.foldRight(List())(_ ::: _)
<console>:8: error: type mismatch;
 found   : List[T]
 required: List[Nothing]
           xss.foldRight(List())(_ ::: _)
                                 ^
```

Чтобы понять, почему не был выполнен надлежащий вывод типа, нужно узнать о типах методов свертки и способах их реализации. Более подробно этот вопрос рассматривается в разделе 16.10.

Пример: реверсирование списков с помощью свертки

Ранее в этой главе была показана реализация метода реверсирования по имени `rev`, время работы которой имело квадратичную вычислительную сложность по длине реверсируемого списка. Теперь будет показана другая реализация метода реверсирования с линейной вычислительной сложностью. Идея состоит в том, чтобы воспользоваться операцией левой свертки, основанной на следующей схеме:

```
def reverseLeft[T](xs: List[T]) =
  xs.foldLeft(стартовое_значение)(операция)
```

Остается заполнить части *стартовое_значение* и *операция*. Собственно, вы можете попытаться вывести эти части из нескольких простых примеров. Чтобы правильно вывести *стартовое_значение*, можно начать с наименьшего потенциального списка, `List()`, и рассуждать следующим образом:

```
List()
  равен (в понятиях свойств reverseLeft)

reverseLeft(List())
  равен (по схеме для reverseLeft)
```

```
List().foldLeft(стартовое_значение)(операция)
  равен (по определению foldLeft)

стартовое_значение
```

Следовательно, *стартовое_значение* должно быть `List()`. Чтобы вывести второй операнд, в качестве примера можно взять следующий наименьший список. Поскольку уже известно, что *стартовое_значение* — это `List()`, можно рассуждать так:

```
List(x)
  равен (в понятиях свойств reverseLeft)

reverseLeft(List(x))
  равен (по схеме для reverseLeft со стартовым_значением = List())

List(x).foldLeft(List())(операция)
  равен (по определению foldLeft)

операция (List(), x)
```

Следовательно, *операция*`(List(), x)` — эквивалент `List(x)`, что можно записать также в виде `x :: List()`. Это наводит на такую мысль: нужно взять в качестве операции оператор `::` с его операндами, которые поменяли местами. (Иногда такую операцию называют «снок», ссылаясь на операцию `::`, которую называют «конс».) И тогда мы приходим к следующей реализации метода `reverseLeft`:

```
def reverseLeft[T](xs: List[T]) =
  xs.foldLeft(List[T]()) { (ys, y) => y :: ys }
```

Чтобы заставить работать механизм вывода типов, здесь также в качестве аннотации типа требуется использовать код `List[T]()`. Если проанализировать вычислительную сложность `reverseLeft`, то можно прийти к выводу, что в нем n раз применяется постоянная по времени выполнения операция («снок»), где n — длина списка-аргумента. Таким образом, вычислительная сложность `reverseLeft` линейна.

Сортировка списков: sortWith

Операция `xs sortWith before`, где `xs` — это список, а `before` — функция, которая может использоваться для сравнения двух элементов, выполняет сортировку элементов списка `xs`. Выражение `x before y` должно возвращать `true`, если в желаемом порядке следования `x` должен стоять перед `y`, например:

```
scala> List(1, -3, 4, 2, 6) sortWith (_ < _)
res51: List[Int] = List(-3, 1, 2, 4, 6)

scala> words sortWith (_.length > _.length)
res52: List[String] = List(quick, brown, the, fox)
```

Обратите внимание: `sortWith` выполняет сортировку слиянием подобно тому, как это делает алгоритм `msort`, показанный в последнем разделе. Но `sortWith` является методом класса `List`, а `msort` определен вне списков.

16.8. Методы объекта List

До сих пор все показанные в этой главе операции реализовывались в качестве методов класса `List`, поэтому вызывались в отношении отдельно взятых списочных объектов. Существует также ряд методов в глобально доступном объекте `scala.List`, который является объектом-компаньоном класса `List`. Одни такие операции — это фабричные методы, создающие списки. Другие же — это операции, работающие со списками некоторых конкретных видов. В этом разделе будут представлены обе разновидности методов.

Создание списков из их элементов: List.apply

В книге уже несколько раз попадались литералы списков вида `List(1, 2, 3)`. В их синтаксисе нет ничего особенного. Литерал вида `List(1, 2, 3)` — простое применение объекта `List` к элементам `1`, `2`, `3`. То есть это эквивалент кода `List.apply(1, 2, 3)`:

```
scala> List.apply(1, 2, 3)
res53: List[Int] = List(1, 2, 3)
```

Создание диапазона чисел: List.range

Метод `range`, который ранее подробно рассматривался при изучении методов `map` и `flatmap`, создает список, состоящий из диапазона чисел. Его самая простая форма, при которой создаются все числа, начиная с `from` и заканчивая `until` минус 1, — `List.range(from, until)`. Следовательно, последнее значение, `until`, в диапазон не входит.

Существует также версия `range`, получающая в качестве третьего параметра значение `step`. В результате выполнения этой операции получится список элементов, которые следуют друг за другом с указанным шагом, начиная с `from`. Указываемый шаг `step` может иметь положительное или отрицательное значение:

```
scala> List.range(1, 5)
res54: List[Int] = List(1, 2, 3, 4)

scala> List.range(1, 9, 2)
res55: List[Int] = List(1, 3, 5, 7)

scala> List.range(9, 1, -3)
res56: List[Int] = List(9, 6, 3)
```

Создание единообразных списков: List.fill

Метод `fill` создает список, состоящий из нуля или более копий одного и того же элемента. Он получает два параметра: длину создаваемого списка и повторяемый элемент. Каждый параметр задается в отдельном списке:

```
scala> List.fill(5)('a')
res57: List[Char] = List(a, a, a, a, a)

scala> List.fill(3)("hello")
res58: List[String] = List(hello, hello, hello)
```

Если методу `fill` дать более двух аргументов, то он будет создавать многомерные списки, то есть списки списков, списки списков из списков и т. д. Дополнительный аргумент помещается в первый список аргументов.

```
scala> List.fill(2, 3)('b')
res59: List[List[Char]] = List(List(b, b, b), List(b, b, b))
```

Табулирование функции: List.tabulate

Метод `tabulate` создает список, элементы которого вычисляются согласно предоставляемой функции. Аргументы у него такие же, как и у метода `List.fill`: в первом списке аргументов задается размерность создаваемого списка, а во втором дается описание элементов списка. Единственное отличие — элементы не фиксируются, а вычисляются из функции:

```
scala> val squares = List.tabulate(5)(n => n * n)
squares: List[Int] = List(0, 1, 4, 9, 16)
scala> val multiplication = List.tabulate(5,5)(_ * _)
multiplication: List[List[Int]] = List(List(0, 0, 0, 0, 0),
    List(0, 1, 2, 3, 4), List(0, 2, 4, 6, 8),
    List(0, 3, 6, 9, 12), List(0, 4, 8, 12, 16))
```

Конкатенация нескольких списков: List.concat

Метод `concat` объединяет несколько списков элементов. Конкатенируемые списки предоставляются `concat` в виде непосредственных аргументов:

```
scala> List.concat(List('a', 'b'), List('c'))
res60: List[Char] = List(a, b, c)

scala> List.concat(List(), List('b'), List('c'))
res61: List[Char] = List(b, c)

scala> List.concat()
res62: List[Nothing] = List()
```

16.9. Совместная обработка нескольких списков

Вы уже знакомы с методом zip, который создает список пар из двух списков, позволяя работать с ними одновременно:

```
scala> (List(10, 20) zip List(3, 4, 5)).
         map { case (x, y) => x * y }
res63: List[Int] = List(30, 80)
```

Метод `map`, примененный к спискам, прошедшим через `zip`, перебирает не отдельные элементы, а их пары. Первая пара содержит элементы, идущие первыми в каждом списке, вторая — элементы, идущие вторыми, и т. д. Количество пар определяется длиной списков. Обратите внимание: третий элемент второго списка отбрасывается. Метод `zip` объединяет только то количество элементов, которое совместно появляется во всех списках. Любые лишние элементы в конце отбрасываются.

Один из недостатков работы с несколькими списками с помощью метода `zip` состоит в том, что мы получаем промежуточный список (после вызова `zip`), который в конечном счете отбрасывается (при вызове метода `map`). Создание этого промежуточного списка может потребовать существенных расходов, если у него много элементов. Еще один недостаток этого подхода — функция, которая передается методу `map`, принимает в качестве параметра кортеж, что исключает использование синтаксиса заместителей, продемонстрированного в разделе 8.5.

Эти две проблемы решает метод `lazyZip`. По своему синтаксису он похож на `zip`:

```
scala> (List(10, 20) lazyZip List(3, 4, 5)).map(_ * _)
res63: List[Int] = List(30, 80)
```

Разница между `lazyZip` и `zip` в том, что первый не возвращает коллекцию сразу (отсюда и префикс lazy — «ленивый»). Вместо этого вы получаете значение, предоставляющее методы (включая `map`) для работы с двумя списками, для которых метод `zip` выполнен отложенно. В приведенном выше примере вы можете видеть, как метод `map` принимает функцию с двумя параметрами (вместо одной пары), позволяя нам использовать синтаксис заместителей.

Существуют также обобщающие аналоги для методов `exists` и `forall`. Они похожи на версии этих методов, предназначенные для работы с одним списком, но оперируют элементами не одного, а нескольких списков:

```
scala> (List("abc", "de") lazyZip List(3, 2)).
         forall(_.length == _)
res64: Boolean = true
scala> (List("abc", "de") lazyZip List(3, 2)).
         exists(_.length != _)
res65: Boolean = false
```

УСКОРЕННЫЙ РЕЖИМ ЧТЕНИЯ

В последнем разделе этой главы дается информация об имеющемся в Scala алгоритме вывода типов. Если такие подробности сейчас вас не интересуют, то можете пропустить весь раздел и сразу перейти к резюме.

16.10. Осмысление имеющегося в Scala алгоритма вывода типов

Одно из отличий предыдущего использования `sortWith` и `msort` касается допустимых синтаксических форм функции сравнения.

Сравните этот диалог с интерпретатором:

```
scala> msort((x: Char, y: Char) => x > y)(abcde)
res66: List[Char] = List(e, d, c, b, a)
```

со следующим:

```
scala> abcde sortWith (_ > _)
res67: List[Char] = List(e, d, c, b, a)
```

Эти два выражения эквивалентны, но в первом используется более длинная форма функции сравнения с именованными параметрами и явно заданными типами. Во втором задействована более краткая форма, `(_ > _)`, в которой вместо именованных параметров стоят знаки подчеркивания. Разумеется, с методом `sortWith` вы можете применить также первую, более длинную форму сравнения.

А вот с `msort` более краткая форма использоваться не может:

```
scala> msort(_ > _)(abcde)
<console>:12: error: missing parameter type for expanded
function ((x$1, x$2) => x$1.$greater(x$2))
       msort(_ > _)(abcde)
             ^
```

Чтобы понять, почему именно так происходит, следует знать некоторые подробности имеющегося в Scala алгоритма вывода типов. Это поточный механизм. При использовании метода `m(args)` механизм вывода типов сначала проверяет, имеется ли известный тип у метода `m`. Если да, то именно он и применяется для вывода ожидаемого типа аргументов. Например, в выражении `abcde.sortWith(_ > _)` типом `abcde` является `List[Char]`. Таким образом, `sortWith` известен как метод, получающий аргумент типа `(Char, Char) => Boolean` и выдающий результат типа `List[Char]`. Поскольку типы параметров аргументов функции известны, то их не нужно записывать явным образом. По совокупности всего известного о методе `sortWith` механизм вывода типов может установить, что код `(_ > _)` нужно раскрыть в `((x: Char, y: Char) => x > y)`, где `x` и `y` — некие произвольные только что полученные имена.

Теперь рассмотрим второй вариант, `msort(_ > _)(abcde)`. Типом `msort` является каррированный полиморфный тип метода, который принимает аргумент типа `(T, T) => Boolean` в функцию из `List[T]` в `List[T]`, *где `T` — некий пока еще неизвестный тип*. Прежде чем он будет применен к своим аргументам, у метода `msort` должен быть создан экземпляр с параметром типа.

Точный тип экземпляра `msort` в приложении еще неизвестен, поэтому он не может быть использован для вывода типа своего первого аргумента. В этом случае механизм вывода типов меняет свою стратегию: сначала он проверяет тип аргументов метода для определения экземпляра метода с подходящим типом. Но, когда перед

ним стоит задача проверки типа функционального литерала в краткой форме записи, (_ > _), он дает сбой из-за отсутствия информации о неявных типах заданных параметров функции, показанных знаками подчеркивания.

Один из способов решить проблему — передать msort явно заданный тип параметра:

```
scala> msort[Char](_ > _)(abcde)
res68: List[Char] = List(e, d, c, b, a)
```

Экземпляр msort подходящего типа теперь известен, поэтому его можно использовать для вывода типов аргументов. Еще одним потенциальным решением может стать перезапись метода msort таким образом, чтобы его параметры поменялись местами:

```
def msortSwapped[T](xs: List[T])(less:
    (T, T) => Boolean): List[T] = {

  // Та же реализация, что и у msort,
  // но с аргументами, которые поменялись местами
}
```

Теперь вывод типов будет выполнен успешно:

```
scala> msortSwapped(abcde)(_ > _)
res69: List[Char] = List(e, d, c, b, a)
```

Получилось так, что механизм вывода типов воспользовался известным типом первого параметра abcde, чтобы определить параметр типа метода msortSwapped. Точный тип msortSwapped был известен, поэтому с его помощью может быть выведен тип второго параметра, (_ > _).

В общем, когда ставится задача вывести параметры типа полиморфного метода, механизм вывода типов принимает во внимание типы всех значений аргументов в первом списке параметров, игнорируя все аргументы, кроме этих. Поскольку msortSwapped — каррированный метод с двумя списками параметров, то не нужно обращать внимание на второй аргумент (то есть на функциональное значение), чтобы определить параметр типа метода.

Эта схема вывода типов предлагает следующий принцип разработки библиотек: при разработке полиморфного метода, который получает некие нефункциональные аргументы и функциональный аргумент, этот функциональный аргумент в самом каррированном списке параметров нужно поставить на последнее место. Тогда экземпляр метода подходящего типа можно вывести из нефункциональных аргументов, и этот тип, в свою очередь, можно использовать для проверки типа функционального аргумента. Совокупный эффект будет состоять в том, что пользователи метода получат возможность предоставить меньший объем информации и написания функциональных литералов более компактными способами.

Теперь рассмотрим более сложный случай, касающийся операции *свертки*. Почему необходимо явно указывать параметр типа в выражении, подобном телу метода flattenRight, показанного на с. 312?

```
xss.foldRight(List[T]())(_ ::: _)
```

Тип метода `flattenRight` полиморфен в двух переменных типа. Если взять выражение:

```
xs.foldRight(z)(op)
```

то типом `xs` должен быть список какого-то произвольного типа `A`, скажем `xs: List[A]`. Начальное значение `z` может быть какого-нибудь другого типа `B`. Тогда операция `op` должна получать два аргумента типа, `A` и `B`, и возвращать результат типа `B`, то есть `op: (A, B) => B`. Тип значения `z` не связан с типом списка `xs`, поэтому у механизма вывода типов нет контекстной информации для `z`.

Теперь рассмотрим выражение в ошибочной версии метода `flattenRight`, также показанного на с. 312:

```
xss.foldRight(List())(_ ::: _) // этот код не пройдет компиляцию
```

Начальное значение `z` в данной свертке — пустой список, `List()`, следовательно, при отсутствии дополнительной информации о типе его тип будет выведен как `List[Nothing]`. Исходя из этого, механизм вывода типов установит, что типом `B` в свертке будет являться `List[Nothing]`. Таким образом, для операции `(_ ::: _)` в свертке будет ожидаться следующий тип:

```
(List[T], List[Nothing]) => List[Nothing]
```

Конечно же, такой тип возможен для операции в данной свертке, но пользы от него никакой! Он сообщает, что операция всегда получает в качестве второго аргумента пустой список и всегда в качестве результата выдает также пустой список.

Иными словами, вопрос вывода типов на основе `List()` был решен слишком рано — он должен был выждать, пока не станет виден тип операции `op`. Следовательно, весьма полезное в иных случаях правило о том, что для определения типа метода нужно принимать во внимание только первый список аргументов, будучи примененным к каррированному методу, становится камнем преткновения. В то же время, даже если бы это правило было смягчено, механизм вывода типов все равно не смог бы определиться с типом для операции `op`, поскольку ее типы параметров не приведены. Таким образом, создается ситуация, как в уловке-22, которую можно разрешить с помощью явной аннотации типа, получаемой от программиста[1].

Данный пример выявляет ряд ограничений локальной, поточной схемы вывода типов, имеющейся в Scala. В более глобальном механизме вывода типов в стиле Хиндли — Милнера (Hindley — Milner), используемом в таких функциональных языках, как ML или Haskell, подобных ограничений нет. Но по сравнению со стилем Хиндли — Милнера имеющийся в Scala механизм вывода типов обходится с объектно-ориентированной системой подтипов намного изящнее. К счастью, ограничения проявляются только в некоторых крайних случаях, и обычно их без особого труда можно обойти, добавив явную аннотацию типа.

---

[1] Уловка-22 (англ. Catch-22) — ситуация, возникающая в результате логического парадокса между взаимоисключающими правилами и процедурами. В этой ситуации индивид, подпадающий под действие таких норм, не может их никак контролировать, так как попытка нарушить эти установки автоматически подразумевает их соблюдение.

Добавление аннотаций типа пригодится также при отладке, когда вас поставят в тупик сообщения об ошибках типа, связанных с полиморфными методами. Если вы не уверены в причине возникновения конкретной ошибки типа, то нужно просто добавить некоторые аргументы типа или другие аннотации типа, в правильности которых вы не сомневаетесь. Тогда можно будет быстро понять, где реальный источник проблемы.

Резюме

В этой главе мы показали множество способов работы со списками. Рассмотрели основные операции, такие как `head` и `tail`; операции первого порядка, такие как `reverse`; операции высшего порядка, такие как `map`; и вдобавок полезные методы, определенные в объекте `List`. Попутно вы изучили принципы работы имеющегося в Scala механизма вывода типов.

Списки в Scala — настоящая рабочая лошадка, поэтому, узнав, как с ними работать, вы сможете извлечь для себя немалую выгоду. Именно с этой целью мы в данной главе погрузились в способы применения списков. Но списки — всего лишь одна из разновидностей коллекций, поддерживаемых в Scala. Тематика следующей главы будет скорее охватывающей, чем углубленной. В ней мы покажем вам порядок использования различных типов коллекций Scala.

17 Работа с другими коллекциями

В Scala содержится весьма богатая библиотека коллекций. В этой главе мы расскажем о наиболее часто используемых типах коллекций и операциях над ними. Более полный обзор доступных коллекций мы представим в главе 24, а в главе 25 покажем, как композиционные конструкции Scala применяются для предоставления насыщенного API.

17.1. Последовательности

Типы последовательностей позволяют работать с группами данных, выстроенных по порядку. Поскольку элементы упорядочены, то можно запрашивать первый элемент, второй, сто третий и т. д. В этом разделе мы бегло пройдемся по наиболее важным последовательностям.

Списки

Возможно, самым важным типом последовательности, о котором следует знать, является класс `List` — неизменяемый связный список, подробно рассмотренный в предыдущей главе. Списки поддерживают быстрое добавление и удаление элементов в начало списка, но не позволяют получить быстрый доступ к произвольным индексам, поскольку, чтобы это реализовать, требуется выполнить последовательный обход всех элементов списка.

Такое сочетание свойств может показаться странным, но оно попало в золотую середину и неплохо функционирует во многих алгоритмах. Как следует из описаний, представленных в главе 15, быстрое добавление и удаление начальных элементов означает хорошую работу сопоставления с образцом. Неизменяемость списков помогает разрабатывать корректные эффективные алгоритмы, поскольку избавляет от необходимости создавать копии списков.

Краткий пример, показывающий способ инициализации списка и получения доступа к его голове и хвосту, выглядит так:

```
scala> val colors = List("red", "blue", "green")
colors: List[String] = List(red, blue, green)

scala> colors.head
res0: String = red

scala> colors.tail
res1: List[String] = List(blue, green)
```

Чтобы освежить в памяти сведения о списках, обратитесь к шагу 8 в главе 3. А подробности использования списков можно найти в главе 16. Списки будут рассматриваться также в главе 22, которая дает представление о том, как именно они реализованы в Scala.

Массивы

Массивы позволяют хранить последовательность элементов и оперативно обращаться к элементу, находящемуся в произвольной позиции, чтобы либо получить его, либо обновить; для этого используется индекс, отсчитываемый от нуля. Массив известной длины, для которого пока неизвестны значения элементов, создается следующим образом:

```
scala> val fiveInts = new Array[Int](5)
fiveInts: Array[Int] = Array(0, 0, 0, 0, 0)
```

А вот как инициализируется массив, когда значения элементов известны:

```
scala> val fiveToOne = Array(5, 4, 3, 2, 1)
fiveToOne: Array[Int] = Array(5, 4, 3, 2, 1)
```

Как уже упоминалось, получить доступ к элементам массивов в Scala можно, указав индекс в круглых, а не в квадратных, как в Java, скобках. Рассмотрим пример доступа к элементу массива и обновления элемента:

```
scala> fiveInts(0) = fiveToOne(4)

scala> fiveInts
res3: Array[Int] = Array(1, 0, 0, 0, 0)
```

Массивы в Scala представлены точно так же, как массивы в Java. Поэтому можно абсолютно свободно использовать имеющиеся в Java методы, возвращающие массивы[1].

В предыдущих главах действия с массивами встречались уже много раз. Основы этих действий были рассмотрены в шаге 7 главы 3. Ряд примеров поэлементного

[1] Разница вариантности массивов в Scala и в Java — то есть является ли Array[String] подтипом Array[AnyRef — будет рассмотрена в разделе 19.3.

обхода массивов с помощью выражения `for` был показан в разделе 7.3. Массивы также занимают видное место в библиотеке двумерной разметки, представленной в главе 10.

Буферы списков

Класс `List` предоставляет быстрый доступ к голове и к хвосту списка, но не к его концу. Таким образом, при необходимости построить список с добавлением элементов в конец следует рассматривать возможность построить список в обратном порядке путем добавления элементов спереди. Затем, когда это будет сделано, нужно вызвать метод реверсирования `reverse`, чтобы получить элементы в требуемом порядке.

Другой вариант, который позволяет избежать реверсирования, — использовать объект `ListBuffer`. Это содержащийся в пакете `scala.collection.mutable` изменяемый объект, который может помочь более эффективно строить списки, когда нужно добавлять элементы в их конец. Объект обеспечивает постоянное время выполнения операций добавления элементов как в конец, так и в начало списка. В конец списка элемент добавляется с помощью оператора `+=`[1], а в начало — с помощью оператора `+=:`. Когда построение будет завершено, можно получить список типа `List`, вызвав в отношении `ListBuffer` метод `toList`. Соответствующий пример выглядит так:

```
scala> import scala.collection.mutable.ListBuffer
import scala.collection.mutable.ListBuffer

scala> val buf = new ListBuffer[Int]
buf: scala.collection.mutable.ListBuffer[Int] = ListBuffer()

scala> buf += 1
res4: buf.type = ListBuffer(1)

scala> buf += 2
res5: buf.type = ListBuffer(1, 2)

scala> buf
res6: scala.collection.mutable.ListBuffer[Int] =
    ListBuffer(1, 2)

scala> 3 +=: buf
res7: buf.type = ListBuffer(3, 1, 2)

scala> buf.toList
res8: List[Int] = List(3, 1, 2)
```

Еще один повод использовать `ListBuffer` вместо `List` — возможность предотвратить потенциальное переполнение стека. Если можно создать список в нужном

[1] Операторы += и +=: — псевдонимы для добавления в конец и в начало соответственно.

порядке, добавив элементы в его начало, но рекурсивный алгоритм, который потребуется, не является алгоритмом с хвостовой рекурсией, то вместо этого можно задействовать выражение `for` или цикл `while` и `ListBuffer`. Такой способ применения `ListBuffer` будет показан в разделе 22.2.

Буферы массивов

Объект `ArrayBuffer` похож на массив, за исключением того, что в дополнение ко всему здесь предоставляет возможность добавлять и удалять элементы в начало и в конец последовательности. Доступны все те же операции, что и в классе `Array`, хотя выполняются они несколько медленнее, поскольку в реализации есть уровень-оболочка. На новые операции добавления и удаления затрачивается в среднем одно и то же время, но иногда требуется время, пропорциональное размеру, из-за реализации, требующей выделить новый массив для хранения содержимого буфера.

Чтобы воспользоваться `ArrayBuffer`, нужно сначала импортировать его из пакета изменяемых коллекций:

```
scala> import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable.ArrayBuffer
```

При создании `ArrayBuffer` нужно указать параметр типа, а длину указывать не обязательно. По мере надобности `ArrayBuffer` автоматически установит выделяемое пространство памяти:

```
scala> val buf = new ArrayBuffer[Int]()
buf: scala.collection.mutable.ArrayBuffer[Int] =
ArrayBuffer()
```

Добавить элемент в `ArrayBuffer` можно с помощью метода `+=`:

```
scala> buf += 12
res9: buf.type = ArrayBuffer(12)

scala> buf += 15
res10: buf.type = ArrayBuffer(12, 15)

scala> buf
res11: scala.collection.mutable.ArrayBuffer[Int] =
    ArrayBuffer(12, 15)
```

Доступны все обычные методы работы с массивами. Например, можно запросить у `ArrayBuffer` его длину или извлечь элемент по его индексу:

```
scala> buf.length
res12: Int = 2

scala> buf(0)
res13: Int = 12
```

Строки (реализуемые через StringOps)

Еще одной последовательностью, заслуживающей упоминания, является `StringOps`. В ней реализованы многие методы работы с последовательностями. Поскольку в `Predef` есть неявное преобразование из `String` в `StringOps`, то с любой строкой можно работать как с последовательностью. Вот пример:

```
scala> def hasUpperCase(s: String) = s.exists(_.isUpper)
hasUpperCase: (s: String)Boolean

scala> hasUpperCase("Robert Frost")
res14: Boolean = true

scala> hasUpperCase("e e cummings")
res15: Boolean = false
```

В этом примере метод `exists` вызывается в отношении строки, которая в теле метода `hasUpperCase` называется `s`. В самом классе `String` не объявлено никакого метода по имени `exists`, поэтому компилятор Scala выполнит неявное преобразование `s` в `StringOps`, где такой метод есть. Метод `exists` считает строку последовательностью символов и вернет значение `true`, если какой-либо из них относится к верхнему регистру[1].

17.2. Множества и отображения

В предыдущих главах, начиная с шага 10 в главе 3, уже были показаны основы множеств и отображений. Прочитав этот раздел, вы получите более глубокое представление о способах их использования и увидите несколько дополнительных примеров.

Ранее мы уже говорили, что библиотека коллекций Scala предлагает как изменяемые, так и неизменяемые версии множеств и отображений. Иерархия множеств показана на рис. 3.2, а иерархия отображений — на рис. 3.3 (см. выше). Из этих схем следует, что простые имена `Set` и `Map` используются тремя трейтами и все они находятся в разных пакетах.

По умолчанию, когда в коде используется `Set` или `Map`, вы получаете неизменяемый объект. Если нужен изменяемый вариант, то следует применить явно указанное импортирование. К неизменяемым вариантам Scala предоставляет самый простой доступ — в качестве небольшого поощрения за то, что предпочтение отдано им, а не их изменяемым аналогам. Доступ предоставляется через объект `Predef`, неявно импортируемый в каждый файл исходного кода на языке Scala. Соответствующие определения показаны в листинге 17.1.

[1] Похожий пример представлен в главе 1.

Листинг 17.1. Исходные определения отображений map и множеств в set в Predef

```
object Predef {
  type Map[A, +B] = collection.immutable.Map[A, B]
  type Set[A] = collection.immutable.Set[A]
  val Map = collection.immutable.Map
  val Set = collection.immutable.Set
  // ...
}
```

Имена `Set` и `Map` в качестве псевдонимов для более длинных полных имен трейтов неизменяемых множеств и отображений в `Predef` определяются с помощью ключевого слова `type`[1]. Чтобы ссылаться на объекты-одиночки для неизменяемых `Set` и `Map`, выполняется инициализация `val`-переменных с именами `Set` и `Map`. Следовательно, `Map` является тем же, что и объект `Predef.Map`, который определен быть тем же самым, что и `scala.collection.immutable.Map`. Это справедливо как для типа `Map`, так и для объекта `Map`.

Если нужно воспользоваться как изменяемыми, так и неизменяемыми множествами или отображениями, то одним из подходов является импортирование имен пакетов, в которых содержатся изменяемые варианты:

```
scala> import scala.collection.mutable
import scala.collection.mutable
```

Можно продолжать ссылаться на неизменяемое множество, как и прежде `Set`, но теперь можно будет сослаться и на изменяемое множество, указав `mutable.Set`. Вот как выглядит соответствующий пример:

```
scala> val mutaSet = mutable.Set(1, 2, 3)
mutaSet: scala.collection.mutable.Set[Int] = Set(1, 2, 3)
```

Использование множеств

Ключевой характеристикой множества является то, что оно гарантирует наличие каждого объекта не более чем в одном экземпляре, согласно определению оператора `==`. В качестве примера воспользуемся множеством, чтобы вычислить количество уникальных слов в строке.

Если указать в качестве разделителей слов пробелы и знаки пунктуации, то метод `split` класса `String` может разбить строку на слова. Для этого вполне достаточно применить регулярное выражение `[ !,.]+`: оно показывает, что строка должна быть разбита во всех местах, где есть один или несколько пробелов и/или знак пунктуации.

```
scala> val text = "See Spot run. Run, Spot. Run!"
text: String = See Spot run. Run, Spot. Run!
```

[1] Более подробно ключевое слово type мы рассмотрим в разделе 20.6.

```
scala> val wordsArray = text.split("[ !,.]+")
wordsArray: Array[String]
  = Array(See, Spot, run, Run, Spot, Run)
```

Чтобы посчитать уникальные слова, их можно преобразовать, приведя их символы к единому регистру, а затем добавить во множество. Поскольку множества исключают дубликаты, то каждое уникальное слово будет появляться во множестве только раз.

Сначала можно создать пустое множество, используя метод `empty`, предоставляемый объектом-компаньоном `Set`:

```
scala> val words = mutable.Set.empty[String]
words: scala.collection.mutable.Set[String] = Set()
```

Далее, просто перебирая слова с помощью выражения `for`, можно преобразовать каждое слово, приведя его символы к нижнему регистру, а затем добавить его в изменяемое множество, воспользовавшись оператором `+=`:

```
scala> for (word <- wordsArray)
         words += word.toLowerCase

scala> words
res17: scala.collection.mutable.Set[String] = Set(see, run, spot)
```

Таким образом, в тексте содержатся три уникальных слова: `spot`, `run` и `see`. Наиболее часто используемые методы, применяемые равно к изменяемым и неизменяемым множествам, показаны в табл. 17.1.

Таблица 17.1. Наиболее распространенные операторы для работы с множествами

Что используется	Что этот метод делает
val nums = Set(1, 2, 3)	Создает неизменяемое множество (nums.toString возвращает Set(1, 2, 3))
nums + 5	Добавляет элемент в неизменяемое множество (возвращает Set(1, 2, 3, 5))
nums – 3	Удаляет элемент из неизменяемого множества (возвращает Set(1, 2))
nums ++ List(5, 6)	Добавляет несколько элементов (возвращает Set(1, 2, 3, 5, 6))
nums –– List(1, 2)	Удаляет несколько элементов из неизменяемого множества (возвращает Set(3))
nums & Set(1, 3, 5, 7)	Выполняет пересечение двух множеств (возвращает Set(1, 3))
nums.size	Возвращает размер множества (возвращает 3)
nums.contains(3)	Проверка включения (возвращает true)
import scala.collection.mutable	Упрощает доступ к изменяемым коллекциям

Продолжение

Таблица 17.1 *(продолжение)*

Что используется	Что этот метод делает
val words = mutable.Set. empty[String]	Создает пустое, изменяемое множество (words.toString возвращает Set())
words += "the"	Добавляет элемент (words.toString возвращает Set(the))
words –= "the"	Удаляет элемент, если он существует (words.toString возвращает Set())
words ++= List("do", "re", "mi")	Добавляет несколько элементов (words.toString возвращает Set(do, re, mi))
words ––= List("do", "re")	Удаляет несколько элементов (words.toString возвращает Set(mi))
words.clear	Удаляет все элементы (words.toString возвращает Set())

Применение отображений

Отображения позволяют связать значение с каждым элементом множества. Отображение и массив используются похожим образом, за исключением того, что вместо индексирования с помощью целых чисел, начинающихся с нуля, можно применить ключи любого вида. Если импортировать пакет с именем `mutable`, то можно создать пустое изменяемое отображение:

```
scala> val map = mutable.Map.empty[String, Int]
map: scala.collection.mutable.Map[String,Int] = Map()
```

Учтите, что при создании отображения следует указать два типа. Первый тип предназначен для *ключей* отображения, а второй — для их *значений*. В данном случае ключами являются строки, а значениями — целые числа. Задание записей в отображении похоже на задание записей в массиве:

```
scala> map("hello") = 1

scala> map("there") = 2

scala> map
res20: scala.collection.mutable.Map[String,Int] =
    Map(hello -> 1, there -> 2)
```

По аналогии с этим чтение отображения похоже на чтение массива:

```
scala> map("hello")
res21: Int = 1
```

Чтобы связать все воедино, рассмотрим метод, подсчитывающий количество появлений каждого из слов в строке:

```
scala> def countWords(text: String) = {
         val counts = mutable.Map.empty[String, Int]
```

```
        for (rawWord <- text.split("[ ,!.]+")) {
          val word = rawWord.toLowerCase
          val oldCount =
            if (counts.contains(word)) counts(word)
            else 0
          counts += (word -> (oldCount + 1))
        }
        counts
      }
countWords: (text:
String)scala.collection.mutable.Map[String,Int]

scala> countWords("See Spot run! Run, Spot. Run!")
res22: scala.collection.mutable.Map[String,Int] =
    Map(spot -> 2, see -> 1, run -> 3)
```

Этот код работает благодаря тому, что используется изменяемое отображение по имени `counts` и отображается каждое слово на количество его появлений в тексте. Для каждого слова в тексте выполняется поиск предыдущего количества появлений слова и его увеличение на единицу, а затем в `counts` сохраняется новое значение количества. Обратите внимание: проверка того, встречалось ли это слово раньше, выполняется с помощью метода `contains`. Если `counts.contains(word)` не возвращает `true`, значит, слово еще не встречалось и за количество принимается нуль.

Многие из наиболее часто используемых методов работы как с изменяемыми, так и с неизменяемыми отображениями показаны в табл. 17.2.

Таблица 17.2. Наиболее часто используемые операции для работы с отображениями

Что используется	Что этот метод делает
val nums = Map("i" –> 1, "ii" –> 2)	Создает неизменяемое отображение (nums.toString возвращает Map(i –> 1, ii –> 2))
nums + ("vi" –> 6)	Добавляет запись в неизменяемое отображение (возвращает Map(i –> 1, ii –> 2, vi –> 6))
nums – "ii"	Удаляет запись из неизменяемого отображения (возвращает Map(i –> 1))
nums ++ List("iii" –> 3, "v" –> 5)	Добавляет несколько записей (возвращает Map(i –> 1, ii –> 2, iii –> 3, v –> 5))
nums –– List("i", "ii")	Удаляет несколько записей из неизменяемого отображения (возвращает Map())
nums.size	Возвращает размер отображения (возвращает 2)
nums.contains("ii")	Проверяет на включение (возвращает true)
nums("ii")	Извлекает значение по указанному ключу (возвращает 2)

Продолжение

Таблица 17.2 *(продолжение)*

Что используется	Что этот метод делает
nums.keys	Возвращает ключи (возвращает результат итерации, выполненной над строками "i" и "ii")
nums.keySet	Возвращает ключи в виде множества (возвращает Set(i, ii))
nums.values	Возвращает значения (возвращает Iterable над целыми числами 1 и 2)
nums.isEmpty	Показывает, является ли отображение пустым (возвращает false)
import scala.collection.mutable	Упрощает доступ к изменяемым коллекциям
val words = mutable.Map.empty[String, Int]	Создает пустое изменяемое отображение
words += ("one" –> 1)	Добавляет запись в отображение из ключа "one" и значения 1 (words.toString возвращает Map(one –> 1))
words –= "one"	Удаляет запись из отображения, если она существует (words.toString возвращает Map())
words ++= List("one" –> 1, "two" –> 2, "three" –> 3)	Добавляет записи в изменяемое отображение (words.toString возвращает Map(one –> 1, two –> 2, three –> 3))
words ––= List("one", "two")	Удаляет несколько объектов (words.toString возвращает Map(three –> 3))

Множества и отображения, используемые по умолчанию

Для большинства случаев реализаций изменяемых и неизменяемых множеств и отображений, предоставляемых `Set()`, `scala.collection.mutable.Map()` и тому подобными фабриками, наверное, вполне достаточно. Реализации, предоставляемые этими фабриками, используют алгоритм ускоренного поиска, в котором обычно задействуется хеш-таблица, поэтому они могут быстро обнаружить наличие или отсутствие объекта в коллекции.

Так, фабричный метод `scala.collection.mutable.Set()` возвращает `scala.collection.mutable.HashSet`, внутри которого используется хеш-таблица. Аналогично этому фабричный метод `scala.collection.mutable.Map()` возвращает `scala.collection.mutable.HashMap`.

История с неизменяемыми множествами и отображениями несколько сложнее. Как показано в табл. 17.3, класс, возвращаемый фабричным методом `scala.collection.immutable.Set()`, зависит, к примеру, от того, сколько элементов ему было передано. В целях достижения максимальной производительности для множеств, состоящих не более чем из пяти элементов, применяется специальный класс. Но при запросе множества из пяти и более элементов фабричный метод вернет реализацию, использующую хеш.

Таблица 17.3. Реализации используемых по умолчанию неизменяемых множеств

Количество элементов	Реализация
0	scala.collection.immutable.EmptySet
1	scala.collection.immutable.Set1
2	scala.collection.immutable.Set2
3	scala.collection.immutable.Set3
4	scala.collection.immutable.Set4
5 или более	scala.collection.immutable.HashSet

По аналогии с этим, как следует из данной таблицы, в результате выполнения фабричного метода `scala.collection.immutable.Map()` будет возвращен нужный класс в зависимости от того, сколько пар «ключ — значение» ему передано. Как и в случае с множествами, для того чтобы неизменяемые отображения с количеством элементов меньше пяти достигли максимальной производительности для отображения каждого конкретного размера, используется специальный класс. Но если отображение содержит пять и более пар «ключ — значение», то используется неизменяемый класс `HashMap`.

В целях обеспечения максимальной производительности используемые по умолчанию реализации неизменяемых классов, показанные в табл. 17.3 и 17.4, работают совместно. Например, если добавляется элемент к `EmptySet`, то возвращается `Set1`. Если добавляется элемент к этому `Set1`, то возвращается `Set2`. Если затем удалить элемент из `Set2`, то будет опять получен `Set1`.

Таблица 17.4. Реализации используемых по умолчанию неизменяемых отображений

Количество элементов	Реализация
0	scala.collection.immutable.EmptyMap
1	scala.collection.immutable.Map1
2	scala.collection.immutable.Map2
3	scala.collection.immutable.Map3
4	scala.collection.immutable.Map4
5 или более	scala.collection.immutable.HashMap

Отсортированные множества и отображения

Иногда может понадобиться множество или отображение, итератор которого возвращает элементы в определенном порядке. Для этого в библиотеке коллекций Scala имеются трейты `SortedSet` и `SortedMap`. Они реализованы с помощью классов `TreeSet` и `TreeMap`, которые в целях хранения элементов в определенном порядке применяют красно-черное дерево (в случае `TreeSet`) или ключи (в случае

с TreeMap). Порядок определяется трейтом Ordered, неявный экземпляр которого должен быть определен для типа элементов н множества или типа ключей отображения. Эти классы поставляются в изменяемых и неизменяемых вариантах. Рассмотрим ряд примеров использования TreeSet:

```
scala> import scala.collection.immutable.TreeSet
import scala.collection.immutable.TreeSet

scala> val ts = TreeSet(9, 3, 1, 8, 0, 2, 7, 4, 6, 5)
ts: scala.collection.immutable.TreeSet[Int] =
    TreeSet(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> val cs = TreeSet('f', 'u', 'n')
cs: scala.collection.immutable.TreeSet[Char] =
    TreeSet(f, n, u)
```

А это ряд примеров использования TreeMap:

```
scala> import scala.collection.immutable.TreeMap
import scala.collection.immutable.TreeMap

scala> var tm = TreeMap(3 -> 'x', 1 -> 'x', 4 -> 'x')
tm: scala.collection.immutable.TreeMap[Int,Char] =
    Map(1 -> x, 3 -> x, 4 -> x)

scala> tm += (2 -> 'x')

scala> tm
res30: scala.collection.immutable.TreeMap[Int,Char] =
    Map(1 -> x, 2 -> x, 3 -> x, 4 -> x)
```

17.3. Выбор между изменяемыми и неизменяемыми коллекциями

При решении одних задач лучше работают изменяемые коллекции, а при решении других — неизменяемые. В случае сомнений лучше начать с неизменяемой коллекции, а позже при необходимости перейти к изменяемой, поскольку разобраться в работе неизменяемых коллекций гораздо проще.

Иногда может оказаться полезно двигаться в обратном направлении. Если код, использующий изменяемые коллекции, становится сложным и в нем трудно разобраться, то следует подумать, стоит ли заменить некоторые коллекции их неизменяемыми альтернативами. В частности, если вас волнует вопрос создания копий изменяемых коллекций только в нужных местах или вы слишком много думаете над тем, кто владеет изменяемой коллекцией или что именно она содержит, то имеет смысл заменить некоторые коллекции на их неизменяемые аналоги.

Помимо того что в неизменяемых коллекциях потенциально легче разобраться, они, как правило, могут храниться более компактно, чем изменяемые, если коли-

чество хранящихся в них элементов невелико. Например, пустое изменяемое отображение в его представлении по умолчанию в виде `HashMap` занимает примерно 80 байт, и около 16 дополнительных байт требуется для добавления к нему каждой записи. А пустое неизменяемое отображение `Map` — это один объект, который совместно используется всеми ссылками, и потому ссылаться на него можно, по сути, одним полем указателя.

Более того, в настоящее время библиотека коллекций Scala хранит до четырех записей неизменяемых отображений и множеств в одном объекте, который в зависимости от количества хранящихся в коллекции записей обычно занимает от 16 до 40 байт[1]. Следовательно, для небольших множеств и отображений неизменяемые версии занимают намного меньше места, чем изменяемые. С учетом того, что многие коллекции весьма невелики, переход на их неизменяемый вариант может существенно сэкономить пространство памяти и дать преимущества в производительности.

Чтобы облегчить переход с неизменяемых на изменяемые коллекции и наоборот, Scala предоставляет немного «синтаксического сахара». Неизменяемые множества и отображения не поддерживают настоящий метод `+=`, однако в Scala дается полезная альтернативная интерпретация `+=`. Когда используется запись `a += b` и `a` не поддерживает метод по имени `+=`, Scala пытается интерпретировать эту запись как `a = a + b`.

Например, неизменяемые множества не поддерживают оператор `+=`:

```
scala> val people = Set("Nancy", "Jane")
people: scala.collection.immutable.Set[String] =
    Set(Nancy, Jane)

scala> people += "Bob"
<console>:14: error: value += is not a member of
scala.collection.immutable.Set[String]
              people += "Bob"
                     ^
```

Но если объявить `people` в качестве `var`-, а не `val`-переменной, то коллекцию можно обновить с помощью операции `+=` даже притом, что она неизменяемая. Сначала создается новая коллекция, а затем переменной `people` присваивается новое значение для ссылки на новую коллекцию:

```
scala> var people = Set("Nancy", "Jane")
people: scala.collection.immutable.Set[String] =
    Set(Nancy, Jane)

scala> people += "Bob"

scala> people
res34: scala.collection.immutable.Set[String] =
    Set(Nancy, Jane, Bob)
```

[1] Под одним объектом, как следует из табл. 17.3 и 17.4, понимается экземпляр одного из классов: от Set1 до Set4 или от Map1 до Map4.

После этой серии инструкций переменная `people` ссылается на новое неизменяемое множество, содержащее добавленную строку `"Bob"`. Та же идея применима не только к методу `+=`, но и к любому другому методу, заканчивающемуся знаком `=`. Вот как тот же самый синтаксис используется с оператором `-=`, который удаляет элемент из множества, и с оператором `++=`, добавляющим в множество коллекцию элементов:

```
scala> people -= "Jane"

scala> people ++= List("Tom", "Harry")

scala> people
res37: scala.collection.immutable.Set[String] =
    Set(Nancy, Bob, Tom, Harry)
```

Чтобы понять, насколько это полезно, рассмотрим еще раз пример отображения `Map` из раздела 1.1:

```
var capital = Map("US" -> "Washington", "France" -> "Paris")
capital += ("Japan" -> "Tokyo")
println(capital("France"))
```

В этом коде используются неизменяемые коллекции. Если захочется попробовать задействовать вместо них изменяемые коллекции, то нужно будет всего лишь импортировать изменяемую версию `Map`, переопределив таким образом выполненный по умолчанию импорт неизменяемой версии `Map`:

```
import scala.collection.mutable.Map    // Единственное требуемое изменение!
var capital = Map("US" -> "Washington", "France" -> "Paris")
capital += ("Japan" -> "Tokyo")
println(capital("France"))
```

Так легко преобразовать удастся не все примеры, но особая трактовка методов, заканчивающихся знаком равенства, зачастую сокращает объем кода, который нуждается в изменениях.

Кстати, эта трактовка синтаксиса работает не только с коллекциями, но и с любыми разновидностями значений. Например, здесь она была использована в отношении чисел с плавающей точкой:

```
scala> var roughlyPi = 3.0
roughlyPi: Double = 3.0

scala> roughlyPi += 0.1

scala> roughlyPi += 0.04

scala> roughlyPi
res40: Double = 3.14
```

Эффект от такого расширяющего преобразования похож на эффект, получаемый от операторов присваивания, использующихся в Java (+=, -=, *= и т. п.), однако носит более общий характер, поскольку преобразован может быть каждый оператор, заканчивающийся на =.

17.4. Инициализация коллекций

Как уже было показано, наиболее широко востребованный способ создания и инициализации коллекции — передача исходных элементов фабричному методу, определенному в объекте-компаньоне класса выбранной вами коллекции. Элементы просто помещаются в круглые скобки после имени объекта-компаньона, и компилятор Scala преобразует это в вызов метода `apply` в отношении объекта-компаньона:

```
scala> List(1, 2, 3)
res41: List[Int] = List(1, 2, 3)

scala> Set('a', 'b', 'c')
res42: scala.collection.immutable.Set[Char] = Set(a, b, c)

scala> import scala.collection.mutable
import scala.collection.mutable

scala> mutable.Map("hi" -> 2, "there" -> 5)
res43: scala.collection.mutable.Map[String,Int] =
    Map(hi -> 2, there -> 5)

scala> Array(1.0, 2.0, 3.0)
res44: Array[Double] = Array(1.0, 2.0, 3.0)
```

Чаще всего можно позволить компилятору Scala вывести тип элемента коллекции из элементов, переданных ее фабричному методу. Но иногда может понадобиться создать коллекцию, указав притом тип, отличающийся от того, который выберет компилятор. Это особенно касается изменяемых коллекций. Рассмотрим пример:

```
scala> import scala.collection.mutable
import scala.collection.mutable

scala> val stuff = mutable.Set(42)
stuff: scala.collection.mutable.Set[Int] = Set(42)

scala> stuff += "abracadabra"
<console>:16: error: type mismatch;
 found   : String("abracadabra")
 required: Int
              stuff += "abracadabra"
                       ^
```

Проблема здесь заключается в том, что переменной `stuff` был задан тип элемента `Int`. Если вы хотите, чтобы типом элемента был `Any`, то это нужно указать явно, поместив тип элемента в квадратные скобки:

```
scala> val stuff = mutable.Set[Any](42)
stuff: scala.collection.mutable.Set[Any] = Set(42)
```

Еще одна особая ситуация возникает при желании инициализировать коллекцию с помощью другой коллекции. Допустим, есть список, но нужно получить коллекцию `TreeSet`, содержащую элементы, которые находятся в нем. Список выглядит так:

```
scala> val colors = List("blue", "yellow", "red", "green")
colors: List[String] = List(blue, yellow, red, green)
```

Передать список названий цветов фабричному методу для `TreeSet` невозможно:

```
scala> import scala.collection.immutable.TreeSet
import scala.collection.immutable.TreeSet

scala> val treeSet = TreeSet(colors)
<console>:16: error: No implicit Ordering defined for
List[String].
       val treeSet = TreeSet(colors)
                             ^
```

Вместо этого вам нужно будет преобразовать список в `TreeSet` с помощью метода `to`:

```
scala> val treeSet = colors to TreeSet
treeSet: scala.collection.immutable.TreeSet[String] =
    TreeSet(blue, green, red, yellow)
```

Метод `to` принимает в качестве параметра объект-компаньон коллекции. С его помощью вы можете преобразовать любую коллекцию в другую.

Преобразование в массив или список

Помимо универсального метода `to` для преобразования коллекции в другую произвольную коллекцию, вы также можете использовать более конкретные методы для преобразования в наиболее распространенные типы коллекций Scala. Как было показано ранее, чтобы инициализировать новый список с помощью другой коллекции, следует просто вызвать в отношении этой коллекции метод `toList`:

```
scala> treeSet.toList
res50: List[String] = List(blue, green, red, yellow)
```

Или же, если нужен массив, вызвать метод `toArray`:

```
scala> treeSet.toArray
res51: Array[String] = Array(blue, green, red, yellow)
```

Обратите внимание: несмотря на неотсортированность исходного списка `colors`, элементы в списке, создаваемом вызовом `toList` в отношении `TreeSet`, стоят в алфавитном порядке. Когда в отношении коллекции вызывается `toList` или `toArray`, порядок следования элементов в списке, получающемся в результате, будет таким же, как и порядок следования элементов, создаваемый итератором на этой коллекции. Поскольку итератор, принадлежащий типу `TreeSet[String]`, будет выдавать строки в алфавитном порядке, то они в том же порядке появятся и в списке, который создается в результате вызова `toList` в отношении объекта `TreeSet`.

Разница между `xs to List` и `xs.toList` в том, что реализация `toList` может быть переопределена конкретным типом коллекции `xs`. Это делает преобразование ее элементов в список более эффективным по сравнению с реализацией по умолчанию, копирующей все элементы коллекции. Например, коллекция `ListBuffer` переопределяет метод `toList` с помощью реализации, которая имеет постоянные время выполнения и объем памяти.

Но следует иметь в виду, что преобразование в списки или массивы, как правило, требует копирования всех элементов коллекции и потому для больших коллекций может выполняться довольно медленно. Но иногда это все же приходится делать из-за уже существующего API. Кроме того, многие коллекции содержат всего несколько элементов, а при этом потери в скорости незначительны.

Преобразования между изменяемыми и неизменяемыми множествами и отображениями

Иногда возникает еще одна ситуация, требующая преобразования изменяемого множества либо отображения в неизменяемый аналог или *наоборот*. Выполнить эти задачи следует с помощью метода, показанного чуть ранее. Преобразование неизменяемого множества `TreeSet` из предыдущего примера в изменяемый и обратно в неизменяемый выполняется так:

```
scala> import scala.collection.mutable
import scala.collection.mutable

scala> treeSet
res52: scala.collection.immutable.TreeSet[String] =
    TreeSet(blue, green, red, yellow)

scala> val mutaSet = treeSet to mutable.Set
mutaSet: scala.collection.mutable.Set[String] =
    Set(red, blue, green, yellow)

scala> val immutaSet = mutaSet to Set
immutaSet: scala.collection.immutable.Set[String] =
    Set(red, blue, green, yellow)
```

Ту же технику можно применить для преобразований между изменяемыми и неизменяемыми отображениями:

```
scala> val muta = mutable.Map("i" -> 1, "ii" -> 2)
muta: scala.collection.mutable.Map[String,Int] =
    Map(ii -> 2,i -> 1)

scala> val immu = muta to Map
immu: scala.collection.immutable.Map[String,Int] =
    Map(ii -> 2, i -> 1)
```

17.5. Кортежи

Согласно описанию, которое дано в шаге 9 в главе 3, кортеж объединяет фиксированное количество элементов, позволяя выполнять их передачу в виде единого целого. В отличие от массива или списка, кортеж может содержать объекты различных типов. Вот как, к примеру, выглядит кортеж, содержащий целое число, строку и консоль:

```
(1, "hello", Console)
```

Кортежи избавляют вас от скуки, возникающей при определении упрощенных, насыщенных данными классов. Даже притом, что определить класс не составляет особого труда, все же требуется приложить определенное количество порой напрасных усилий. Кортежи избавляют вас от необходимости выбирать имя класса, область видимости, в которой определяется класс, и имена для членов класса. Если класс просто хранит целое число и строку, то добавление класса по имени `AnIntegerAndAString` особой ясности не внесет.

Поскольку кортежи могут сочетать объекты различных типов, они не являются наследниками класса `Iterable`. Если потребуется сгруппировать ровно одно целое число и ровно одну строку, то понадобится кортеж, а не `List` или `Array`.

Довольно часто кортежи применяются для возвращения из метода нескольких значений. Рассмотрим, к примеру, метод, который выполняет поиск самого длинного слова в коллекции и возвращает наряду с ним его индекс:

```
def longestWord(words: Array[String]): (String, Int) = {
  var word = words(0)
  var idx = 0
  for (i <- 1 until words.length)
    if (words(i).length > word.length) {
      word = words(i)
      idx = i
    }
  (word, idx)
}
```

А вот пример использования этого метода:

```
scala> val longest =
         longestWord("The quick brown fox".split(" "))
longest: (String, Int) = (quick,1)
```

Функция `longestWord` выполняет здесь два вычисления, получая при этом слово `word`, являющееся в массиве самым длинным, и его индекс `idx`. Во избежание усложнений в функции предполагается, что список имеет хотя бы одно слово, и она отдает предпочтение тому из одинаковых по длине слов, которое стоит в списке первым. Как только функция выберет, какое слово и какой индекс возвращать, она возвращает их вместе, используя синтаксис кортежа `(word, idx)`.

Доступ к элементам кортежа можно получить с помощью метода `_1`, чтобы получить первый элемент, метода `_2`, чтобы получить второй, и т. д.:

```
scala> longest._1
res53: String = quick

scala> longest._2
res54: Int = 1
```

Кроме того, значение каждого элемента кортежа можно присвоить собственной переменной[1]:

```
scala> val (word, idx) = longest
word: String = quick
idx: Int = 1

scala> word
res55: String = quick
```

Кстати, если не поставить круглые скобки, то будет получен совершенно иной результат:

```
scala> val word, idx = longest
word: (String, Int) = (quick,1)
idx: (String, Int) = (quick,1)
```

Представленный синтаксис дает *множественное определение* одного и того же выражения. Каждая переменная инициализируется собственным вычислением выражения в правой части. В данном случае неважно, что это выражение вычисляется в кортеж. Обе переменные инициализируются всем кортежем целиком. Ряд примеров, в которых удобно применять множественные определения, можно увидеть в главе 18.

Следует заметить, что кортежи довольно просты в использовании. Они очень хорошо подходят для объединения данных, не имеющих никакого другого смысла,

---

[1] Этот синтаксис является, по сути, особым случаем сопоставления с образцом, подробно рассмотренным в разделе 15.7.

кроме «А и Б». Но когда объединение имеет какое-либо значение или нужно добавить к объединению некие методы, то лучше пойти дальше и создать класс. Например, не стоит использовать кортеж из трех значений, чтобы объединить месяц, день и год, — нужно создать класс `Date`. Так вы явно обозначите свои намерения, благодаря чему код станет более понятным для читателей, и это позволит компилятору и средствам самого языка помочь вам отловить возможные ошибки.

Резюме

В данной главе мы дали обзор библиотеки коллекций Scala и рассмотрели наиболее важные ее классы и трейты. Опираясь на эти знания, вы сможете эффективно работать с коллекциями Scala и будете знать, что именно нужно искать в Scaladoc, когда возникнет необходимость в дополнительных сведениях. Более подробную информацию о коллекциях Scala можно найти в главах 24 и 25. А в следующей главе мы переключим внимание с библиотеки Scala на сам язык, и нам предстоит рассмотреть имеющуюся в Scala поддержку изменяемых объектов.

18 Изменяемые объекты

В предыдущих главах в центре внимания были функциональные (неизменяемые) объекты. Дело в том, что идея использования объектов без какого-либо изменяемого состояния заслуживала более пристального рассмотрения. Но в Scala также вполне возможно определять объекты с изменяемым состоянием. Подобные изменяемые объекты зачастую появляются естественным образом, когда нужно смоделировать объекты из реального мира, которые со временем подвергаются изменениям.

В этой главе мы раскроем суть изменяемых объектов и рассмотрим синтаксические средства для их выражения, предлагаемые Scala. Кроме того, рассмотрим большой пример моделирования дискретных событий, в котором используются изменяемые объекты, а также описан внутренний предметно-ориентированный язык (domain-specific language, DSL), предназначенный для определения моделируемых цифровых электронных схем.

18.1. Что делает объект изменяемым

Принципиальную разницу между чисто функциональным и изменяемым объектами можно проследить, даже не изучая реализацию объектов. При вызове метода или получении значения поля по указателю в отношении функционального объекта вы всегда будете получать один и тот же результат.

Например, если есть следующий список символов:

```
val cs = List('a', 'b', 'c')
```

то применение `cs.head` всегда будет возвращать `'a'`. То же самое произойдет, даже если между местом определения `cs` и местом, где будет применено обращение `cs.head`, над списком `cs` будет проделано произвольное количество других операций.

Что же касается изменяемого объекта, то результат вызова метода или обращения к полю может зависеть от того, какие операции были ранее выполнены в отношении объекта. Хороший пример изменяемого объекта — банковский счет. Его упрощенная реализация показана в листинге 18.1.

Листинг 18.1. Изменяемый класс банковского счета

```
class BankAccount {

  private var bal: Int = 0

  def balance: Int = bal

  def deposit(amount: Int) = {
    require(amount > 0)
    bal += amount
  }

  def withdraw(amount: Int): Boolean =
    if (amount > bal) false
    else {
      bal -= amount
      true
    }
}
```

В классе `BankAccount` определяются приватная переменная `bal` и три публичных метода: `balance` возвращает текущий баланс, `deposit` добавляет к `bal` заданную сумму, `withdraw` предпринимает попытку вывести из `bal` заданную сумму, гарантируя при этом, что баланс не станет отрицательным. Возвращаемое `withdraw` значение, имеющее тип `Boolean`, показывает, были ли запрошенные средства успешно выведены.

Даже если ничего не знать о внутренней работе класса `BankAccount`, все же можно сказать, что экземпляры `BankAccounts` являются изменяемыми объектами:

```
scala> val account = new BankAccount
account: BankAccount = BankAccount@d504137

scala> account deposit 100

scala> account withdraw 80
res1: Boolean = true

scala> account withdraw 80
res2: Boolean = false
```

Обратите внимание: в двух последних операциях вывода средств в ходе работы с программой были возвращены разные результаты. По выполнении первой операции было возвращено значение `true`, поскольку на банковском счету содержался достаточный объем, позволяющий вывести средства. Вторая операция вывода средств была такой же, как и первая, однако по ее выполнении было возвращено значение `false`, поскольку баланс счета уменьшился настолько, что уже не мог покрыть запрошенные средства. Исходя из этого, мы понимаем, что банковским счетам присуще изменяемое состояние, так как в результате одной и той же операции в разное время получаются разные результаты.

Можно подумать, будто изменяемость BankAccount априори не вызывает сомнений, поскольку в нем содержится определение var-переменной. Изменяемость и var-переменные обычно идут рука об руку, но ситуация не всегда бывает столь очевидной. Например, класс может быть изменяемым и без определения или наследования каких-либо var-переменных, поскольку перенаправляет вызовы методов другим объектам, которые находятся в изменяемом состоянии. Может сложиться и обратная ситуация: класс содержит var-переменные и все же является чисто функциональным. Как образец можно привести класс, кэширующий результаты дорогой операции в поле в целях оптимизации. Чтобы подобрать пример, предположим наличие неоптимизированного класса Keyed с дорогой операцией computeKey:

```
class Keyed {
  def computeKey: Int = ... // Это займет некоторое время
  ...
}
```

При условии, что computeKey не читает и не записывает никаких var-переменных, эффективность Keyed можно увеличить, добавив кэш:

```
class MemoKeyed extends Keyed {
  private var keyCache: Option[Int] = None
  override def computeKey: Int = {
    if (!keyCache.isDefined) keyCache = Some(super.computeKey)
    keyCache.get
  }
}
```

Использование MemoKeyed вместо Keyed может ускорить работу: когда результат выполнения операции computeKey будет запрошен повторно, вместо еще одного запуска computeKey может быть возвращено значение, сохраненное в поле keyCache. Но за исключением такого ускорения поведение классов Keyed и MemoKeyed абсолютно одинаково. Следовательно, если Keyed является чисто функциональным классом, то таковым будет и класс MemoKeyed, даже притом, что содержит переназначаемую переменную.

18.2. Переназначаемые переменные и свойства

В отношении переназначаемой переменной допускается выполнение двух основных операции: получения ее значения или присваивания ей нового. В таких библиотеках, как JavaBeans, эти операции часто инкапсулированы в отдельные методы считывания (getter) и записи значения (setter), которые необходимо объявлять явно.

В Scala каждая var-переменная представляет собой неприватный член какого-либо объекта, в отношении которого в нем неявно определены методы геттер и сеттер. Но названия таких методов отличаются от предписанных соглашениями Java. Метод получения значения (геттер) var-переменной x называется просто x, а метод присваивания значения (сеттер) — x_=.

Например, появляясь в классе, определение var-переменной:

```
var hour = 12
```

вдобавок к переназначаемому полю создает геттер hour и сеттер hour_=. У поля всегда имеется пометка private[this], означающая, что значение для него может устанавливаться только из содержащего это поле объекта. В то же время геттер и сеттер обеспечивают исходной var-переменной некоторую видимость. Если var-переменная объявлена публичной (public), то таковыми же являются и геттер и сеттер. Если она является защищенной (protected), то и они тоже, и т. д.

Рассмотрим, к примеру, класс Time, показанный в листинге 18.2, в котором определены две публичные var-переменные с именами hour и minute.

Листинг 18.2. Класс с публичными var-переменными

```
class Time {
  var hour = 12
  var minute = 0
}
```

Эта реализация в точности соответствует определению класса, показанного в листинге 18.3. В данном определении имена локальных полей h и m были выбраны произвольно, чтобы не конфликтовали с уже используемыми именами.

Листинг 18.3. Как публичные var-переменные расширяются в геттер и сеттер

```
class Time {

  private[this] var h = 12
  private[this] var m = 0

  def hour: Int = h
  def hour_=(x: Int) = { h = x }

  def minute: Int = m
  def minute_=(x: Int) = { m = x }
}
```

Интересным аспектом такого расширения var-переменных в геттер и сеттер является то, что вместо определения var-переменной можно также выбрать вариант непосредственного определения этих методов доступа. Он позволяет как угодно интерпретировать операции доступа к переменной и присваивания ей значения. Например, вариант класса Time, показанный в листинге 18.4, содержит необходимые условия, благодаря которым перехватываются все присваивания недопустимых значений часам и минутам, хранящимся в переменных hour и minute.

Листинг 18.4. Непосредственное определение геттера и сеттера

```
class Time {

  private[this] var h = 12
  private[this] var m = 0
```

```
  def hour: Int = h
  def hour_= (x: Int) = {
    require(0 <= x && x < 24)
    h = x
  }

  def minute = m
  def minute_= (x: Int) = {
    require(0 <= x && x < 60)
    m = x
  }
}
```

В некоторых языках для этих похожих на переменные величин, не являющихся простыми переменными из-за того, что их геттер и сеттер могут быть переопределены, имеются специальные синтаксические конструкции. Например, в C# эту роль играют свойства. По сути, принятое в Scala соглашение о постоянной интерпретации переменной как имеющей пару геттер и сеттер предоставляет вам такие же возможности, что и свойства C#, но при этом не требует какого-то специального синтаксиса.

Свойства могут иметь множество назначений. В примере, показанном в листинге 18.4 (см. выше), методы присваивания значений навязывают соблюдение конкретных условий, защищая таким образом переменную от присваивания ей недопустимых значений. Кроме того, свойства позволяют регистрировать все обращения к переменной со стороны геттера и сеттера. Или же можно объединять переменные с событиями, например уведомляя с помощью методов-подписчиков о каждом изменении переменной (соответствующие примеры будут показаны в главе 35).

Вдобавок возможно, а иногда и полезно определять геттер и сеттер без связанных с ними полей. Например, в листинге 18.5 показан класс `Thermometer`, в котором инкапсулирована переменная `temperature`, позволяющая читать и обновлять ее значение. Температурные значения могут выражаться в градусах Цельсия или Фаренгейта. Этот класс позволяет получать и устанавливать значение температуры в любых единицах измерения.

Листинг 18.5. Определение геттера и сеттера без связанного с ними поля

```
class Thermometer {

  var celsius: Float = _

  def fahrenheit = celsius * 9 / 5 + 32
  def fahrenheit_= (f: Float) = {
    celsius = (f - 32) * 5 / 9
  }
  override def toString = s"${fahrenheit}F/${celsius}C"
}
```

В первой строке тела этого класса определяется `var`-переменная `celsius`, в которой будет храниться значение температуры в градусах Цельсия. Для переменной

`celsius` изначально устанавливается значение по умолчанию: в качестве инициализирующего значения для нее устанавливается знак _. Точнее, инициализатором поля `= _` данному полю присваивается нулевое значение. Суть нулевого значения зависит от типа поля. Для числовых типов это `0`, для булевых — `false`, а для ссылочных — `null`. Получается то же самое, что и при определении в Java некой переменной без инициализатора.

Учтите, что в Scala просто отбросить инициализатор `= _` нельзя. Если использовать код:

```
var celsius: Float
```

то получится объявление абстрактной, а не инициализированной переменной[1].

За определением переменной `celsius` следуют геттер по имени `fahrenheit` и сеттер `fahrenheit_=`, которые обращаются к той же температуре, но в градусах Фаренгейта. В листинге нет отдельного поля, содержащего значение текущей температуры в таких градусах. Вместо этого геттер и сеттер для значений в градусах Фаренгейта выполняют автоматическое преобразование из градусов Цельсия и в них же соответственно. Пример взаимодействия с объектом `Thermometer` выглядит следующим образом:

```
scala> val t = new Thermometer
t: Thermometer = 32.0F/0.0C

scala> t.celsius = 100
mutated t.celsius

scala> t
res3: Thermometer = 212.0F/100.0C

scala> t.fahrenheit = -40
mutated t.fahrenheit

scala> t
res4: Thermometer = -40.0F/-40.0C
```

18.3. Практический пример: моделирование дискретных событий

Далее в главе на расширенном примере будут показаны интересные способы возможного сочетания изменяемых объектов с функциями, являющимися значениями первого класса. Речь идет о проектировании и реализации симулятора цифровых схем. Эта задача разбита на несколько подзадач, каждая из которых интересна сама по себе.

---

[1] Абстрактные переменные будут рассматриваться в главе 20.

Сначала мы покажем весьма лаконичный язык для цифровых схем. Его определение подчеркнет общий метод встраивания предметно-ориентированных языков (domain-specific languages, DSL) в язык их реализации, подобный Scala. Затем представим простую, но всеобъемлющую среду для моделирования дискретных событий. Ее основной задачей будет являться отслеживание действий, выполняемых в ходе моделирования. И наконец, мы покажем, как структурировать и создавать программы дискретного моделирования. Цели создания таких программ — моделирование физических объектов объектами-симуляторами и использование среды для моделирования физического времени.

Этот пример взят из классического учебного пособия Structure and Interpretation of Computer Programs[1]. Наша ситуация отличается тем, что языком реализации является Scala, а не Scheme, и тем, что различные аспекты примера структурно выделены в четыре программных уровня. Первый относится к среде моделирования, второй — к основному пакету моделирования схем, третий касается библиотеки определяемых пользователем электронных схем, а четвертый, последний уровень предназначен для каждой моделируемой схемы как таковой. Каждый уровень выражен в виде класса, и более конкретные уровни являются наследниками более общих.

РЕЖИМ УСКОРЕННОГО ЧТЕНИЯ

На разбор примера моделирования дискретных событий, представленного в данной главе, потребуется некоторое время. Если вы считаете, что его лучше было бы потратить на дальнейшее изучение самого языка Scala, то можете перейти к чтению следующей главы.

18.4. Язык для цифровых схем

Начнем с краткого языка для описания цифровых схем, состоящих из *проводников* и *функциональных блоков*. По проводникам проходят *сигналы*, преобразованием которых занимаются функциональные блоки. Сигналы представлены булевыми значениями, где `true` используется для сигнала высокого уровня, а `false` — для сигнала низкого уровня.

Основные функциональные блоки (или *логические элементы*) показаны на рис. 18.1.

- *Блок «НЕ»* выполняет инверсию входного сигнала.
- *Блок «И»* устанавливает на своем выходе конъюнкцию сигналов на входе.
- *Блок «ИЛИ»* устанавливает на своем выходе дизъюнкцию сигналов на входе.

---

[1] *Abelson H., Sussman G. J.* Structure and Interpretation of Computer Programs. 2nd ed. — The MIT Press, 1996.

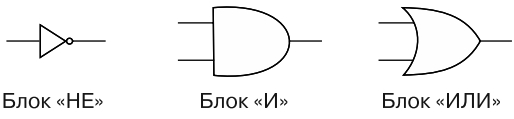


Рис. 18.1. Логические элементы

Этих логических элементов вполне достаточно для построения всех остальных функциональных блоков. У логических элементов существуют *задержки*, следовательно, сигнал на выходе элемента будет изменяться через некоторое время после изменения сигнала на его входе.

Элементы цифровой схемы будут описаны с применением набора классов и функций Scala. Сначала создадим класс `Wire` для проводников. Их можно сконструировать следующим образом:

```
val a = new Wire
val b = new Wire
val c = new Wire
```

или то же самое, но покороче:

```
val a, b, c = new Wire
```

Затем понадобятся три процедуры, создающие логические элементы:

```
def inverter(input: Wire, output: Wire)
def andGate(a1: Wire, a2: Wire, output: Wire)
def orGate(o1: Wire, o2: Wire, output: Wire)
```

Необычно то, что в силу имеющегося в Scala функционального уклона логические элементы в этих процедурах вместо возвращения в качестве результата сконструированных элементов конструируются в виде побочных эффектов. Например, вызов `inverter(a, b)` помещает элемент «НЕ» между проводниками `a` и `b`. Получается, что данная конструкция, основанная на побочном эффекте, позволяет упростить постепенное создание все более сложных схем. Вдобавок притом, что имена большинства методов происходят от глаголов, имена этих методов происходят от существительных, показывающих, какой именно элемент создается. Тем самым отображается декларативная природа DSL-языка: он должен давать описание электронной схемы, а не выполняемых в ней действий.

Из логических элементов могут создаваться более сложные функциональные блоки. Например, метод, показанный в листинге 18.6, создает полусумматор. Метод `halfAdder` получает два входных параметра, `a` и `b`, и выдает сумму `s`, определяемую как `s = (a + b) % 2`, и перенос в следующий разряд `c`, определяемый как `c = (a + b) / 2`. Схема полусумматора показана на рис. 18.2.

Листинг 18.6. Метод halfAdder

```
def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire) = {
  val d, e = new Wire
  orGate(a, b, d)
```

```
  andGate(a, b, c)
  inverter(c, e)
  andGate(d, e, s)
}
```

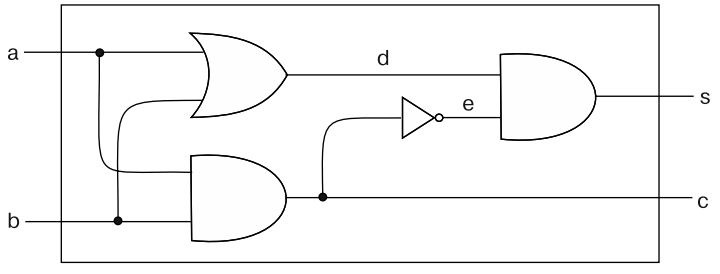


Рис. 18.2. Схема полусумматора

Обратите внимание: `halfAdder` является параметризованным функциональным блоком, как и три метода, составляющие логические элементы. Его можно использовать для составления более сложных схем. Например, в листинге 18.7 определяется полный одноразрядный сумматор (рис. 18.3), который получает два входных параметра, `a` и `b`, а также перенос из младшего разряда (carry-in) `cin` и выдает на выходе значение `sum`, определяемое как `sum = (a + b + cin) % 2`, и перенос в старший разряд (carry-out), определяемый как `cout = (a + b + cin) / 2`.

Листинг 18.7. Метод fullAdder

```
def fullAdder(a: Wire, b: Wire, cin: Wire,
    sum: Wire, cout: Wire) = {

  val s, c1, c2 = new Wire
  halfAdder(a, cin, s, c1)
  halfAdder(b, s, sum, c2)
  orGate(c1, c2, cout)
}
```

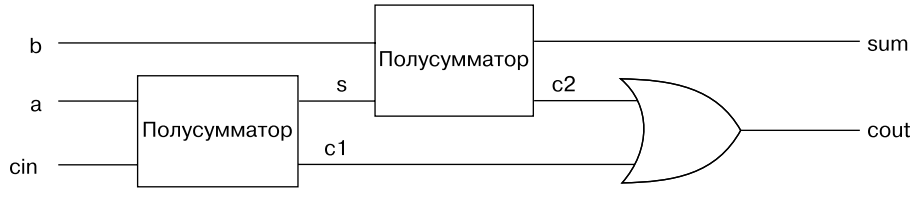


Рис. 18.3. Схема сумматора

Класс `Wire` и функции `inverter`, `andGate` и `orGate` представляют собой краткий язык, с помощью которого пользователи могут определять цифровые схемы. Это неплохой пример *внутреннего* DSL — предметно-ориентированного языка,

определенного в виде не какой-то самостоятельной реализации, а библиотеки в языке его реализации.

Реализацию DSL-языка электронных логических схем еще предстоит разработать. Цель определения схемы средствами DSL — моделирование электронной схемы, поэтому вполне разумно будет основой реализации DSL сделать общий API для моделирования дискретных событий. В следующих двух разделах мы представим первый API моделирования, а затем в качестве надстройки над ним покажем реализацию DSL-языка электронных логических схем.

18.5. API моделирования

API моделирования показан в листинге 18.8. Он состоит из класса `Simulation` в пакете `org.stairwaybook.simulation`. Наследниками этого класса являются конкретные библиотеки моделирования, дополняющие его предметно-ориентированную функциональность. В данном разделе представлены элементы класса `Simulation`.

Листинг 18.8. Класс Simulation

```
abstract class Simulation {

  type Action = () => Unit

  case class WorkItem(time: Int, action: Action)

  private var curtime = 0
  def currentTime: Int = curtime

  private var agenda: List[WorkItem] = List()

  private def insert(ag: List[WorkItem],
      item: WorkItem): List[WorkItem] = {

    if (ag.isEmpty || item.time < ag.head.time) item :: ag
    else ag.head :: insert(ag.tail, item)
  }

  def afterDelay(delay: Int)(block: => Unit) = {
    val item = WorkItem(currentTime + delay, () => block)
    agenda = insert(agenda, item)
  }

  private def next() = {
    (agenda: @unchecked) match {
      case item :: rest =>
        agenda = rest
        curtime = item.time
```

```
        item.action()
      }
  }

  def run() = {
    afterDelay(0) {
      println("*** simulation started, time = " +
          currentTime + " ***")
    }
    while (!agenda.isEmpty) next()
  }
}
```

При моделировании дискретных событий *действия*, определенные пользователем, выполняются в указанные *моменты времени*. Действия, определенные конкретными подклассами моделирования, имеют один и тот же тип:

```
type Action = () => Unit
```

Эта инструкция определяет `Action` в качестве псевдонима типа процедуры, принимающей пустой список параметров и возвращающей тип `Unit`. Тип `Action` является *членом типа* класса `Simulation`. Его можно рассматривать как гораздо более легко читаемое имя для типа `() => Unit`. Члены типов будут подробно рассмотрены в разделе 20.6.

Момент времени, в который выполняется действие, является моментом моделирования — он не имеет ничего общего со временем «настенных часов». Моменты времени моделирования представлены просто как целые числа. Текущий момент хранится в приватной переменной:

```
private var curtime: Int = 0
```

У переменной есть публичный метод доступа, извлекающий текущее время:

```
def currentTime: Int = curtime
```

Это сочетание приватной переменной с публичным методом доступа служит гарантией невозможности изменить текущее время за пределами класса `Simulation`. Обычно не нужно, чтобы моделируемые вами объекты манипулировали текущим временем, за исключением, возможно, случая, когда моделируется путешествие во времени.

Действие, которое должно быть выполнено в указанное время, называется *рабочим элементом*. Рабочие элементы реализуются следующим классом:

```
case class WorkItem(time: Int, action: Action)
```

Класс `WorkItem` сделан `case`-классом, чтобы иметь возможность получить следующие синтаксические удобства: для создания экземпляров класса можно использовать фабричный метод `WorkItem` и при этом без каких-либо усилий получить средства доступа к параметрам конструктора `time` и `action`. Следует также заметить, что класс `WorkItem` вложен в класс `Simulation`. Вложенные классы в Scala обрабатываются аналогично Java. Более подробно этот вопрос рассматривается в разделе 20.7.

В классе `Simulation` хранится *план действий* (agenda) всех остальных, еще не выполненных рабочих элементов. Они отсортированы по моделируемому времени, в которое должны быть запущены:

```
private var agenda: List[WorkItem] = List()
```

Список `agenda` будет храниться в надлежащем отсортированном порядке благодаря использованию метода `insert`, который обновляет этот список. Вызов метода `insert` в качестве единственного способа добавить рабочий элемент к плану действий можно увидеть в методе `afterDelay`:

```
def afterDelay(delay: Int)(block: => Unit) = {
  val item = WorkItem(currentTime + delay, () => block)
  agenda = insert(agenda, item)
}
```

Как следует из названия, этот метод вставляет действие, задаваемое блоком, в план действий, планируя время задержки его выполнения `delay` после текущего момента моделируемого времени. Например, следующий вызов создаст новый рабочий элемент к выполнению в моделируемое время `currentTime + delay`:

```
afterDelay(delay) { count += 1 }
```

Код, предназначенный для выполнения, содержится во втором аргументе метода. Формальный параметр имеет тип `=> Unit`, то есть это вычисление типа `Unit`, передаваемое по имени. Следует напомнить, что параметры, передаваемые по имени (by-name parameters), при передаче методу не вычисляются. Следовательно, в показанном ранее вызове значение `count` будет увеличено на единицу, только когда среда моделирования вызовет действие, сохраненное в рабочем элементе. Обратите внимание: `afterDelay` — каррированная функция. Это хороший пример разъясненного в разделе 9.5 принципа, согласно которому карринг может использоваться для выполнения вызовов методов, больше похожих на встроенный синтаксис языка.

Созданный рабочий элемент еще нужно вставить в план действий. Это делается с помощью метода `insert`, в котором поддерживается предварительное условие отсортированности плана по времени:

```
private def insert(ag: List[WorkItem],
    item: WorkItem): List[WorkItem] = {

  if (ag.isEmpty || item.time < ag.head.time) item :: ag
  else ag.head :: insert(ag.tail, item)
}
```

Ядро класса `Simulation` определяется методом `run`:

```
def run() = {
  afterDelay(0) {
    println("*** simulation started, time = " +
        currentTime + " ***")
  }
```

```
  while (!agenda.isEmpty) next()
}
```

Этот метод периодически берет первый элемент из плана действий, удаляет его из данного плана и выполняет. Он продолжает свою работу до тех пор, пока в плане не останется элементов для выполнения. Каждый шаг выполняется с помощью вызова метода `next`, имеющего такое определение:

```
private def next() = {
  (agenda: @unchecked) match {
    case item :: rest =>
      agenda = rest
      curtime = item.time
      item.action()
  }
}
```

В методе `next` текущий план действий разбивается с помощью сопоставления с образцом на первый элемент `item` и весь остальной список рабочих элементов `rest`. Первый элемент из текущего плана удаляется, моделируемое время `curtime` устанавливается на время рабочего элемента, и выполняется действие рабочего элемента.

Обратите внимание: `next` может быть вызван, только если план действий еще не пуст. Варианта для пустого плана нет, поэтому при попытке запуска `next` в отношении пустого списка `agenda` будет выдано исключение `MatchError`.

В действительности же компилятор Scala обычно выдает предупреждение, что для списка не указан один из возможных паттернов:

```
Simulator.scala:19: warning: match is not exhaustive!
missing combination            Nil

    agenda match {
    ^
one warning found
```

В данном случае неуказанный вариант никакой проблемы не создает, поскольку известно, что `next` вызывается только в отношении непустого плана действий. Поэтому может возникнуть желание отключить предупреждение. Как было показано в разделе 15.5, это можно сделать, добавив к выражению селектора сопоставления с образцом аннотацию `@unchecked`. Именно поэтому в коде `Simulation` используется `(agenda: @unchecked) match`, а не `agenda match`.

И это правильно. Объем кода для среды моделирования может показаться весьма скромным. Может возникнуть вопрос: а как эта среда вообще может поддерживать содержательное моделирование, если всего лишь выполняет список рабочих элементов? В действительности же эффективность среды моделирования определяется тем фактом, что действия, сохраненные в рабочих элементах, в ходе своего выполнения могут самостоятельно добавлять следующие рабочие элементы в план действий. Тем самым открывается возможность получить из вычисления простых начальных действий довольно продолжительную симуляцию.

18.6. Моделирование электронной логической схемы

Следующим шагом станет использование среды моделирования в целях реализации предметно-ориентированного языка для логических схем, показанного в разделе 18.4. Следует напомнить, что DSL логических схем состоит из класса для проводников и методов, создающих логические элементы «И», «ИЛИ» и «НЕ». Все это содержится в классе `BasicCircuitSimulation`, который расширяет среду моделирования. Он показан в листинге 18.9.

Листинг 18.9. Первая половина класса BasicCircuitSimulation

```
package org.stairwaybook.simulation

abstract class BasicCircuitSimulation extends Simulation {

  def InverterDelay: Int
  def AndGateDelay: Int
  def OrGateDelay: Int

  class Wire {

    private var sigVal = false
    private var actions: List[Action] = List()

    def getSignal = sigVal

    def setSignal(s: Boolean) =
      if (s != sigVal) {
        sigVal = s
        actions foreach (_ ())
      }

    def addAction(a: Action) = {
      actions = a :: actions
      a()
    }
  }

  def inverter(input: Wire, output: Wire) = {
    def invertAction() = {
      val inputSig = input.getSignal
      afterDelay(InverterDelay) {
        output setSignal !inputSig
      }
    }
    input addAction invertAction
  }

  def andGate(a1: Wire, a2: Wire, output: Wire) = {
    def andAction() = {
```

```
      val a1Sig = a1.getSignal
      val a2Sig = a2.getSignal
      afterDelay(AndGateDelay) {
        output setSignal (a1Sig & a2Sig)
      }
    }
    a1 addAction andAction
    a2 addAction andAction
  }

  def orGate(o1: Wire, o2: Wire, output: Wire) = {
    def orAction() = {
      val o1Sig = o1.getSignal
      val o2Sig = o2.getSignal
      afterDelay(OrGateDelay) {
        output setSignal (o1Sig | o2Sig)
      }
    }
    o1 addAction orAction
    o2 addAction orAction
  }

  def probe(name: String, wire: Wire) = {
    def probeAction() = {
      println(name + " " + currentTime +
          " new-value = " + wire.getSignal)
    }
    wire addAction probeAction
  }
}
```

В классе `BasicCircuitSimulation` объявляются три абстрактных метода, представляющих задержки основных логических элементов: `InverterDelay`, `AndGateDelay` и `OrGateDelay`. Настоящие задержки на уровне этого класса неизвестны, поскольку зависят от технологии моделируемых логических микросхем. Поэтому задержки в классе `BasicCircuitSimulation` остаются абстрактными, и их конкретное определение делегируется подклассам[1]. Далее мы рассмотрим реализацию остальных членов класса `BasicCircuitSimulation`.

Класс Wire

Проводникам нужно поддерживать три основных действия:

- `getSignal: Boolean` возвращает текущий сигнал в проводнике;
- `setSignal(sig: Boolean)` выставляет сигнал проводника в `sig`;

[1] Имена этих методов задержки начинаются с прописных букв, поскольку представляют собой константы. Но это методы, и они могут быть переопределены в подклассах. Как те же вопросы решаются с помощью val-переменных, мы покажем в разделе 20.3.

- `addAction(p: Action)` прикрепляет указанную процедуру `p` к *действиям* проводника. Замысел заключается в том, чтобы все процедуры действий, прикрепленные к какому-либо проводнику, выполнялись всякий раз, когда сигнал на проводнике изменяется. Как правило, действия добавляются к проводнику подключенными к нему компонентами. Прикрепленное действие выполняется в момент его добавления к проводнику, а после этого всякий раз при изменении сигнала в проводнике.

Реализация класса `Wire` имеет следующий вид:

```
class Wire {

  private var sigVal = false
  private var actions: List[Action] = List()

  def getSignal = sigVal

  def setSignal(s: Boolean) =
    if (s != sigVal) {
      sigVal = s
      actions foreach (_ ())
    }

  def addAction(a: Action) = {
    actions = a :: actions
    a()
  }
}
```

Состояние проводника формируется двумя приватными переменными. Переменная `sigVal` представляет текущий сигнал, а переменная `actions` — процедуры действий, прикрепленные в данный момент к проводнику. В реализациях методов представляет интерес только та часть, которая относится к методу `setSignal`: когда сигнал проводника изменяется, в переменной `sigVal` сохраняется новое значение. Кроме того, выполняются все действия, прикрепленные к проводнику. Обратите внимание на используемую для этого сокращенную форму синтаксиса: выражение `actions foreach (_ ())` вызывает применение функции `_ ()` к каждому элементу в списке действий. В соответствии с описанием, приведенным в разделе 8.5, функция `_ ()` является сокращенной формой записи для `f => f ()`, то есть получает функцию (назовем ее `f`) и применяет ее к пустому списку параметров.

Метод inverter

Единственный результат создания инвертора — то, что действие устанавливается на его входном проводнике. Данное действие вызывается при его установке, а затем всякий раз при изменении сигнала на входе. Эффект от действия заключается в установке выходного значения (с помощью `setSignal`) на отрицание его входного значения. Поскольку у логического элемента «НЕ» имеется задержка, это изменение должно наступить только по прошествии определенного количества единиц

моделируемого времени, хранящегося в переменной `InverterDelay`, после изменения входного значения и выполнения действия. Эти обстоятельства подсказывают следующий вариант реализации:

```
def inverter(input: Wire, output: Wire) = {
  def invertAction() = {
    val inputSig = input.getSignal
    afterDelay(InverterDelay) {
      output setSignal !inputSig
    }
  }
  input addAction invertAction
}
```

Эффект метода `inverter` заключается в добавлении действия `invertAction` к `input`. При вызове данного действия берется входной сигнал и устанавливается еще одно действие, инвертирующее выходной сигнал в плане действий моделирования. Это другое действие должно быть выполнено по прошествии того количества единиц времени моделирования, которое хранится в переменной `InverterDelay`. Обратите внимание на то, как для создания нового рабочего элемента, предназначенного для выполнения в будущем, в методе используется метод `afterDelay`.

Методы andGate и orGate

Реализация моделирования логического элемента «И» аналогична реализации моделирования элемента «НЕ». Цель — выставить на выходе конъюнкцию его входных сигналов. Это должно произойти по прошествии того количества единиц времени моделирования, которое хранится в переменной `AndGateDelay`, после изменения любого из его двух входных сигналов. Стало быть, подойдет следующая реализация:

```
def andGate(a1: Wire, a2: Wire, output: Wire) = {
  def andAction() = {
    val a1Sig = a1.getSignal
    val a2Sig = a2.getSignal
    afterDelay(AndGateDelay) {
      output setSignal (a1Sig & a2Sig)
    }
  }
  a1 addAction andAction
  a2 addAction andAction
}
```

Эффект от вызова метода `andGate` заключается в добавлении действия `andAction` к обоим входным проводникам, `a1` и `a2`. При вызове данного действия берутся оба входных сигнала и устанавливается еще одно действие, которое выдает выходной сигнал в виде конъюнкции обоих входных сигналов. Это другое действие должно быть выполнено по прошествии того количества единиц времени моделирования, которое хранится в переменной `AndGateDelay`. Учтите, что при смене любого сигнала на входных проводниках выход должен вычисляться заново. Именно поэтому

одно и то же действие `andAction` устанавливается на каждом из двух входных проводников, `a1` и `a2`. Метод `orGate` реализуется аналогичным образом, за исключением того, что моделирует логическую операцию «ИЛИ», а не «И».

Вывод симуляции

Для запуска симулятора нужен способ проверки изменения сигналов на проводниках. Чтобы выполнить эту задачу, можно смоделировать действие проверки (пробы) проводника:

```
def probe(name: String, wire: Wire) = {
  def probeAction() = {
    println(name + " " + currentTime +
        " new-value = " + wire.getSignal)
  }
  wire addAction probeAction
}
```

Эффект от процедуры `probe` заключается в установке на заданный проводник действия `probeAction`. Как обычно, установленное действие выполняется всякий раз при изменении сигнала на проводнике. В данном случае он просто выводит на стандартное устройство название проводника (которое передается `probe` в качестве первого параметра), а также текущее моделируемое время и новое значение проводника.

Запуск симулятора

После всех этих приготовлений настало время посмотреть на симулятор в действии. Для определения конкретной симуляции нужно выполнить наследование из класса среды моделирования. Чтобы увидеть кое-что интересное, будет создан абстрактный класс моделирования, расширяющий `BasicCircuitSimulation` и содержащий определения методов для полусумматора и сумматора в том виде, в котором они были представлены в листингах 18.6 и 18.7 соответственно (см. выше). Этот класс, который будет назван `CircuitSimulation`, показан в листинге 18.10.

Листинг 18.10. Класс CircuitSimulation

```
package org.stairwaybook.simulation

abstract class CircuitSimulation
  extends BasicCircuitSimulation {

  def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire) = {
    val d, e = new Wire
    orGate(a, b, d)
    andGate(a, b, c)
    inverter(c, e)
    andGate(d, e, s)
  }
```

```
  def fullAdder(a: Wire, b: Wire, cin: Wire,
      sum: Wire, cout: Wire) = {

    val s, c1, c2 = new Wire
    halfAdder(a, cin, s, c1)
    halfAdder(b, s, sum, c2)
    orGate(c1, c2, cout)
  }
}
```

Конкретная модель логической схемы будет объектом-наследником класса `CircuitSimulation`. Этому объекту по-прежнему необходимо зафиксировать задержки на логических элементах в соответствии с технологией реализации моделируемой логической микросхемы. И наконец, понадобится также определить конкретную моделируемую схему.

Эти шаги можно проделать в интерактивном режиме в интерпретаторе Scala:

```
scala> import org.stairwaybook.simulation._
import org.stairwaybook.simulation._
```

Сначала займемся задержками логических элементов. Определим объект (назовем его `MySimulation`), предоставляющий несколько чисел:

```
scala> object MySimulation extends CircuitSimulation {
         def InverterDelay = 1
         def AndGateDelay = 3
         def OrGateDelay = 5
       }
defined module MySimulation
```

Поскольку предполагается периодически получать доступ к элементам объекта `MySimulation`, импортирование этого объекта укоротит последующий код:

```
scala> import MySimulation._
import MySimulation._
```

Далее займемся схемой. Определим четыре проводника и поместим пробы на два из них:

```
scala> val input1, input2, sum, carry = new Wire
input1: MySimulation.Wire =
    BasicCircuitSimulation$Wire@111089b
input2: MySimulation.Wire =
    BasicCircuitSimulation$Wire@14c352e
sum: MySimulation.Wire =
    BasicCircuitSimulation$Wire@37a04c
carry: MySimulation.Wire =
    BasicCircuitSimulation$Wire@1fd10fa

scala> probe("sum", sum)
sum 0 new-value = false

scala> probe("carry", carry)
carry 0 new-value = false
```

Обратите внимание: пробы немедленно выводят выходные данные. Дело в том, что каждое действие, установленное на проводнике, первый раз выполняется при его установке.

Теперь определим подключение к проводникам полусумматора:

```
scala> halfAdder(input1, input2, sum, carry)
```

И наконец, установим один за другим сигналы на двух входящих проводниках на `true` и запустим моделирование:

```
scala> input1 setSignal true

scala> run()
*** simulation started, time = 0 ***
sum 8 new-value = true

scala> input2 setSignal true

scala> run()
*** simulation started, time = 8 ***
carry 11 new-value = true
sum 15 new-value = false
```

Резюме

В данной главе мы собрали воедино две на первый взгляд несопоставимые технологии: изменяемое состояние и функции высшего порядка. Изменяемое состояние было использовано для моделирования физических объектов, состояние которых со временем изменяется. Функции высшего порядка были применены в среде моделирования в целях выполнения действий в указанные моменты моделируемого времени. Они также были использованы в моделировании логических схем в качестве *триггеров*, связывающих действия с изменениями состояния. Попутно мы показали простой способ определить предметно-ориентированный язык в виде библиотеки. Вероятно, для одной главы этого вполне достаточно.

Если вас привлекла эта тема, то можете попробовать создать дополнительные примеры моделирования. Можно объединить полусумматоры и сумматоры для формирования более крупных схем или на основе ранее определенных логических элементов разработать новые схемы и смоделировать их. В главе 19 мы рассмотрим имеющуюся в Scala параметризацию типов и покажем еще один пример, который сочетает в себе функциональный и императивный подходы, дающие весьма неплохое решение.

19 Параметризация типов

В этой главе мы рассмотрим детали параметризации типов в Scala. Попутно продемонстрируем несколько техник скрытия информации, представленных в главе 13, на конкретном примере: проектирования класса для чисто функциональных очередей. Параметризация типов и скрытие информации рассматриваются вместе, поскольку скрытие может пригодиться для получения более общих вариантов аннотаций параметризации типов.

Параметризация типов позволяет создавать обобщенные классы и трейты. Например, множества имеют обобщенный характер и получают параметр типа: они определяются как `Set[T]`. В результате любой отдельно взятый экземпляр множества может иметь тип `Set[String]`, `Set[Int]` и т. д., но должен быть множеством *чего-либо*. В отличие от языка Java, в котором разрешено использовать сырые типы (raw types), Scala требует указывать параметры типа. Вариантность определяет взаимоотношения наследования параметризованных типов, к примеру таких, при которых `Set[String]` является подтипом `Set[AnyRef]`.

Глава состоит из трех частей. В первой разрабатывается структура данных для чисто функциональных очередей. Во второй разрабатываются технологические приемы скрытия внутреннего представления деталей этой структуры. В третьей объясняется вариантность параметров типов и то, как она взаимодействует со скрытием информации.

19.1. Функциональные очереди

Функциональная очередь представляет собой структуру данных с тремя операциями:

- `head` — возвращает первый элемент очереди;
- `tail` — возвращает очередь без первого элемента;
- `enqueue` — возвращает новую очередь с заданным элементом, добавленным в ее конец.

В отличие от изменяемой функциональная очередь не изменяет свое содержимое при добавлении элемента. Вместо этого возвращается новая очередь, содержащая элемент. В данной главе наша цель — создать класс по имени `Queue`, работающий так:

```
scala> val q = Queue(1, 2, 3)
q: Queue[Int] = Queue(1, 2, 3)

scala> val q1 = q enqueue 4
q1: Queue[Int] = Queue(1, 2, 3, 4)

scala> q
res0: Queue[Int] = Queue(1, 2, 3)
```

Будь у `Queue` изменяемая реализация, операция `enqueue` в показанной ранее второй строке ввода повлияла бы на содержимое `q`: по сути, после этой операции оба результата, и `q1`, и исходная очередь `q`, будут содержать последовательность 1, 2, 3, 4. А для функциональной очереди добавленное значение обнаруживается только в результате `q1`, но не в очереди `q`, в отношении которой выполнялась операция.

Кроме того, чистые функциональные очереди слегка похожи на списки. Обе эти коллекции имеют так называемую *абсолютно постоянную* структуру данных, где старые версии остаются доступными даже после расширений или модификаций. Обе они поддерживают операции `head` и `tail`. Но список растет, как правило, с начала с помощью операции `::`, а очередь — с конца с помощью операции `enqueue`.

Как добиться эффективной реализации очереди? В идеале функциональная (неизменяемая) очередь не должна иметь высоких издержек, существенно больших по сравнению с императивной (изменяемой) очередью. То есть все три операции: `head`, `tail` и `enqueue` — должны выполняться за постоянное время.

Одним из простых подходов к реализации функциональной очереди станет использование списка в качестве типа представления. Тогда `head` и `tail` — просто аналогичные операции над списком, а `enqueue` — конкатенация.

В таком варианте получится следующая реализация:

```
class SlowAppendQueue[T](elems: List[T]) { // Неэффективное решение
  def head = elems.head
  def tail = new SlowAppendQueue(elems.tail)
  def enqueue(x: T) = new SlowAppendQueue(elems ::: List(x))
}
```

Проблемной в данной реализации является операция `enqueue`. На нее уходит время, пропорциональное количеству элементов, хранящихся в очереди. Если требуется постоянное время добавления, то можно также попробовать изменить в списке, представляющем очередь, порядок следования элементов на обратный, чтобы последний добавляемый элемент стал в списке первым. Тогда получится такая реализация:

```
class SlowHeadQueue[T](smele: List[T]) { // Неэффективное решение
  // smele — это реверсированный elems
  def head = smele.last
```

```
  def tail = new SlowHeadQueue(smele.init)
  def enqueue(x: T) = new SlowHeadQueue(x :: smele)
}
```

Теперь у операции `enqueue` постоянное время выполнения, а вот у `head` и `tail` — нет. На их выполнение теперь уходит время, пропорциональное количеству элементов, хранящихся в очереди.

При изучении этих двух примеров реализация, в которой на все три операции будет затрачиваться постоянное время, не представляется такой уж простой. И действительно, возникают серьезные сомнения в возможности подобной реализации! Но, воспользовавшись сочетанием двух операций, можно подойти к желаемому результату очень близко. Замысел состоит в представлении очереди в виде двух списков: `leading` и `trailing`. Список `leading` содержит элементы, которые располагаются от конца к началу, а элементы списка `trailing` следуют из начала в конец очереди, то есть в обратном порядке. Содержимое всей очереди в любой момент времени равно коду `leading ::: trailing.reverse`.

Теперь, чтобы добавить элемент, следует просто провести конс-операцию в отношении списка `trailing`, воспользовавшись оператором `::`, и тогда операция `enqueue` будет выполняться за постоянное время. Это значит, если изначально пустая очередь выстраивается на основе последовательно проведенных операций `enqueue`, то список `trailing` будет расти, а список `leading` останется пустым. Затем перед выполнением первой операции `head` или `tail` в отношении пустого списка `leading` весь список `trailing` копируется в `leading` в обратном порядке следования элементов. Это делается с помощью операции по имени `mirror`. Реализация очередей с использованием данного подхода показана в листинге 19.1.

Листинг 19.1. Базовая функциональная очередь

```
class Queue[T](
  private val leading: List[T],
  private val trailing: List[T]
) {
  private def mirror =
    if (leading.isEmpty)
      new Queue(trailing.reverse, Nil)
    else
      this

  def head = mirror.leading.head

  def tail = {
    val q = mirror
    new Queue(q.leading.tail, q.trailing)
  }

  def enqueue(x: T) =
    new Queue(leading, x :: trailing)
}
```

Какова вычислительная сложность этой реализации очереди? Операция `mirror` может занять время, пропорциональное количеству элементов очереди, но только

при условии, что список `leading` пуст. Если же нет, то возврат из метода происходит немедленно. Поскольку `head` и `tail` вызывают `mirror`, то их вычислительная сложность также может иметь линейную зависимость от размера очереди. Но чем длиннее становится очередь, тем реже вызывается `mirror`.

И действительно, допустим, есть очередь длиной *n* с пустым списком `leading`. Тогда операции `mirror` придется скопировать в обратном порядке список длиной *n*. Однако следующий раз, когда операции `mirror` придется делать что-либо, наступит только по опустошении списка `leading`, что произойдет после *n* операций `tail`. То есть вы можете расплатиться за каждую из этих *n* операций `tail` одной *n*-ной от вычислительной сложности операции `mirror`, что означает постоянный объем работы. При условии, что операции `head`, `tail` и `enqueue` используются примерно с одинаковой частотой, *амортизированная* вычислительная сложность является, таким образом, константой для каждой операции. Следовательно, функциональные очереди асимптотически так же эффективны, как и изменяемые очереди.

Но этот аргумент следует сопроводить некоторыми оговорками. Во-первых, речь шла только об асимптотическом поведении, а постоянно действующие факторы могут несколько различаться. Во-вторых, аргумент основывается на том, что операции `head`, `tail` и `enqueue` вызываются с примерно одинаковой частотой. Если операция `head` вызывается намного чаще, чем остальные две, то данный аргумент утратит силу, поскольку каждый вызов `head` может повлечь за собой весьма затратную реорганизацию списка с помощью операции `mirror`. Негативных последствий, названных во второй оговорке, можно избежать, поскольку есть возможность сконструировать функциональные очереди таким образом, что в последовательности операций `head` реорганизация потребуется только для первой из них. Как это делается, мы покажем в конце главы.

19.2. Скрытие информации

Теперь по эффективности реализация класса `Queue`, показанная в листинге 19.1 (см. выше), нас вполне устраивает. Конечно, вы можете возразить, что за эту эффективность пришлось расплачиваться неоправданно подробной детализацией. Глобально доступный конструктор `Queue` получает в качестве параметров два списка, в одном из которых элементы следуют в обратном порядке, что вряд ли совпадает с интуитивным представлением об очереди. Нам нужен способ, позволяющий скрыть этот конструктор от кода клиента. Некоторые способы решения данной задачи на Scala мы покажем в текущем разделе.

Приватные конструкторы и фабричные методы

В Java конструктор можно скрыть, объявив его приватным. В Scala первичный конструктор не имеет явно указываемого определения — подразумевается, что он автоматически определяется с параметрами и телом класса. Тем не менее, как

показано в листинге 19.2, скрыть первичный конструктор можно, добавив перед списком параметров класса модификатор `private`.

Листинг 19.2. Скрытие первичного конструктора путем превращения его в приватный

```
class Queue[T] private (
  private val leading: List[T],
  private val trailing: List[T]
)
```

Модификатор `private`, указанный между именем класса и его параметрами, служит признаком того, что конструктор класса `Queue` является приватным: доступ к нему можно получить только изнутри самого класса и из его объекта-компаньона. Имя класса `Queue` по-прежнему остается публичным, поэтому вы можете использовать его в качестве типа, но не можете вызвать его конструктор:

```
scala> new Queue(List(1, 2), List(3))
       ^
       error: constructor Queue in class Queue cannot be accessed in object $iw
```

Теперь, когда первичный конструктор класса `Queue` уже не может быть вызван из кода клиента, нужен какой-то другой способ создания новых очередей. Одна из возможностей — добавить вспомогательный конструктор:

```
def this() = this(Nil, Nil)
```

Этот конструктор, показанный в предыдущем примере, создает пустую очередь. В качестве уточнения он может получать список исходных элементов очереди:

```
def this(elems: T*) = this(elems.toList, Nil)
```

Следует напомнить, что в соответствии с описанием из раздела 8.8 `T*` — форма записи для повторяющихся параметров.

Еще одна возможность состоит в добавлении фабричного метода, создающего очередь из последовательности исходных элементов. Прямой путь сделать это — определить объект `Queue`, который имеет такое же имя, как и определяемый класс, и, как показано в листинге 19.3, содержит метод `apply`.

Листинг 19.3. Фабричный метод apply в объекте-компаньоне

```
object Queue {
  // Создает очередь с исходными элементами xs
  def apply[T](xs: T*) = new Queue[T](xs.toList, Nil)
}
```

Помещая этот объект в тот же самый исходный файл, в котором находится определение класса `Queue`, вы превращаете объект в объект-компаньон этого класса. В разделе 13.5 было показано: объект-компаньон имеет такие же права доступа, что и его класс. Поэтому метод `apply` в объекте `Queue` может создать новый объект `Queue`, несмотря на то что конструктор класса `Queue` является приватным.

Обратите внимание: по причине вызова фабричным методом метода `apply` клиенты могут создавать очереди с помощью выражений вида `Queue(1, 2, 3)`. Данное выражение разворачивается в `Queue.apply(1, 2, 3)`, поскольку `Queue` — объект,

заменяющий функцию. В результате этого клиенты видят Queue в качестве глобально определенного фабричного метода. На самом же деле в Scala нет методов с глобальной областью видимости, поскольку каждый метод должен быть заключен в объект или класс. Но, используя методы по имени apply внутри глобальных объектов, вы можете поддерживать схемы, похожие на вызов глобальных методов.

Альтернативный вариант: приватные классы

Приватные конструкторы и приватные члены класса — всего лишь один из способов скрыть инициализацию и представить класс. Еще один более радикальный способ — скрытие самого класса и экспорт только трейта, показывающего публичный интерфейс класса. Код реализации такой конструкции представлен в листинге 19.4. В нем показан трейт Queue, в котором объявляются методы head, tail и enqueue. Все три метода реализованы в подклассе QueueImpl, который является приватным подклассом внутри класса объекта Queue. Тем самым клиентам открывается доступ к той же информации, что и раньше, но с помощью другого приема. Вместо скрытия отдельно взятых конструкторов и методов в этой версии скрывается весь реализующий очереди класс.

Листинг 19.4. Абстракция типа для функциональных очередей

```
trait Queue[T] {
  def head: T
  def tail: Queue[T]
  def enqueue(x: T): Queue[T]
}

object Queue {

  def apply[T](xs: T*): Queue[T] =
    new QueueImpl[T](xs.toList, Nil)

  private class QueueImpl[T](
    private val leading: List[T],
    private val trailing: List[T]
  ) extends Queue[T] {

    def mirror =
      if (leading.isEmpty)
        new QueueImpl(trailing.reverse, Nil)
      else
        this

    def head: T = mirror.leading.head

    def tail: QueueImpl[T] = {
      val q = mirror
      new QueueImpl(q.leading.tail, q.trailing)
    }
```

```
    def enqueue(x: T) =
      new QueueImpl(leading, x :: trailing)
  }
}
```

19.3. Аннотации вариантности

Элемент `Queue` согласно определению в листинге 19.4 — трейт, но не тип, поскольку получает параметра типа. В результате вы не можете создавать переменные типа `Queue`:

```
scala> def doesNotCompile(q: Queue) = {}
                              ^
       error: class Queue takes type parameters
```

Вместо этого `Queue` позволяет указывать *параметризованные* типы, такие как `Queue[String]`, `Queue[Int]` или `Queue[AnyRef]`:

```
scala> def doesCompile(q: Queue[AnyRef]) = {}
doesCompile: (q: Queue[AnyRef])Unit
```

Таким образом, `Queue` — трейт, а `Queue[String]` — тип. `Queue` также называют *конструктором типа*, поскольку вы можете сконструировать тип с его участием, указав параметр типа. (Это аналогично конструированию экземпляра объекта с использованием самого обычного конструктора с указанием параметра значения.) Конструктор типа `Queue` генерирует семейство типов, включающее `Queue[Int]`, `Queue[String]` и `Queue[AnyRef]`.

Можно также сказать, что `Queue` — *обобщенный* трейт. (Классы и трейты, которые получают параметры типа, являются обобщенными, а вот типы, генерируемые ими, являются параметризованными, а не обобщенными.) Понятие «обобщенный» означает, что вы определяете множество конкретных типов, используя один обобщенно написанный класс или трейта. Например, трейт `Queue` в листинге 19.4 (см. выше) определяет обобщенную очередь. Конкретными очередями будут `Queue[Int]`, `Queue[String]` и т. д.

Сочетание параметров типа и системы подтипов вызывает ряд интересных вопросов. Например, существуют ли какие-то особые подтиповые отношения между членами семейства типов, генерируемого `Queue[T]`? Конкретнее говоря, следует ли рассматривать `Queue[String]` как подтип `Queue[AnyRef]`? Или в более широком смысле если `S` — подтип `T`, то следует ли рассматривать `Queue[S]` как подтип `Queue[T]`? Если да, то можно сказать, что трейт `Queue` *ковариантный* (или «гибкий») в своем параметре типа `T`. Или же, поскольку у него всего один параметр типа, можно просто сказать, что `Queue`-очереди ковариантны. Такая ковариантность `Queue` будет означать, к примеру, что вы можете передать `Queue[String]` ранее показанному методу `doesCompile`, который принимает параметр значения типа `Queue[AnyRef]`.

Интуитивно все это выглядит хорошо, поскольку очередь из `String`-элементов похожа на частный случай очереди из `AnyRef`-элементов. Но в Scala у обобщенных типов изначально имеется *нонвариантная* (или жесткая) подтипизация. То есть при использовании трейта `Queue`, определенного в показанном выше листинге 19.4, очереди с различными типами элементов никогда не будут состоять в подтиповых отношениях. `Queue[String]` не будет использоваться как `Queue[AnyRef]`. Но получить ковариантность (гибкость) подтипизации очередей можно следующим изменением первой строчки определения `Queue`:

```
trait Queue[+T] { ... }
```

Указанный знак плюс (`+`) в качестве префикса формального параметра типа показывает, что подтипизация в этом параметре ковариантна (гибка). Добавляя этот единственный знак, вы сообщаете Scala о необходимости, к примеру, рассматривать тип `Queue[String]` как подтип `Queue[AnyRef]`. Компилятор проверит факт определения `Queue` в соответствии со способом, предполагаемым подобной подтипизацией.

Помимо префикса `+`, существует префикс `-`, который показывает *контравариантность* подтипизации. Если определение `Queue` имеет вид:

```
trait Queue[-T] { ... }
```

и если тип `T` — подтип типа `S`, то это будет означать, что `Queue[S]` — подтип `Queue[T]` (что в случае с очередями было бы довольно неожиданно!). Ковариантность, контравариантность или нонвариантость параметра типа называются *вариантностью* параметров. Знаки `+` и `-`, которые могут размещаться рядом с параметрами типа, называются *аннотациями вариантности*.

В чисто функциональном мире многие типы по своей природе ковариантны (гибки). Но ситуация становится иной, как только появляются изменяемые данные. Чтобы выяснить, почему так происходит, рассмотрим показанный в листинге 19.5 простой тип одноэлементных ячеек, допускающих чтение и запись.

Листинг 19.5. Нонвариантный (жесткий) класс Cell

```
class Cell[T](init: T) {
  private[this] var current = init
  def get = current
  def set(x: T) = { current = x }
}
```

Показанный здесь тип `Cell` объявлен нонвариантным (жестким). Ради аргументации представим на время, что `Cell` был объявлен ковариантным, то есть был объявлен класс `Cell[+T]`, и этот код передан компилятору Scala. (Этого делать не стоит, и скоро мы объясним почему.) Значит, можно сконструировать следующую проблематичную последовательность инструкций:

```
val c1 = new Cell[String]("abc")
val c2: Cell[Any] = c1
c2.set(1)
val s: String = c1.get
```

Если рассмотреть строки по отдельности, то все они выглядят вполне нормально. В первой строке создается строковая ячейка, которая сохраняется в `val`-переменной по имени `c1`. Во второй строке определяется новая `val`-переменная `c2`, имеющая тип `Cell[Any]`, которая инициализируется значением переменной `c1`. Кажется, все в порядке, поскольку экземпляры класса `Cell` считаются ковариантными. В третьей строке для `c2` устанавливается значение `1`. С этим тоже все в порядке, так как присваиваемое значение `1` — экземпляр, относящийся к объявленному для `c2` типу элемента `Any`. И наконец, в последней строке значение элемента `c1` присваивается строковой переменной. Здесь нет ничего странного, поскольку с обеих сторон выражения находятся значения одного и того же типа. Но если взять все в совокупности, то эти четыре строки в конечном счете присваивают целочисленное значение `1` строковому значению `s`. Это явное нарушение целостности типа.

Какая из операций вызывает сбой в ходе выполнения кода? Видимо, вторая, в которой используется ковариантная подтипизация. Все другие операции слишком простые и базовые. Стало быть, ячейка `Cell`, хранящая значение типа `String`, *не* является также ячейкой `Cell`, хранящей значение типа `Any`, поскольку есть вещи, которые можно делать с `Cell` из `Any`, но нельзя делать с `Cell` из `String`. К примеру, в отношении `Cell` из `String` нельзя использовать `set` с `Int`-аргументом.

Получается, если передать ковариантную версию `Cell` компилятору Scala, то будет выдана ошибка компиляции:

```
Cell.scala:7: error: covariant type T occurs in
contravariant position in type T of value x
   def set(x: T) = current = x
               ^
```

Вариантность и массивы

Интересно сравнить данное поведение с массивами в Java. В принципе, массивы подобны ячейкам, за исключением того, что могут иметь несколько элементов. При этом массивы в Java считаются ковариантными.

Пример, аналогичный работе с ячейкой, можно проверить в работе с массивами Java:

```
// Это код Java
String[] a1 = { "abc" };
Object[] a2 = a1;
a2[0] = new Integer(17);
String s = a1[0];
```

Если запустить данный пример, то окажется, что он пройдет компиляцию. Но в ходе выполнения, когда элементу `a2[0]` будет присваиваться значение `Integer`, программа сгенерирует исключение `ArrayStore`:

```
Exception in thread "main" java.lang.ArrayStoreException:
java.lang.Integer
        at JavaArrays.main(JavaArrays.java:8)
```

Дело в том, что в Java сохраняется тип элемента массива во время выполнения программы. При каждом обновлении элемента массива новое значение элемента проверяется на принадлежность к сохраненному типу. Если оно не является экземпляром этого типа, то выдается исключение `ArrayStore`.

Может возникнуть вопрос: а почему в Java принята такая на вид небезопасная и затратная конструкция? Отвечая на данный вопрос, Джеймс Гослинг (James Gosling), основной изобретатель языка Java, говорил, что создатели хотели получить простые средства обобщенной обработки массивов. Например, им хотелось получить возможность писать метод сортировки всех элементов массива с использованием следующей сигнатуры, которая получает массив из `Object`-элементов:

```
void sort(Object[] a, Comparator cmp) { ... }
```

Ковариантность массивов требовалась для того, чтобы этому методу сортировки можно было передавать массивы произвольных ссылочных типов. Разумеется, после появления в Java обобщенных типов (дженериков) подобный метод сортировки уже мог быть написан с параметрами типа, поэтому необходимость в ковариантности массивов отпала. Сохранение ее до наших дней обусловлено соображениями совместимости.

Scala старается быть чище, чем Java, и не считает массивы ковариантными. Вот что получится, если перевести первые две строки кода примера с массивом на язык Scala:

```
scala> val a1 = Array("abc")
a1: Array[String] = Array(abc)

scala> val a2: Array[Any] = a1
                            ^
error: type mismatch;
  found : Array[String]
  required: Array[Any]
Note: String <: Any, but class Array is invariant in type T.
```

В данном случае получилось, что Scala считает массивы нонвариантными (жесткими), следовательно, `Array[String]` не считается соответствующим `Array[Any]`. Но иногда необходимо организовать взаимодействие между имеющимися в Java устаревшими методами, которые используют в качестве средства эмуляции обобщенного массива `Object`-массив. Например, может возникнуть потребность вызвать метод сортировки наподобие того, который был рассмотрен ранее в отношении `String`-массива, передаваемого в качестве аргумента. Чтобы допустить такую возможность, Scala позволяет выполнять приведение массива из элементов типа `T` к массиву элементов любого из супертипов `T`:

```
scala> val a2: Array[Object] =
         a1.asInstanceOf[Array[Object]]
a2: Array[Object] = Array(abc)
```

Приведение типов во время компиляции не вызывает никаких возражений и всегда будет успешно работать во время выполнения программы, поскольку

положенная в основу работы JVM-модель времени выполнения рассматривает массивы в качестве ковариантных точно так же, как это делается в языке Java. Но впоследствии, как и на Java, может быть получено исключение `ArrayStore`.

19.4. Проверка аннотаций вариантности

Рассмотрев несколько примеров ненадежности вариантности, можно задать вопрос: какие именно определения классов следует отвергать, а какие — принимать? До сих пор во всех нарушениях целостности типов фигурировали переназначаемые поля или элементы массивов. В то же время чисто функциональная реализация очередей представляется хорошим кандидатом на ковариантность. Но в следующем примере показано, что ситуацию ненадежности можно подстроить даже при отсутствии переназначаемого поля.

Чтобы выстроить пример, предположим: очереди из показанного выше листинга 19.4 определены как ковариантные. Затем создадим подкласс очередей с указанным типом `Int` и переопределим метод `enqueue`:

```
class StrangeIntQueue extends Queue[Int] {
  override def enqueue(x: Int) = {
    println(math.sqrt(x))
    super.enqueue(x)
  }
}
```

Перед выполнением добавления метод `enqueue` в подклассе `StrangeIntQueue` выводит на стандартное устройство квадратный корень из своего (целочисленного) аргумента.

Теперь можно создать контрпример из двух строк кода:

```
val x: Queue[Any] = new StrangeIntQueue
x.enqueue("abc")
```

Первая из этих двух строк вполне допустима, поскольку `StrangeIntQueue` — подкласс `Queue[Int]`, и если предполагается ковариантность очередей, то `Queue[Int]` является подтипом `Queue[Any]`. Вполне допустима и вторая строка, так как `String`-значение можно добавлять в `Queue[Any]`. Но если взять их вместе, то у этих двух строк проявляется не имеющий никакого смысла эффект применения метода извлечения квадратного корня к строке.

Это явно не те простые изменяемые поля, которые делают ковариантные поля ненадежными. Проблема носит более общий характер. Получается, как только обобщенный параметр типа появляется в качестве типа параметра метода, содержащие данный метод класс или трейт в этом параметре типа могут не быть ковариантными.

Для очередей метод `enqueue` нарушает это условие:

```
class Queue[+T] {
  def enqueue(x: T) =
    ...
}
```

При запуске модифицированного класса очередей, подобного показанному ранее, в компиляторе Scala последний выдаст следующий результат:

```
Queues.scala:11: error: covariant type T occurs in
contravariant position in type T of value x
  def enqueue(x: T) =
              ^
```

Переназначаемые поля — частный случай правила, которое не позволяет параметрам типа, имеющим аннотацию `+`, использоваться в качестве типов параметра метода. Как упоминалось в разделе 18.2, переназначаемое поле `var x: T` рассматривается в Scala как геттер `def x: T` и как сеттер `def x_=(y: T)`. Как видите, сеттер имеет параметр поля типа `T`. Следовательно, этот тип не может быть ковариантным.

УСКОРЕННЫЙ РЕЖИМ ЧТЕНИЯ

> Далее в этом разделе мы рассмотрим механизм, с помощью которого компилятор Scala проверяет аннотацию вариантности. Если данные подробности вас пока не интересуют, то можете смело переходить к разделу 19.5. Следует усвоить главное: компилятор Scala будет проверять любую аннотацию вариантности, которую вы укажете в отношении параметров типа. Например, при попытке объявить ковариантный параметр типа (путем добавления знака `+`), способного вызвать потенциальные ошибки в ходе выполнения программы, программа откомпилирована не будет.

Чтобы проверить правильность аннотаций вариантности, компилятор Scala классифицирует все позиции в теле класса или трейта как *положительные*, *отрицательные* или *нейтральные*. Под позицией понимается любое место в теле класса или трейта (далее в тексте будет фигурировать только термин «класс»), где может использоваться параметр типа. Например, позицией является каждый параметр значения в методе, поскольку у параметра значения метода есть тип. Следовательно, в этой позиции может применяться параметр типа.

Компилятор проверяет каждое использование каждого из имеющихся в классе параметров типа. Параметры типа, для аннотации которых применяется знак `+`, могут быть задействованы только в положительных позициях, а параметры, для аннотации которых используется знак `-`, могут применяться лишь в отрицательных. Параметр типа, не имеющий аннотации вариантности, может использоваться в любой позиции. Таким образом, он является единственной разновидностью параметров типа, которая применима в нейтральных позициях тела класса.

В целях классификации позиций компилятор начинает работу с объявления параметра типа, а затем идет через более глубокие уровни вложенности. Позиции на верхнем уровне объявляемого класса классифицируются как положительные. По умолчанию позиции на более глубоких уровнях вложенности классифицируются таким же образом, как и охватывающие их уровни, но есть небольшое количество исключений, в которых классификация меняется. Позиции параметров значений метода классифицируются по *перевернутой* схеме относительно позиций за пределами метода, там положительная классификация становится

отрицательной, отрицательная — положительной, а нейтральная классификация так и остается нейтральной.

Текущая классификация действует по перевернутой схеме в отношении не только позиций параметров значений методов, но и параметров типа методов. В зависимости от вариантности соответствующего параметра типа классификация иногда бывает перевернута в имеющейся в типе позиции аргумента типа, например, это касается `Arg` в `C[Arg]`. Если параметр типа у `C` аннотирован с помощью знака `+`, то классификация остается такой же. Если с помощью знака `-`, то классификация определяется по перевернутой схеме. При отсутствии у параметра типа у `C` аннотации вариантности текущая классификация изменяется на нейтральную.

В качестве слегка надуманного примера рассмотрим следующее определение класса, в котором несколько позиций аннотированы согласно их классификации с помощью обозначений `^+` (для положительных) или `^-` (для отрицательных):

```
abstract class Cat[-T, +U] {
  def meow[W^-](volume: T^-, listener: Cat[U^+, T^-]^-)
    : Cat[Cat[U^+, T^-]^-, U^+]^+
}
```

Позиции параметра типа `W` и двух параметров значений, `volume` и `listener`, помечены как отрицательные. Если посмотреть на результирующий тип метода `meow`, то позиция первого аргумента, `Cat[U, T]`, помечена как отрицательная, поскольку первый параметр типа у `Cat`, `T`, аннотирован с помощью знака `-`. Тип `U` внутри этого аргумента опять имеет положительную позицию (после двух перевертываний), а тип `T` внутри этого аргумента остается в отрицательной.

Из всего вышесказанного можно сделать вывод, что отследить позиции вариантностей очень трудно. Поэтому помощь компилятора Scala, проделывающего эту работу за вас, несомненно, приветствуется.

После вычисления классификации компилятор проверяет, используется ли каждый параметр типа только в позициях, которые имеют соответствующую классификацию. В данном случае `T` используется лишь в отрицательных позициях, а `U` — лишь в положительных. Следовательно, класс `Cat` типизирован корректно.

19.5. Нижние ограничители

Вернемся к классу `Queue`. Вы видели, что прежнее определение `Queue[T]`, показанное в листинге 19.4, не может быть превращено в ковариантное в отношении `T`, поскольку `T` фигурирует в качестве типа параметра метода `enqueue` и находится в отрицательной позиции.

К счастью, существует способ выйти из этого положения: можно обобщить `enqueue`, превратив данный метод в полиморфный (то есть предоставить самому методу `enqueue` параметр типа), и воспользоваться *нижним ограничителем* его параметра типа. Новая формулировка `Queue` с реализацией этой идеи показана в листинге 19.6.

Листинг 19.6. Параметр типа с нижним ограничителем

```
class Queue[+T] (private val leading: List[T],
    private val trailing: List[T] ) {
  def enqueue[U >: T](x: U) =
    new Queue[U](leading, x :: trailing) // ...
}
```

В новом определении `enqueue` дается параметр типа `U`, и с помощью синтаксиса `U >: T` тип `T` определяется как нижний ограничитель для `U`. В результате от типа `U` требуется, чтобы он был супертипом для `T`[1]. Теперь параметр для `enqueue` имеет тип `U`, а не `T`, а возвращаемое значение метода теперь не `Queue[T]`, а `Queue[U]`.

Предположим, есть класс `Fruit`, имеющий два подкласса: `Apple` и `Orange`. С новым определением класса `Queue` появилась возможность добавить `Orange` в `Queue[Apple]`. Результатом будет `Queue[Fruit]`.

В этом пересмотренном определении `enqueue` типы используются правильно. Интуитивно понятно, что если `T` — более конкретный тип, чем ожидалось (например, `Apple` вместо `Fruit`), то вызов `enqueue` все равно будет работать, поскольку `U` (`Fruit`) по-прежнему будет супертипом для `T` (`Apple`)[2].

Возможно, новое определение `enqueue` лучше старого, поскольку имеет более обобщенный характер. В отличие от старой версии новое определение позволяет добавлять в очередь с элементами типа `T` элементы произвольного супертипа `U`. Результат получается типа `Queue[U]`. Наряду с ковариантностью очереди это позволяет получить правильную разновидность гибкости для моделирования очередей из различных типов элементов вполне естественным образом.

Это показывает, что в совокупности аннотации вариантности и нижние ограничители — хорошо сыгранная команда. Они являются хорошим примером *разработки, управляемой типами*, где типы интерфейса управляют его детальным проектированием и ее реализацией. В случае с очередями высока вероятность того, что вы не стали бы продумывать улучшенную реализацию `enqueue` с нижним ограничителем. Но у вас могло созреть решение сделать очереди ковариантными, в случае чего компилятор указал бы для `enqueue` на ошибку вариантности. Исправление ошибки вариантности путем добавления нижнего ограничителя придает `enqueue` большую обобщенность, а очереди в целом делает более полезными.

Вдобавок это наблюдение — главная причина того, почему в Scala предпочитаема вариантность по месту объявления (declaration-site variance), а не вариантность по месту использования (use-site variance), встречающаяся в Java в подстановочных символах (wildcards). В случае вариантности по месту использования вы разрабатываете класс самостоятельно. А вот клиентам данного класса придется вставлять подстановочные символы, и если они сделают это

---

[1] Отношения супертипов и подтипов рефлексивны. Это значит, тип является одновременно супертипом и подтипом по отношению к себе. Даже притом, что T — нижняя граница для U, T все же можно передавать методу enqueue.

[2] С технической точки зрения произошедшее — переворот для нижних границ. Параметр типа U находится в отрицательной позиции (один переворот), а нижняя граница (>: T) — в положительной (два переворота).

неправильно, то применить некоторые важные методы экземпляра станет невозможно. Вариантность — дело непростое, пользователи зачастую понимают ее неправильно и избегают ее, полагая, что подстановочные символы и дженерики для них слишком сложны. При использовании вариантности по месту объявления ваши намерения выражаются для компилятора, который выполнит двойную проверку, чтобы убедиться, что метод, который вам нужно сделать доступным, будет действительно доступен.

19.6. Контравариантность

До сих пор во всех представленных в данной главе примерах встречалась либо ковариантность, либо нонвариантность. Но бывают такие обстоятельства, при которых вполне естественно выглядит и контравариантность. Рассмотрим, к примеру, трейт каналов вывода, показанный в листинге 19.7.

Листинг 19.7. Контравариантный канал вывода

```
trait OutputChannel[-T] {
  def write(x: T): Unit
}
```

Здесь трейт `OutputChannel` определен с контравариантностью, указанной для `T`. Следовательно, получается, что канал вывода для `AnyRef` является подтипом канала вывода для `String`. Хотя на интуитивном уровне это может показаться непонятным, в действительности здесь есть определенный смысл. Понять, почему так, можно, рассмотрев возможные действия с `OutputChannel[String]`. Единственная поддерживаемая операция — запись в него значения типа `String`. Аналогичная операция может быть выполнена также в отношении `OutputChannel[AnyRef]`. Следовательно, вполне безопасно будет вместо `OutputChannel[String]` подставить `OutputChannel[AnyRef]`. В отличие от этого подставить `OutputChannel[String]` туда, где требуется `OutputChannel[AnyRef]`, будет небезопасно. В конце концов, на `OutputChannel[AnyRef]` можно отправить любой объект, а `OutputChannel[String]` требует, чтобы все записываемые значения были строками.

Эти рассуждения указывают на общий принцип разработки систем типов: вполне безопасно предположить, что тип `T` — подтип типа `U`, если значение типа `T` можно подставить там, где требуется значение типа `U`. Это называется *принципом подстановки Лисков*. Он соблюдается, если `T` поддерживает те же операции, что и `U`, и все принадлежащие `T` операции требуют меньшего, а предоставляют большее, чем соответствующие операции в `U`. В случае с каналами вывода `OutputChannel[AnyRef]` может быть подтипом `OutputChannel[String]`, поскольку в обоих типах поддерживается одна и та же операция `write`, и она требует меньшего в `OutputChannel[AnyRef]`, чем в `OutputChannel[String]`. Меньшее означает следующее: от аргумента в первом случае требуется только, чтобы он был типа `AnyRef`, а вот во втором случае от него требуется, чтобы он был типа `String`.

Иногда в одном и том же типе смешиваются ковариантность и контравариантность. Известный пример — функциональные трейты Scala. Например, при

написании функционального типа `A => B` Scala разворачивает этот код, приводя его к виду `Function1[A, B]`. Определение `Function1` в стандартной библиотеке использует как ковариантность, так и контравариантность: в листинге 19.8 показано, что трейт `Function1` контравариантен в аргументе функции типа `S` и ковариантен в результирующем типе `T`. Принцип подстановки Лисков здесь не нарушается, поскольку аргументы — это то, что требуется, а вот результаты — то, что предоставляется.

Листинг 19.8. Ковариантность и контравариантность Function1

```
trait Function1[-S, +T] {
  def apply(x: S): T
}
```

Рассмотрим в качестве примера приложение, показанное в листинге 19.9. Здесь класс `Publication` содержит одно параметрическое поле `title` типа `String`. Класс `Book` расширяет `Publication` и пересылает свой строковый параметр `title` конструктору своего суперкласса. В объекте-одиночке `Library` определяются набор книг `books` и метод `printBookList`, получающий функцию `info`, у которой есть тип `Book => AnyRef`. Иными словами, типом единственного параметра `printBookList` является функция, которая получает один аргумент типа `Book` и возвращает значение типа `AnyRef`. В приложении `Customer` определяется метод `getTitle`, получающий в качестве единственного своего параметра значение типа `Publication` и возвращающий значение типа `String`, которое содержит название переданной публикации `Publication`.

Листинг 19.9. Демонстрация вариантности параметра типа функции

```
class Publication(val title: String)
class Book(title: String) extends Publication(title)

object Library {
  val books: Set[Book] =
    Set(
      new Book("Programming in Scala"),
      new Book("Walden")
    )
  def printBookList(info: Book => AnyRef) = {
    for (book <- books) println(info(book))
  }
}

object Customer extends App {
  def getTitle(p: Publication): String = p.title
  Library.printBookList(getTitle)
}
```

Теперь посмотрим на последнюю строку в объекте `Customer`. В ней вызывается принадлежащий `Library` метод `printBookList`, которому в инкапсулированном в значение функции виде передается `getTitle`:

```
Library.printBookList(getTitle)
```

Эта строка кода проходит проверку на соответствие типу даже притом, что `String`, результирующий тип выполнения функции, является подтипом `AnyRef`, типом результата параметра `info` метода `printBookList`. Данный код проходит компиляцию, поскольку результирующие типы функций объявлены ковариантными (`+T` в листинге 19.8). Если заглянуть в тело `printBookList`, то можно получить представление о том, почему в этом есть определенный смысл.

Метод `printBookList` последовательно перебирает элементы своего списка книг и вызывает переданную ему функцию в отношении каждой книги. Он передает `AnyRef`-результат, возвращенный `info`, методу `println`, который вызывает в отношении этого результата метод `toString` и выводит на стандартное устройство возвращенную им строку. Данный процесс будет работать со `String`-значениями, а также с любыми другими подклассами `AnyRef`, в чем, собственно, и заключается смысл ковариантности результирующих типов функций.

Теперь рассмотрим параметр типа той функции, которая была передана методу `printBookList`. Хотя тип параметра, принадлежащего функции `info`, объявлен как `Book`, функция `getTitle` при ее передаче в этот метод получает значение типа `Publication`, а этот тип является для `Book` *супертипом*. Все это работает, поскольку, хотя типом параметра метода `printBookList` является `Book`, телу метода `printBookList` будет разрешено только передать значение типа `Book` в функцию. А ввиду того, что параметром типа функции `getTitle` является `Publication`, телу этой функции будет лишь разрешено обращаться к его параметру `p`, относящемуся к элементам, объявленным в классе `Publication`. Любой метод, объявленный в классе `Publication`, доступен также в его подклассе `Book`, поэтому все должно работать, в чем, собственно, и заключается смысл контравариантности типов результатов функций. Графическое представление всего вышесказанного можно увидеть на рис. 19.1.

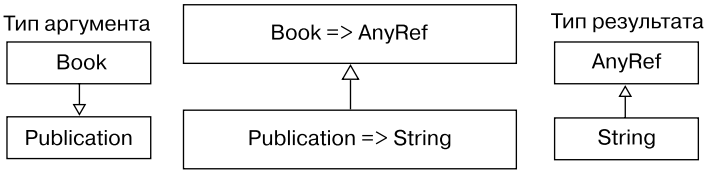


Рис. 19.1. Ковариантность и контравариантность в параметрах типа функции

Код в представленном выше листинге 19.9 проходит компиляцию, поскольку `Publication => String` является подтипом `Book => AnyRef`, что и показано в центре схемы. Результирующий тип `Function1` определен в качестве ковариантного, и потому показанное в правой части схемы отношение наследования двух результирующих типов имеет то же самое направление, что и две функции, показанные в центре. В отличие от этого, поскольку тип параметра функции `Function1` определен в качестве контравариантного, отношение наследования двух типов параметров, показанное в левой части схемы, имеет направление, обратное направлению отношения наследования двух функций.

19.7. Приватные данные объекта

У показанного ранее класса `Queue` есть проблема; она состоит в том, что операция `mirror` будет многократно копировать список `trailing` в список `leading`, если метод `head` вызывается несколько раз подряд в отношении списка, когда список `leading` пуст. Неэкономного копирования можно избежать, добавив некоторые рациональные побочные эффекты. В листинге 19.10 представлена новая реализация `Queue`, выполняющая не более одной коррекции списка `leading` за счет списка `trailing` для любой последовательности операций `head`.

Листинг 19.10. Оптимизированная функциональная очередь

```
class Queue[+T] private (
  private[this] var leading: List[T],
  private[this] var trailing: List[T]
) {

  private def mirror() =
    if (leading.isEmpty) {
      while (!trailing.isEmpty) {
        leading = trailing.head :: leading
        trailing = trailing.tail
      }
    }

  def head: T = {
    mirror()
    leading.head
  }

  def tail: Queue[T] = {
    mirror()
    new Queue(leading.tail, trailing)
  }

  def enqueue[U >: T](x: U) =
    new Queue[U](leading, x :: trailing)
}
```

В отличие от прежней версии, теперь `leading` и `trailing` являются переназначаемыми переменными, а `mirror` не возвращает новую очередь, а создает в отношении текущей очереди в качестве побочного эффекта перевернутую копию из `trailing` в `leading`. По отношению к реализации `Queue` этот побочный эффект носит чисто внутренний характер: поскольку `leading` и `trailing` — приватные переменные, клиентам `Queue` данный эффект не виден. Следовательно, согласно терминологии, установленной в главе 18, новая версия `Queue` все еще определяет чисто функциональные объекты, несмотря на то что теперь в них содержатся переназначаемые поля.

Прохождение этого кода через имеющийся в Scala механизм проверки типов может вызвать удивление. Все-таки у очередей теперь есть два переназначаемых поля ковариантного параметра типа `T`. Не нарушаются ли здесь правила вариант-

ности? Может быть, нарушение и существовало бы, если бы не маленькая деталь: у `leading` и `trailing` имеется модификатор `private[this]`, стало быть, они объявлены приватными.

Как упоминалось в разделе 13.5, к приватным элементам есть доступ только из того объекта, в котором они определены. Получается, что обращение к переменным из такого объекта не вызывает проблем с вариантностью. Интуитивно понятное объяснение состоит в том, что для создания обстоятельств, при которых вариантность вызовет ошибки типа, нужно иметь ссылку на объект, содержащий статически более слабый тип, чем тот, с которым был определен объект. Что же касается обращений к приватным значениям объекта, то создать подобные обстоятельства невозможно.

Для приватных определений объекта в используемых в Scala правилах проверки вариантности есть исключение. Когда параметр типа с аннотацией либо `+`, либо `–` находится только в тех позициях, которые имеют такую же классификацию вариантности, такие определения пропускаются. Следовательно, код, показанный выше в листинге 19.10, компилируется без ошибок. В то же время, если в двух модификаторах `private` пропустить квалификатор `[this]`, будут показаны две ошибки типа:

```
Queues.scala:1: error: covariant type T occurs in
contravariant position in type List[T] of parameter of
setter leading_=
class Queue[+T] private (private var leading: List[T],
                                      ^

Queues.scala:1: error: covariant type T occurs in
contravariant position in type List[T] of parameter of
setter trailing_=
                         private var trailing: List[T]) {
                                      ^
```

19.8. Верхние ограничители

В листинге 16.1 (см. выше) была показана предназначенная для списков функция сортировки слиянием, получавшая в качестве своего первого аргумента функцию сравнения, а в качестве второго, каррированного — сортируемый список. Еще один способ, который может вам пригодиться для организации подобной функции сортировки, заключается в требовании того, чтобы тип списка примешивал трейт `Ordered`. Как упоминалось в разделе 12.4, примешивание `Ordered` к классу и реализация в `Ordered` одного абстрактного метода, `compare`, позволит клиентам сравнивать экземпляры класса с помощью операторов `<`, `>`, `<=` и `>=`. В качестве примера в листинге 19.11 показан трейт `Ordered`, примешанный к классу `Person`.

В результате двух людей можно сравнивать так:

```
scala> val robert = new Person("Robert", "Jones")
robert: Person = Robert Jones
```

```
scala> val sally = new Person("Sally", "Smith")
sally: Person = Sally Smith

scala> robert < sally
res0: Boolean = true
```

Листинг 19.11. Класс Person, к которому примешан трейт Ordered

```
class Person(val firstName: String, val lastName: String)
    extends Ordered[Person] {

  def compare(that: Person) = {
    val lastNameComparison =
      lastName.compareToIgnoreCase(that.lastName)
    if (lastNameComparison != 0)
      lastNameComparison
    else
      firstName.compareToIgnoreCase(that.firstName)
  }

  override def toString = firstName + " " + lastName
}
```

Листинг 19.12. Функция сравнения с верхним ограничителем

```
def orderedMergeSort[T <: Ordered[T]](xs: List[T]): List[T] = {
  def merge(xs: List[T], ys: List[T]): List[T] =
    (xs, ys) match {
      case (Nil, _) => ys
      case (_, Nil) => xs
      case (x :: xs1, y :: ys1) =>
        if (x < y) x :: merge(xs1, ys)
        else y :: merge(xs, ys1)
    }
  val n = xs.length / 2
  if (n == 0) xs
  else {
    val (ys, zs) = xs splitAt n
    merge(orderedMergeSort(ys), orderedMergeSort(zs))
  }
}
```

Чтобы выставить требование о примешивании `Ordered` в тип списка, переданного вашей новой функции сортировки, следует задействовать *верхний ограничитель*. Он указывается так же, как нижний, за исключением того, что вместо обозначения `>:`, используемого для нижних ограничителей, применяется, как показано выше, в листинге 19.12, обозначение `<:`.

Используя синтаксис `T <: Ordered[T]`, вы показываете, что параметр типа `T` имеет верхний ограничитель `Ordered[T]`. Это значит, что тип элемента, переданного `orderedMergeSort`, должен быть подтипом `Ordered`. Следовательно, `List[Person]` можно передать `orderedMergeSort`, поскольку `Person` примешивает `Ordered`.

Рассмотрим, к примеру, следующий список:

```
scala> val people = List(
         new Person("Larry", "Wall"),
         new Person("Anders", "Hejlsberg"),
         new Person("Guido", "van Rossum"),
         new Person("Alan", "Kay"),
         new Person("Yukihiro", "Matsumoto")
       )
people: List[Person] = List(Larry Wall, Anders Hejlsberg,
  Guido van Rossum, Alan Kay, Yukihiro Matsumoto)
```

Поскольку тип элемента этого списка `Person` примешивает `Ordered[People]` (и поэтому является его подтипом), список можно передать методу `orderedMergeSort`:

```
scala> val sortedPeople = orderedMergeSort(people)
sortedPeople: List[Person] = List(Anders Hejlsberg, Alan Kay,
  Yukihiro Matsumoto, Guido van Rossum, Larry Wall)
```

А теперь следует заметить, что, хоть функция сортировки, показанная в том же листинге 19.12, и служит неплохой иллюстрацией верхних ограничителей, в действительности это не самый универсальный способ в Scala для разработки функции сортировки, получающей преимущества от трейта `Ordered`.

Так, функцию `orderedMergeSort` нельзя использовать для сортировки списка целых чисел, поскольку класс `Int` не является подтипом `Ordered[Int]`:

```
scala> val wontCompile = orderedMergeSort(List(3, 2, 1))
<console>:5: error: inferred type arguments [Int] do
   not conform to method orderedMergeSort's type
     parameter bounds [T <: Ordered[T]]
       val wontCompile = orderedMergeSort(List(3, 2, 1))
                         ^
```

В целях получения более универсального решения в разделе 21.6 будет показан порядок использования *неявных параметров* и *контекстных ограничителей*.

Резюме

В этой главе мы показали ряд техник, применяемых для скрытия информации: приватные конструкторы, фабричные методы, абстракцию типов и приватные члены объекта. Кроме этого, продемонстрировали способы указать вариантность типов данных и объяснили, что вариантность означает для реализации класса. И наконец, показали две техники, помогающие получить гибкие аннотации вариантности: нижние ограничители для параметров типов методов и аннотации `private[this]` для локальных полей и методов.

20 Абстрактные члены

Член класса или трейта называется *абстрактным*, если у него нет в классе полного определения. Реализовывать абстрактные элементы предполагается в подклассах того класса, в котором они объявлены. Воплощение этой идеи можно найти во многих объектно-ориентированных языках. Например, в Java можно объявить абстрактные методы. В Scala тоже, что было показано в разделе 10.2. Но Scala этим не ограничивается, и в нем данная идея реализуется самым универсальным образом: в качестве членов классов и трейтов можно объявлять не только методы, но и абстрактные поля и даже абстрактные типы.

В этой главе мы рассмотрим все четыре разновидности абстрактных членов: `val`- и `var`-переменные, методы и типы. Попутно изучим предварительно инициализированные поля, ленивые `val`-переменные, типы, зависящие от пути, и перечисления.

20.1. Краткий обзор абстрактных членов

В следующем трейте объявляется по одному абстрактному члену каждого вида: абстрактный тип (`T`), метод (`transform`), `val`-переменная (`initial`) и `var`-переменная (`current`):

```
trait Abstract {
  type T
  def transform(x: T): T
  val initial: T
  var current: T
}
```

Конкретная реализация трейта `Abstract` нуждается в заполнении определений для каждого из его абстрактных членов. Пример реализации, предоставляющий эти определения, выглядит так:

```
class Concrete extends Abstract {
  type T = String
  def transform(x: String) = x + x
  val initial = "hi"
  var current = initial
}
```

Реализация придает конкретное значение типу `T`, определяя его в качестве псевдонима типа `String`. Операция `transform` конкатенирует предоставленную ей строку с нею же самой, а для `initial`, как и для `current`, устанавливается значение `"hi"`.

Указанные примеры дают вам первое приблизительное представление о том, какие разновидности абстрактных членов существуют в Scala. Далее мы рассмотрим подробности, касающиеся этих членов, и объясним, для чего могут пригодиться эти новые формы абстрактных членов, а также члены-типы в целом.

20.2. Члены-типы

В примере, приведенном в предыдущем разделе, было показано, что понятие *«абстрактный тип»* в Scala означает объявление типа (с ключевым словом `type`) в качестве члена класса или трейта без указания определения. Абстрактными могут быть и сами классы, а трейты по определению абстрактные, однако ни один из них не является в Scala тем, что называют *абстрактным типом*. Абстрактный тип в Scala всегда выступает членом какого-либо класса или трейта, как тип `T` в трейте `Abstract`.

Неабстрактный (или конкретный) член-тип, такой как тип `T` в классе `Concrete`, можно представить себе в качестве способа определения нового имени или *псевдонима* для типа. К примеру, в классе `Concrete` типу `String` дается псевдоним `T`. В результате везде, где в определении класса `Concrete` появляется `T`, подразумевается `String`. Сюда включаются преобразования типов параметров и результирующих типов, как исходных, так и текущих, в которых при их объявлении в супертрейте `Abstract` упоминается `T`. Следовательно, когда в классе `Concrete` реализуются эти методы, такие обозначения `T` интерпретируются как `String`.

Один из поводов использовать член-тип — определение краткого описательного псевдонима для типа, чье имя длиннее или значение менее понятно, чем у псевдонима. Такие члены-типы могут сделать понятнее код класса или трейта. Другое основное применение членов-типов — объявление абстрактного типа, который должен быть определен в подклассе. Более подробно этот вариант использования, продемонстрированный в предыдущем разделе, мы рассмотрим чуть позже в данной главе.

20.3. Абстрактные val-переменные

Объявление абстрактной `val`-переменной выглядит следующим образом:

```
val initial: String
```

`Val`-переменной даются имя и тип, но не присваивается значение. Оно должно быть предоставлено конкретным определением `val`-переменной в подклассе. Например, в классе `Concrete` для реализации `val`-переменной используется такой код:

```
val initial = "hi"
```

Объявление в классе абстрактной `val`-переменной применяется, когда в этом классе еще неведомо нужное ей значение, но известно, что переменная в каждом экземпляре класса получит неизменяемое значение.

Объявление абстрактной `val`-переменной напоминает объявление абстрактного метода без параметров:

```
def initial: String
```

Клиентский код будет ссылаться как на `val`-переменную, так и на метод абсолютно одинаково (то есть `obj.initial`). Но если `initial` является абстрактной `val`-переменной, то клиенту гарантируется, что `obj.initial` будет при каждом обращении выдавать одно и то же значение. Если `initial` — абстрактный метод, то данная гарантия соблюдаться не будет, поскольку в таком случае метод `initial` можно реализовать с помощью конкретного метода, возвращающего при каждом своем вызове разные значения.

Иными словами, `val`-переменная ограничивает свою допустимую реализацию: любая реализация должна быть определением `val`-переменной — она не может быть `var`- или `def`-определением. А вот объявления абстрактных методов можно реализовать как конкретными определениями методов, так и конкретными определениями `var`-переменных. Если взять абстрактный класс `Fruit`, показанный в листинге 20.1, то класс `Apple` будет допустимой реализацией подкласса, а класс `BadApple` — нет.

Листинг 20.1. Переопределение абстрактных val-переменных и методов без параметров

```
abstract class Fruit {
  val v: String // 'v' – значение (value)
  def m: String // 'm' – метод (method)
}

abstract class Apple extends Fruit {
  val v: String
  val m: String // Нормально воспринимаемое переопределение 'def' в 'val'
}

abstract class BadApple extends Fruit {
  def v: String // ОШИБКА: переопределять 'val' в 'def' нельзя
  def m: String
}
```

20.4. Абстрактные var-переменные

Как и для абстрактной `val`-переменной, для абстрактной `var`-переменной объявляются только имя и тип, но не начальное значение. Например, в листинге 20.2 показан трейт `AbstractTime`, в котором объявляются две абстрактные переменные с именами `hour` и `minute`.

Листинг 20.2. Объявление абстрактных var-переменных

```
trait AbstractTime {
  var hour: Int
  var minute: Int
}
```

Что означают такие абстрактные `var`-переменные, как `hour` и `minute`? В разделе 18.2 было показано, что `var`-переменные, объявленные в качестве членов класса, оснащаются геттером и сеттером. Это справедливо и для абстрактных `var`-переменных. Если, к примеру, объявляется абстрактная `var`-переменная по имени `hour`, то подразумевается, что для нее объявляется абстрактный геттер `hour` и абстрактный сеттер `hour_=`. Тем самым не определяется никакое переназначаемое поле, а конкретная реализация абстрактной `var`-переменной будет выполнена в подклассах. Например, определение `AbstractTime`, показанное выше, в листинге 20.2, абсолютно эквивалентно определению, показанному в листинге 20.3.

Листинг 20.3. Расширение абстрактных var-переменных в геттеры и сеттеры

```
trait AbstractTime {
  def hour: Int                // get-метод для 'hour'
  def hour_=(x: Int): Unit     // set-метод для 'hour'
  def minute: Int              // get-метод для 'minute'
  def minute_=(x: Int) : Unit  // set-метод для 'minute'
}
```

20.5. Инициализация абстрактных val-переменных

Иногда абстрактные `val`-переменные играют роль, аналогичную роли параметров суперкласса: они позволяют предоставить в подклассе подробности, не указанные в суперклассе. На практике это обстоятельство играет важную роль для трейтов, поскольку у них нет конструкторов, которым можно передавать параметры. Таким образом, обычный принцип параметризации трейта работает через абстрактные `val`-переменные, реализованные в подклассах.

Рассмотрим в качестве примера переформулировку класса `Rational` из главы 6, который был показан в листинге 6.5, в трейт:

```
trait RationalTrait {
  val numerArg: Int
  val denomArg: Int
}
```

У класса `Rational` из главы 6 были два параметра: `n` для числителя рационального числа и `d` для его знаменателя. Представленный здесь трейт `RationalTrait`

определяет вместо них две абстрактные val-переменные: numerArg и denomArg. Чтобы создать конкретный экземпляр этого трейта, нужно реализовать определения абстрактных val-переменных, например:

```
new RationalTrait {
  val numerArg = 1
  val denomArg = 2
}
```

Здесь появляется ключевое слово new, после которого в фигурных скобках стоит тело трейта RationalTrait. Это выражение выдает экземпляр *анонимного класса*, примешивающего трейт и определяемого телом. Создание экземпляра данного анонимного класса дает эффект, аналогичный созданию экземпляра с помощью кода new Rational(1, 2).

Но аналогия здесь неполная. Есть небольшое различие, касающееся порядка, в котором инициализируются выражения. При записи следующего кода:

```
new Rational(expr1, expr2)
```

два выражения, expr1 и expr2, вычисляются перед инициализацией класса Rational, следовательно, значения expr1 и expr2 доступны для инициализации класса Rational.

С трейтами складывается обратная ситуация. При записи кода:

```
new RationalTrait {
  val numerArg = expr1
  val denomArg = expr2
}
```

выражения expr1 и expr2 вычисляются как часть инициализации анонимного класса, но анонимный класс инициализируется *после* RationalTrait. Следовательно, значения numerArg и denomArg в ходе инициализации RationalTrait недоступны (точнее говоря, выбор любого значения даст значение по умолчанию для типа Int, то есть нуль). Для представленного ранее определения RationalTrait это не проблема, поскольку при инициализации трейта значения numerArg или denomArg не используются. Но проблема возникает в варианте RationalTrait, показанном в листинге 20.4, где определяются нормализованные числитель и знаменатель.

Листинг 20.4. Трейт, использующий абстрактные val-переменные

```
trait RationalTrait {
  val numerArg: Int
  val denomArg: Int
  require(denomArg != 0)
  private val g = gcd(numerArg, denomArg)
  val numer = numerArg / g
  val denom = denomArg / g
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
  override def toString = s"$numer/$denom"
}
```

При попытке создать экземпляр этого трейта с какими-либо выражениями для числителя и знаменателя, не являющимися простыми литералами, выдается исключение:

```
scala> val x = 2
x: Int = 2

scala> new RationalTrait {
         val numerArg = 1 * x
         val denomArg = 2 * x
       }
java.lang.IllegalArgumentException: requirement failed
  at scala.Predef$.require(Predef.scala:280)
  at RationalTrait.$init$(<console>:4)
  ... 28 elided
```

Исключение в этом примере было сгенерировано потому, что при инициализации класса `RationalTrait` у `denomArg` сохранилось исходное нулевое значение, из-за чего вызов `require` завершился сбоем.

В данном примере показано, что порядок инициализации для параметров класса и абстрактных полей разный. Аргумент параметра класса вычисляется *до* его передачи конструктору класса (если только это не параметр, передаваемый по имени). А вот реализация определения `val`-переменной, которая находится в подклассе, вычисляется только *после* инициализации суперкласса.

Теперь вы понимаете, почему поведение абстрактных `val`-переменных отличается от поведения параметров, и было бы неплохо узнать, что с этим делать. Получится ли определить `RationalTrait`, который можно надежно инициализировать, не опасаясь, что возникнут ошибки из-за неинициализированных полей? В действительности в Scala предлагается два альтернативных решения этой проблемы: *предынициализируемые поля* и *ленивые `val`-переменные*. Эти решения рассматриваются в остальной части раздела.

Предынициализируемые поля

Первое решение — предынициализируемые поля — позволяет инициализировать поле подкласса до вызова суперкласса. Для этого нужно просто поместить определение поля в фигурные скобки перед вызовом конструктора класса. В качестве примера в листинге 20.5 показана еще одна попытка создания экземпляра `RationalTrait`. Из этого примера видно, что инициализирующая часть стоит перед упоминанием супертрейта `RationalTrait`. Они разделены ключевым словом `with`.

Листинг 20.5. Предынициализируемые поля в выражении анонимного класса

```
scala> new {
         val numerArg = 1 * x
         val denomArg = 2 * x
       } with RationalTrait
res1: RationalTrait = 1/2
```

Сфера применения предынициализируемых полей не ограничивается анонимными классами, они могут использоваться также в объектах или именованных подклассах. Соответствующие примеры показаны в листингах 20.6 и 20.7. Из них видно, что в каждом случае секция предварительной инициализации ставится после ключевого слова `extends`, относящегося к определяемому объекту или классу. Класс `RationalClass`, показанный в листинге 20.7, иллюстрирует общую схему доступности параметров класса для инициализации супертрейта.

Листинг 20.6. Предынициализируемые поля в определении объекта

```
object twoThirds extends {
  val numerArg = 2
  val denomArg = 3
} with RationalTrait
```

Листинг 20.7. Предынициализируемые поля в определении класса

```
class RationalClass(n: Int, d: Int) extends {
  val numerArg = n
  val denomArg = d
} with RationalTrait {
  def + (that: RationalClass) = new RationalClass(
    numer * that.denom + that.numer * denom,
    denom * that.denom
  )
}
```

Поскольку инициализация полей происходит до того, как вызывается конструктор суперкласса, их инициализаторы не могут ссылаться на создаваемый объект. Следовательно, если такой инициализатор ссылается на `this`, то происходит обращение к объекту, который содержит класс, или к уже созданному объекту, а не к создаваемому.

Рассмотрим пример:

```
scala> new {
       val numerArg = 1
       val denomArg = this.numerArg * 2
     } with RationalTrait
On line 3: error: value numerArg is not a member of object
$iw
               val denomArg = this.numerArg * 2
                                   ^
```

Данный пример не проходит компиляцию, поскольку ссылка `this.numerArg` нацеливалась на поле `numerArg` в объекте, содержащем `new` (в качестве которого в данном случае выступал синтезированный объект по имени `$iw`, в который интерпретатор помещает строки пользовательского ввода). Итак, еще раз: предынициализируемые поля ведут себя в этом смысле подобно аргументам конструктора класса.

Ленивые val-переменные

Предынициализируемые поля могут применяться для точной имитации поведения инициализации аргументов конструктора класса. Но иногда лучше позволить самой системе разобраться, как и что должно быть проинициализировано. Добиться этого можно с помощью определения *ленивой* `val`-переменной. Если перед определением `val`-переменной поставить модификатор `lazy`, то выражение инициализации справа будет вычисляться только при первом использовании `val`-переменной.

Определим, к примеру, объект `Demo` с `val`-переменной:

```
scala> object Demo {
         val x = { println("initializing x"); "done" }
       }
defined object Demo
```

Теперь сначала сошлемся на `Demo`, а затем на `Demo.x`:

```
scala> Demo
initializing x
res3: Demo.type = Demo$@2129a843

scala> Demo.x
res4: String = done
```

Как видите, на момент использования объекта `Demo` его поле `x` становится проинициализированным. Инициализация `x` составляет часть инициализации `Demo`. Но ситуация изменится, если определить поле `x` как `lazy`:

```
scala> object Demo {
         lazy val x = { println("initializing x"); "done" }
       }
defined object Demo

scala> Demo
res5: Demo.type = Demo$@1a18e68a

scala> Demo.x
initializing x
res6: String = done
```

Теперь инициализация `Demo` не включает инициализацию `x`. Она будет отложена до первого использования `x`. Это похоже на ситуацию определения `x` в качестве метода без параметров с помощью ключевого слова `def`. Но в отличие от `def` ленивая `val`-переменная никогда не вычисляется более одного раза. Фактически после первого вычисления ленивой `val`-переменной результат вычисления сохраняется, чтобы его можно было применить повторно при последующем использовании той же самой `val`-переменной.

При изучении данного примера создается впечатление, что объекты, подобные `Demo`, сами ведут себя как ленивые `val`-переменные, поскольку инициализируются

по необходимости при их первом использовании. Так и есть. Действительно, определение объекта может рассматриваться как сокращенная запись для определения ленивой `val`-переменной с анонимным классом, в котором описывается содержимое объекта.

Используя ленивые `val`-переменные, можно переделать `RationalTrait`, как показано в листинге 20.8. В новом определении трейта все конкретные поля определены как `lazy`. Есть еще одно изменение, касающееся предыдущего определения `RationalTrait`, показанного выше, в листинге 20.4. Данное изменение заключается в том, что условие `require` было перемещено из тела трейта в инициализатор приватного поля `g`, вычисляющий наибольший общий делитель для `numerArg` и `denomArg`. После внесения этих изменений при инициализации `LazyRationalTrait` делать больше ничего не нужно, поскольку весь код инициализации теперь является правосторонней частью ленивой `val`-переменной. Таким образом, теперь вполне безопасно инициализировать абстрактные поля `LazyRationalTrait` после того, как класс уже определен.

Листинг 20.8. Инициализация трейта с ленивыми val-переменными

```
trait LazyRationalTrait {
  val numerArg: Int
  val denomArg: Int
  lazy val numer = numerArg / g
  lazy val denom = denomArg / g
  override def toString = s"$numer/$denom"
  private lazy val g = {
    require(denomArg != 0)
    gcd(numerArg, denomArg)
  }
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

Рассмотрим пример:

```
scala> val x = 2
x: Int = 2

scala> new LazyRationalTrait {
         val numerArg = 1 * x
         val denomArg = 2 * x
       }
res7: LazyRationalTrait = 1/2
```

Какая-либо предварительная инициализация здесь не нужна. Проследим последовательность инициализации, приводящей к тому, что в показанном ранее коде на стандартное устройство будет выведена строка `1/2`.

1. Создается новый экземпляр `LazyRationalTrait`, и запускается код инициализации `LazyRationalTrait`. Этот код пуст — ни одно из полей `LazyRationalTrait` еще не проинициализировано.

2. С помощью вычисления выражения `new` определяется первичный конструктор анонимного подкласса. Данная процедура включает в себя инициализацию `numerArg` значением `2` и инициализацию `denomArg` значением `4`.
3. Далее интерпретатор в отношении создаваемого объекта вызывает метод `toString`, чтобы получившееся значение можно было бы вывести на стандартное устройство.
4. Метод `toString`, определенный в трейте `LazyRationalTrait`, выполняет первое обращение к полю `numer`, что вызывает вычисление инициализатора.
5. Инициализатор поля `numer` обращается к приватному полю `g`; таким образом, следующим вычисляется `g`. При этом вычислении происходит обращение к `numerArg` и `denomArg`, которые были определены на шаге 2.
6. Метод `toString` обращается к значению `denom`, что вызывает вычисление `denom`. При этом происходит обращение к значениям `denomArg` и `g`. Инициализатор поля `g` заново уже не вычисляется, поскольку был вычислен на шаге 5.
7. Создается и выводится строка результата `1/2`.

Обратите внимание: в классе `LazyRationalTrait` определение `g` появляется в тексте кода после определений в нем `numer` и `denom`. Несмотря на это, ввиду того что все три значения ленивые, `g` инициализируется до завершения инициализации `numer` и `denom`.

Тем самым демонстрируется важное свойство ленивых `val`-переменных: порядок следования их определений в тексте кода не играет никакой роли, поскольку инициализация значений выполняется по требованию. Стало быть, ленивые `val`-переменные могут освободить вас как программиста от необходимости обдумывать порядок расстановки определений `val`-переменных, чтобы гарантировать, что к моменту востребованности все будет определено.

Но данное преимущество сохраняется до тех пор, пока инициализация ленивых `val`-переменных не производит никаких побочных эффектов, а также не зависит от них. Если есть побочные эффекты, то порядок инициализации становится значимым. И тогда могут возникнуть серьезные трудности в отслеживании порядка запуска инициализационного кода, как было показано в предыдущем примере. Следовательно, ленивые `val`-переменные — идеальное дополнение к функциональным объектам, в которых порядок инициализации не имеет значения до тех пор, пока все в конечном счете не будет проинициализировано. А вот для преимущественно императивного кода эти переменные подходят меньше.

Ленивые функциональные языки

Scala — далеко не первый язык, использующий идеальную пару ленивых определений и функционального кода. Существует целая категория ленивых языков функционального программирования, в которых каждое значение и каждый параметр инициализируются лениво. Яркий представитель этого класса языков — Haskell.

20.6. Абстрактные типы

В начале этой главы в качестве объявления абстрактного типа мы показали код `type T`. Далее мы рассмотрим, что означает такое объявление абстрактного типа и для чего оно может пригодиться. Как и все остальные объявления абстракций, объявление абстрактного типа — заместитель для чего-либо, что будет конкретно определено в подклассах. В данном случае это тип, который будет определен ниже по иерархии классов. Следовательно, обозначение `T` ссылается на тип, который на момент его объявления еще неизвестен. Разные подклассы могут обеспечивать различные реализации `T`.

Рассмотрим широко известный пример, в который абстрактные типы вписываются вполне естественно. Предположим, что получена задача смоделировать привычный рацион животных. Начать можно с определения класса питания `Food` и класса животных `Animal` с методом питания `eat`:

```
class Food
abstract class Animal {
  def eat(food: Food): Unit
}
```

Затем можно попробовать создать специализацию этих двух классов в виде класса коров `Cow`, питающихся травой `Grass`:

```
class Grass extends Food
class Cow extends Animal {
  override def eat(food: Grass) = {} // Этот код не пройдет компиляцию
}
```

Но при попытке компиляции этих новых классов будут получены следующие ошибки:

```
BuggyAnimals.scala:7: error: class Cow needs to be
abstract, since method eat in class Animal of type
    (Food)Unit is not defined
class Cow extends Animal {
      ^
BuggyAnimals.scala:8: error: method eat overrides nothing
  override def eat(food: Grass) = {}
               ^
```

Дело в том, что метод `eat` в классе `Cow` не переопределяет метод `eat` класса `Animal`, поскольку типы их параметров различаются: в классе `Cow` это `Grass`, а в классе `Animal` это `Food`.

Некоторые считают, что в отклонении этих двух классов виновата слишком строгая система типов. Они говорят, что должно быть допустимо специализировать параметр метода в подклассе. Но если бы классы были позволены в том виде, в котором написаны, вы быстро оказались бы в весьма небезопасной ситуации.

К примеру, механизму проверки типов будет передан следующий скрипт:

```
class Food
abstract class Animal {
  def eat(food: Food): Unit
}
class Grass extends Food
class Cow extends Animal {
  override def eat(food: Grass) = {} // Этот код не пройдет компиляцию,
}                                    // но если бы это случилось...
class Fish extends Food
val bessy: Animal = new Cow
bessy eat (new Fish)     // ...коров можно было бы накормить рыбой.
```

Если снять ограничения, то программа пройдет компиляцию, поскольку коровы из класса `Cow` — животные из класса `Animal`, а у класса `Animal` есть метод кормления `eat`, который принимает любую разновидность питания `Food`, включая рыбу, то есть `Fish`. Но коровы не едят рыбу!

Вместо этого вам нужно применить более точное моделирование. Животные из класса `Animal` потребляют (`eat`) питание `Food`, но какое именно питание потребляет каждое животное, зависит от самого животного. Это довольно четко можно выразить с помощью абстрактного типа, что и показано в листинге 20.9.

Листинг 20.9. Моделирование подходящего питания с помощью абстрактных типов

```
class Food
abstract class Animal {
  type SuitableFood <: Food
  def eat(food: SuitableFood): Unit
}
```

С новым определением класса животное `Animal` может потреблять только то питание, которое ему подходит. Какое именно питание будет подходящим, нельзя определить на уровне класса `Animal`. Поэтому подходящее питание `SuitableFood` моделируется в виде абстрактного типа. У него есть верхний ограничитель `Food`, что выражено условием `<: Food`. Это значит, что любая конкретная реализация `SuitableFood` (в подклассе класса `Animal`) должна быть подклассом `Food`. К примеру, реализовать `SuitableFood` классом `IOException` не получится.

После определения `Animal` можно, как показано в листинге 20.10, перейти к коровам. Класс `Cow` устанавливает в качестве подходящего для коров питания `SuitableFood` траву `Grass`, а также определяет конкретный метод `eat` для данной разновидности питания.

Листинг 20.10. Реализация абстрактного типа в подклассе

```
class Grass extends Food
class Cow extends Animal {
  type SuitableFood = Grass
  override def eat(food: Grass) = {}
}
```

Эти новые определения класса компилируются без ошибок. При попытке запустить с новыми определениями класс контрпримера про коров, которые едят рыбу (cows that eat fish), будут получены следующие ошибки компиляции:

```
scala> class Fish extends Food
defined class Fish

scala> val bessy: Animal = new Cow
bessy: Animal = Cow@4df50829

scala> bessy eat (new Fish)
                  ^

       error: type mismatch;

        found   : Fish
        required: bessy.SuitableFood
```

20.7. Типы, зависящие от пути

Еще раз посмотрим на последнее сообщение об ошибке. Нас интересует тип, требующийся для метода `eat`: `bessy.SuitableFood`. Указание типа состоит из ссылки на объект, `bessy`, за которой следует поле типа объекта, `SuitableFood`. Тем самым показывается, что объекты в Scala в качестве членов могут иметь типы. Смысл `bessy.SuitableFood` — «тип `SuitableFood`, являющийся членом объекта, на который ссылается `bessy`, или, иначе, тип питания, подходящего для `bessy`.

Тип вида `bessy.SuitableFood` называется *типом, зависящим от пути* (path-dependent type). Слово «путь» здесь означает ссылку на объект. Это может быть единственное имя, такое как `bessy`, или более длинный путь доступа, такой как `farm.barn.bessy`, где все составляющие, `farm`, `barn` и `bessy`, — переменные (или имена объектов-одиночек), которые ссылаются на объекты.

Термин «тип, зависящий от пути» подразумевает, что тип зависит от пути; и в целом, различные пути дают начало разным типам. Например, предположим, что для определения классов собачьего питания `DogFood` и собак `Dog` используется следующий код:

```
class DogFood extends Food
class Dog extends Animal {
  type SuitableFood = DogFood
  override def eat(food: DogFood) = {}
}
```

При попытке накормить собаку едой для коров ваш код не пройдет компиляцию:

```
scala> val bessy = new Cow
bessy: Cow = Cow@6740b169
```

```
scala> val lassie = new Dog
lassie: Dog = Dog@31419d4a

scala> lassie eat (new bessy.SuitableFood)
                   ^
       error: type mismatch;

        found   : Grass
        required: DogFood
```

Проблема заключается в том, что типом объекта `SuitableFood`, переданного методу `eat`, выступает `bessy.SuitableFood`, а он несовместим с параметром типа `eat`, которым является `lassie.SuitableFood`.

В случае с двумя собаками из класса `Dog` ситуация другая. Поскольку в классе `Dog` тип `SuitableFood` определен в качестве псевдонима для класса `DogFood`, то типы `SuitableFood` двух представителей класса `Dog` по факту одинаковы. В результате экземпляр класса `Dog`, называемый `lassie`, фактически может питаться тем, что подходит другому экземпляру класса `Dog`, который мы назовем `bootsie`:

```
scala> val bootsie = new Dog
bootsie: Dog = Dog@2740c1e

scala> lassie eat (new bootsie.SuitableFood)
```

Тип, зависящий от пути, напоминает синтаксис для типа внутреннего класса в Java, но есть существенное различие: в типе, зависящем от пути, называется внешний *объект*, а в типе внутреннего класса — внешний *класс*. Типы внутренних классов в стиле Java могут быть выражены и в Scala, но записываются по-другому. Рассмотрим два класса, наружный `Outer` и внутренний `Inner`:

```
class Outer {
  class Inner
}
```

В Scala вместо применяемого в Java выражения `Outer.Inner` к внутреннему классу обращаются с помощью выражения `Outer#Inner`. Синтаксис с использованием точки (`.`) зарезервирован для объектов. Представим, к примеру, что создаются экземпляры двух объектов типа `Outer`:

```
val o1 = new Outer
val o2 = new Outer
```

Здесь `o1.Inner` и `o2.Inner` — два типа, зависящие от пути, и это разные типы. Оба они соответствуют более общему типу `Outer#Inner` (являются его подтипами), который представляет класс `Inner` с *произвольным* внешним объектом типа `Outer`. В отличие от этого тип `o1.Inner` ссылается на класс `Inner` с *конкретным* внешним объектом, на который ссылается `o1`. Точно так же тип `o2.Inner` ссылается на класс `Inner` с другим конкретным внешним объектом, на который ссылается `o2`.

В Scala, как и в Java, экземпляры внутреннего класса содержат ссылку на экземпляр охватывающего их внешнего класса. Это, к примеру, позволяет внутреннему классу обращаться к членам его внешнего класса. Таким образом, невозможно создать экземпляр внутреннего класса, не имея какого-либо способа указать экземпляр внешнего класса. Один из способов заключается в создании экземпляра внутреннего класса внутри тела внешнего класса. В подобном случае будет использован текущий экземпляр внешнего класса (в ссылке на который можно задействовать `this`).

Еще один способ заключается в использовании типа, зависящего от пути. Например, в типе `o1.Inner` присутствует название конкретного внешнего объекта, поэтому можно создать его экземпляр:

```
scala> new o1.Inner
res11: o1.Inner = Outer$Inner@2e13c72b
```

Получившийся внутренний объект будет содержать ссылку на свой внешний объект, то есть на объект, на который ссылается `o1`. В отличие от этого, поскольку тип `Outer#Inner` не содержит названия какого-либо конкретного экземпляра класса `Outer`, создать экземпляр данного класса невозможно:

```
scala> new Outer#Inner
                 ^
       error: Outer is not a legal prefix for
           a constructor
```

20.8. Уточняющие типы

Когда класс является наследником другого класса, первый класс называют *номинальным* подтипом другого класса. Этот подтип *номинальный*, поскольку у каждого типа есть *имя* и имена явным образом объявлены имеющими отношение подтипирования. Кроме того, в Scala дополнительно поддерживается *структурное* подтипирование, где отношение подтипирования возникает просто потому, что у двух типов есть совместимые элементы. Для получения в Scala структурного подтипирования нужно задействовать имеющиеся в данном языке *уточняющие типы*.

Обычно удобнее применять номинальное подтипирование, поэтому в любой новой конструкции нужно сначала попробовать воспользоваться именно им. Имя — один краткий идентификатор, следовательно, короче явного перечисления типов членов. Кроме того, структурное подтипирование зачастую более гибко, чем требуется. Тем не менее оно имеет свои преимущества. Одно из них заключается в том, что иногда действительно не нужно ничего определять в виде типов, кроме членов самого класса. Предположим, к примеру, что необходимо определить класс пастбища `Pasture`, который может содержать животных, поедающих траву. Одним из вариантов может быть определение трейта для животных, питающихся

травой, — AnimalThatEatsGrass и его примешивание в каждый класс, где он применяется. Но это будет слишком многословным решением. В классе Cow уже есть объявление, что это животное и оно ест траву, а теперь придется еще раз объявлять, что это и животное, поедающее траву.

Вместо определения AnimalThatEatsGrass можно воспользоваться уточняющим типом. Просто запишите основной тип, Animal, а за ним последовательность членов, перечисленных в фигурных скобках. Члены в фигурных скобках представляют дальнейшие указания, или, если хотите, уточнения, типов элементов из основного класса.

Вот как записывается тип «животное, поедающее траву»:

```
Animal { type SuitableFood = Grass }
```

Теперь, имея в своем распоряжении этот тип, класс пастбища можно записать следующим образом:

```
class Pasture {
  var animals: List[Animal { type SuitableFood = Grass }] = Nil
  // ...
}
```

20.9. Перечисления

Как интересный вариант применения типов, зависящих от пути, можно рассмотреть имеющуюся в Scala поддержку перечислений. В некоторых других языках, включая Java и C#, перечисления фигурируют в качестве встроенных в язык конструкций для определения новых типов. Scala не нуждается в специальном синтаксисе для перечислений. Вместо этого в стандартной библиотеке языка есть класс scala.Enumeration.

В целях создания нового перечисления определяется объект, расширяющий этот класс. В примере ниже показано определение нового перечисления под названием Colors:

```
object Color extends Enumeration {
  val Red = Value
  val Green = Value
  val Blue = Value
}
```

Scala позволяет также сократить ряд последовательных определений val- или var-переменных, у которых правая сторона выражений одинаковая. В качестве эквивалента показанного ранее кода можно сделать следующую запись:

```
object Color extends Enumeration {
  val Red, Green, Blue = Value
}
```

В этом определении объекта предоставляются три значения: `Color.Red`, `Color.Green` и `Color.Blue`. Кроме того, все, что имеется в `Color`, можно импортировать с помощью такого кода:

```
import Color._
```

а затем просто использовать имена `Red`, `Green` и `Blue`. Но каким будет тип этих значений?

Класс `Enumeration` определяет внутренний класс по имени `Value`, а метод без параметров с таким же именем `Value` возвращает новый экземпляр этого класса. Иными словами, типом такого значения, как `Color.Red`, выступает `Color.Value`, который является типом всех перечисляемых значений, определенных в объекте `Color`. Это тип, зависящий от пути, где `Color` — путь, а `Value` — зависящий от него тип. Важно, что это совершенно новый тип, отличающийся от других типов.

В частности, если определить еще одно перечисление, такое как это:

```
object Direction extends Enumeration {
  val North, East, South, West = Value
}
```

то `Direction.Value` будет отличаться от `Color.Value`, поскольку та часть, которая относится к пути, у этих двух типов разная.

Имеющийся в Scala класс `Enumeration` предлагает также многие другие свойства, которые можно найти в конструкциях перечислений других языков. Со значениями перечислений можно связать имена, воспользовавшись для этого другим перегруженным вариантом метода `Value`:

```
object Direction extends Enumeration {
  val North = Value("North")
  val East = Value("East")
  val South = Value("South")
  val West = Value("West")
}
```

Значения в перечислении можно перебрать через множество, которое возвращает имеющийся в перечислении метод `values`:

```
scala> for (d <- Direction.values) print(d + " ")
North East South West
```

Значения в перечислении нумеруются с нуля, а номер значения в перечислении можно определить с помощью имеющегося в нем метода `id`:

```
scala> Direction.East.id
res14: Int = 1
```

Можно пойти и другим путем — от неотрицательного целого числа, которое служит этому номеру идентификатором в перечислении:

```
scala> Direction(1)
res15: Direction.Value = East
```

Этого должно быть вполне достаточно, чтобы приступить к работе с перечислениями. Дополнительные сведения можно получить в комментариях Scaladoc для класса `scala.Enumeration`.

20.10. Практический пример: работа с валютой

Далее в главе рассмотрен практический пример, объясняющий порядок использования в Scala абстрактных типов. При этом будет поставлена задача разработать класс `Currency`. Обычный его экземпляр будет представлять денежную сумму в долларах, евро, йенах и некоторых других валютах. Он позволит совершать с валютой ряд арифметических операций. Например, можно будет сложить две суммы в одной и той же валюте. Или умножить текущую сумму на коэффициент процентной ставки.

Эти соображения приводят к следующей первой конструкции класса валют:

```
// Первая (нерабочая) конструкция класса Currency
abstract class Currency {
  val amount: Long
  def designation: String
  override def toString = s"$amount $designation"
  def + (that: Currency): Currency = ...
  def * (x: Double): Currency = ...
}
```

Поле `amount` (сумма) в классе валют — количество представляемых ею валютных единиц. Оно имеет тип `Long`, то есть представляемая сумма денежных средств может быть очень крупной, сравнимой с рыночной капитализацией Google или Apple. Здесь поле оставлено абстрактным в ожидании своего определения, когда в подклассе зайдет речь о конкретной сумме. Наименование валюты `designation` — строка, которая идентифицирует эту валюту. Метод `toString` класса `Currency` показывает сумму и наименование валюты. Он будет выдавать результат следующего вида:

```
79 USD
11000 Yen
99 Euro
```

И наконец, имеются методы `+` для сложения сумм в валюте и `*` для умножения суммы в валюте на число с плавающей точкой. Конкретное значение в валюте можно создать, предоставив конкретные значения суммы и наименования валюты:

```
new Currency {
  val amount = 79L
  def designation = "USD"
}
```

Эта конструкция не вызовет нареканий, если задумано моделирование с использованием лишь одной валюты, например только долларов или только евро. Но она не будет работать при необходимости иметь дело сразу с несколькими валютами. Предположим, выполняется моделирование долларов и евро в качестве двух подклассов класса валюты:

```
abstract class Dollar extends Currency {
  def designation = "USD"
}
abstract class Euro extends Currency {
  def designation = "Euro"
}
```

На первый взгляд все выглядит вполне разумно. Но данный код позволит складывать доллары с евро. Результатом такого сложения будет тип `Currency`. Но это будет весьма забавная валюта — смесь евро и долларов. Вместо этого нужно получить более специализированную версию метода `+`. При его реализации в классе `Dollar` он должен получать аргументы типа `Dollar` и выдавать результат типа `Dollar`; при реализации в классе `Euro` — получать аргументы типа `Euro` и выдавать результат типа `Euro`. Следовательно, тип метода сложения будет изменяться в зависимости от того, в каком классе находится. И все же хотелось бы создать метод сложения единожды, а не делать это при каждом новом определении валюты.

Чтобы помочь справиться с подобными ситуациями, Scala предоставляет весьма простую технологию. Если к моменту определения класса что-то еще неизвестно, то нужно сделать это «что-то» абстрактным. Технология применима как к значениям, так и к типам. В случае с валютами точные типы аргументов и результирующие типы метода сложения неизвестны, следовательно, являются подходящими кандидатами для того, чтобы стать абстрактными.

Это привело бы к следующей предварительной версии кода класса `AbstractCurrency`:

```
// Вторая (все еще несовершенная) конструкция класса Currency
abstract class AbstractCurrency {
  type Currency <: AbstractCurrency
  val amount: Long
  def designation: String
  override def toString = s"$amount $designation"
  def + (that: Currency): Currency = ...
  def * (x: Double): Currency = ...
}
```

Единственное отличие от прежней ситуации заключается в том, что класс теперь называется `AbstractCurrency` и содержит абстрактный тип `Currency`, представляющий стоящую под вопросом реальную валюту. Каждому конкретному подклассу `AbstractCurrency` придется фиксировать тип `Currency`, чтобы обозначать конкретный подкласс как таковой, тем самым затягивая узел.

Вот как, к примеру, выглядит новая версия класса `Dollar`, которая теперь расширяет класс `AbstractCurrency`:

```
abstract class Dollar extends AbstractCurrency {
  type Currency = Dollar
  def designation = "USD"
}
```

Данная конструкция вполне работоспособна, но по-прежнему далека от совершенства. Есть проблема, скрывающаяся за многоточиями, которые показывают в классе `AbstractCurrency` пропущенные определения методов `+` и `*`. В частности, как в этом классе должен быть реализован метод сложения? Нетрудно вычислить правильную сумму в новой валюте как `this.amount + that.amount`, но как преобразовать сумму в валюту нужного типа?

Можно попробовать применить следующий код:

```
def + (that: Currency): Currency = new Currency {
  val amount = this.amount + that.amount
}
```

Но он не пройдет компиляцию:

```
error: class type required
  def + (that: Currency): Currency = new Currency {
                                     ^
```

Одно из ограничений в трактовке в Scala абстрактных типов заключается в невозможности создать экземпляр абстрактного типа, а также в невозможности абстрактного типа играть роль супертипа для другого класса[1]. Следовательно, компилятор будет отвергать код показанного здесь примера, в котором предпринимается попытка создать экземпляр `Currency`.

Но эти ограничения можно обойти, используя *фабричный метод*. Вместо того чтобы создавать экземпляр абстрактного класса, напрямую объявите абстрактный метод, который делает это. Затем там, где абстрактный тип устанавливается в какой-либо конкретный тип, нужно предоставить конкретную реализацию фабричного метода. Для класса `AbstractCurrency` все вышесказанное будет выглядеть так:

```
abstract class AbstractCurrency {
  type Currency <: AbstractCurrency // абстрактный тип
  def make(amount: Long): Currency  // фабричный метод
  ...                       // вся остальная часть определения класса
}
```

Подобную конструкцию, конечно, можно заставить работать, но выглядит она как-то подозрительно. Зачем помещать фабричный метод *внутрь* класса `AbstractCurrency`? Это выглядит довольно сомнительно как минимум по двум причинам. Во-первых, если есть некая сумма в валюте (скажем, один доллар), то есть и возможность нарастить сумму в той же валюте, используя следующий код:

```
myDollar.make(100) // здесь еще сто!
```

[1] В последнее время появились многообещающие исследования виртуальных классов, которые позволили бы сделать это, но пока такие классы в Scala не поддерживаются.

В эпоху цветных ксероксов данный скрипт может стать заманчивым, но следует надеяться, что никто не сможет проделывать это слишком долго, не будучи пойманным за руку. Во-вторых, этот код, если у вас уже есть ссылка на объект `Currency`, позволяет создавать дополнительные объекты `Currency`. Но как получить первый объект данной валюты `Currency`? Вам понадобится другой метод создания, выполняющий практически ту же работу, что и `make`. То есть вы столкнулись со случаем дублирования кода, являющимся верным признаком «кода с душком».

Решение, конечно же, будет заключаться в перемещении абстрактного типа и фабричного класса за пределы класса `AbstractCurrency`. Нужно создать другой класс, содержащий класс `AbstractCurrency`, тип `Currency` и фабричный метод `make`.

Назовем этот класс `CurrencyZone`:

```
abstract class CurrencyZone {
  type Currency <: AbstractCurrency
  def make(x: Long): Currency
  abstract class AbstractCurrency {
    val amount: Long
    def designation: String
    override def toString = s"$amount $designation"
    def + (that: Currency): Currency =
      make(this.amount + that.amount)
    def * (x: Double): Currency =
      make((this.amount * x).toLong)
  }
}
```

Пример конкретизации `CurrencyZone` — объект `US`, который можно определить следующим образом:

```
object US extends CurrencyZone {
  abstract class Dollar extends AbstractCurrency {
    def designation = "USD"
  }
  type Currency = Dollar
  def make(x: Long) = new Dollar { val amount = x }
}
```

Здесь `US` — объект, расширяющий `CurrencyZone`. В нем определяется класс `Dollar`, являющийся подклассом `AbstractCurrency`. Следовательно, тип денежных единиц в этой зоне — доллар США, `US.Dollar`. Объект `US` также устанавливает, что тип `Currency` будет псевдонимом для `Dollar`, и предоставляет реализацию фабричного метода `make` для возвращения суммы в долларах.

Конструкция вполне работоспособна. Нужно лишь добавить несколько уточнений. Первое из них касается разменных монет. До сих пор каждая валюта измерялась в целых единицах: долларах, евро или йенах. Но у большинства валют имеются разменные монеты, например, в США есть доллары и центы. Наиболее простой способ моделировать центы — использовать поле `amount` в `US.Currency`, представленное в центах, а не в долларах. Чтобы вернуться к доллару, будет полез-

но ввести в класс CurrencyZone поле CurrencyUnit, содержащее одну стандартную единицу в данной валюте:

```
abstract class CurrencyZone {
  ...
  val CurrencyUnit: Currency
}
```

Как показано в листинге 20.11, в объекте US могут быть определены величины Cent, Dollar и CurrencyUnit. Это определение объекта похоже на предыдущее, за исключением того, что в него добавлены три новых поля. Поле Cent представляет сумму 1 US.Currency. Это объект, аналогичный одноцентовой монете. Поле Dollar представляет сумму 100 US.Currency. Следовательно, объект US теперь определяет имя Dollar двумя способами. *Тип* Dollar, определенный абстрактным внутренним классом по имени Dollar, представляет общее название валюты Currency, действительное в валютной зоне US. В отличие от этого, *значение* Dollar, на которое ссылается val-поле по имени Dollar, представляет 1 доллар США, аналогичный однодолларовой купюре. Третье определение поля CurrencyUnit указывает на то, что стандартная денежная единица в зоне США — доллар, Dollar, то есть значение Dollar, на которое ссылается поле, не является типом Dollar.

Листинг 20.11. Зона валюты США

```
object US extends CurrencyZone {
  abstract class Dollar extends AbstractCurrency {
    def designation = "USD"
  }
  type Currency = Dollar
  def make(cents: Long) = new Dollar {
    val amount = cents
  }
  val Cent = make(1)
  val Dollar = make(100)
  val CurrencyUnit = Dollar
}
```

Метод toString в классе Currency также нуждается в адаптации для восприятия разменных монет на счету. Например, сумма 10 долларов 23 цента должна выводиться как десятичное число: 10.23 USD. Чтобы добиться этого результата, принадлежащий Currency метод toString можно реализовать следующим образом:

```
override def toString =
  ((amount.toDouble / CurrencyUnit.amount.toDouble)
   formatted ("%." + decimals(CurrencyUnit.amount) + "f")
   + " " + designation)
```

Здесь formatted является методом, доступным в Scala в нескольких классах, включая Double[1]. Метод formatted возвращает строку, полученную в результате

[1] Чтобы обеспечить доступность метода formatted, в Scala используются обогащающие оболочки, рассмотренные в разделе 5.10.

форматирования исходной строки, в отношении которой он был вызван, в соответствии со строкой форматирования, переданной ему в виде его правого операнда. Синтаксис строк форматирования, передаваемых методу `formatted`, аналогичен синтаксису, используемому для Java-метода `String.format`.

Например, строка форматирования `%.2f` форматирует число с двумя знаками после точки. Строка форматирования, примененная в показанном ранее методе `toString`, собирается путем вызова метода `decimals` в отношении `CurrencyUnit.amount`. Данный метод возвращает число десятичных знаков десятичной степени за вычетом единицы. Например, `decimals(10)` — это 1, `decimals(100)` — это 2 и т. д. Метод `decimals` реализован в виде простой рекурсии:

```
private def decimals(n: Long): Int =
  if (n == 1) 0 else 1 + decimals(n / 10)
```

В листинге 20.12 показаны некоторые другие валютные зоны. В качестве еще одного уточнения к модели можно добавить свойство обмена валют. Сначала, как показано в листинге 20.13, можно создать объект `Converter`, содержащий применяемые обменные курсы валют. Затем к классу `Currency` можно добавить метод обмена, `from`, который выполняет конвертацию из заданной исходной валюты в текущий объект `Currency`:

```
def from(other: CurrencyZone#AbstractCurrency): Currency =
  make(math.round(
    other.amount.toDouble * Converter.exchangeRate
      (other.designation)(this.designation)))
```

Листинг 20.12. Валютные зоны для Европы и Японии

```
object Europe extends CurrencyZone {
  abstract class Euro extends AbstractCurrency {
    def designation = "EUR"
  }
  type Currency = Euro
  def make(cents: Long) = new Euro {
    val amount = cents
  }
  val Cent = make(1)
  val Euro = make(100)
  val CurrencyUnit = Euro
}

object Japan extends CurrencyZone {
  abstract class Yen extends AbstractCurrency {
    def designation = "JPY"
  }
  type Currency = Yen
  def make(yen: Long) = new Yen {
    val amount = yen
  }
  val Yen = make(1)
  val CurrencyUnit = Yen
}
```

Листинг 20.13. Объект converter с отображением курсов обмена

```
object Converter {
  var exchangeRate = Map(
    "USD" -> Map("USD" -> 1.0 , "EUR" -> 0.7596,
                 "JPY" -> 1.211 , "CHF" -> 1.223),
    "EUR" -> Map("USD" -> 1.316 , "EUR" -> 1.0 ,
                 "JPY" -> 1.594 , "CHF" -> 1.623),
    "JPY" -> Map("USD" -> 0.8257, "EUR" -> 0.6272,
                 "JPY" -> 1.0 , "CHF" -> 1.018),
    "CHF" -> Map("USD" -> 0.8108, "EUR" -> 0.6160,
                 "JPY" -> 0.982 , "CHF" -> 1.0 )
  )
}
```

Метод `from` получает в качестве аргумента произвольную валюту. Это выражено его формальным типом параметра `CurrencyZone#AbstractCurrency`, который показывает, что переданный как `other` аргумент должен быть типа `AbstractCurrency` в некоторой произвольной и неизвестной валютной зоне `CurrencyZone`. Результат метода — перемножение суммы в другой валюте с курсом обмена между другой и текущей валютами[1].

Финальная версия класса `CurrencyZone` показана в листинге 20.14. Класс можно протестировать в командной оболочке Scala. Предполагается, что класс `CurrencyZone` и все конкретные объекты `CurrencyZone` определены в пакете `org.stairwaybook.currencies`. Сперва нужно импортировать в командную оболочку `org.stairwaybook.currencies._`. Затем можно будет выполнить ряд обменных операций с валютой:

```
scala> Japan.Yen from US.Dollar * 100
res16: Japan.Currency = 12110 JPY

scala> Europe.Euro from res16
res17: Europe.Currency = 75.95 EUR

scala> US.Dollar from res17
res18: US.Currency = 99.95 USD
```

Листинг 20.14. Полный код класса CurrencyZone

```
abstract class CurrencyZone {

  type Currency <: AbstractCurrency
  def make(x: Long): Currency

  abstract class AbstractCurrency {

    val amount: Long
    def designation: String
```

[1] Кстати, если вы полагаете, что сделка по японской йене будет неудачной, то курсы обмена валют основаны на числовых показателях в их CurrencyZone. Таким образом, 1.211 — курс обмена центов США на японскую йену.

```
    def + (that: Currency): Currency =
      make(this.amount + that.amount)
    def * (x: Double): Currency =
      make((this.amount * x).toLong)
    def - (that: Currency): Currency =
      make(this.amount - that.amount)
    def / (that: Double) =
      make((this.amount / that).toLong)
    def / (that: Currency) =
      this.amount.toDouble / that.amount

    def from(other: CurrencyZone#AbstractCurrency): Currency =
      make(math.round(
        other.amount.toDouble * Converter.exchangeRate
          (other.designation)(this.designation)))

    private def decimals(n: Long): Int =
      if (n == 1) 0 else 1 + decimals(n / 10)

    override def toString =
      ((amount.toDouble / CurrencyUnit.amount.toDouble)
       formatted ("%." + decimals(CurrencyUnit.amount) + "f")
       + " " + designation)
  }

  val CurrencyUnit: Currency
}
```

Из факта получения почти такого же значения после трех конвертаций следует, что у нас весьма выгодные курсы обмена! Кроме того, можно нарастить значение в некоторой валюте:

```
scala> US.Dollar * 100 + res18
res19: US.Currency = 199.95 USD
```

В то же время складывать суммы разных валют нельзя:

```
scala> US.Dollar + Europe.Euro
                   ^
       error: type mismatch;
        found   : Europe.Euro
        required: US.Currency
           (which expands to) US.Dollar
```

Абстракция типов выполняет свою работу, не позволяя складывать два значения в разных единицах измерения (в данном случае валютах). Она мешает нам выполнять необоснованные вычисления. Неверные преобразования между различными единицами могут показаться тривиальными недочетами, но способны привести к весьма серьезным системным сбоям. Например, к аварии спутника Mars Climate Orbiter 23 сентября 1999 года, вызванной тем, что одна команда инженеров использовала метрическую систему мер, а другая — систему мер, принятую в Великобритании. Если бы единицы измерения были запрограммированы

так же, как сделано с валютой в текущей главе, то данная ошибка была бы выявлена во время простого запуска кода на компиляцию. Вместо этого она стала причиной аварии космического аппарата после почти десятимесячного полета.

Резюме

В Scala предлагается рационально структурированная и самая общая поддержка объектно-ориентированной абстракции. При этом допускается применение не только абстрактных методов, но и значений, переменных и типов. В данной главе мы показали способы извлечь преимущества из использования абстрактных членов класса. С их помощью реализуется простой, но весьма эффективный принцип структурирования систем: все неизвестное при разработке класса нужно превращать в абстрактные члены. Тогда система типов задаст направление развитию вашей модели точно так же, как вы увидели в примере с валютой. И неважно, что именно будет неизвестно: тип, метод, переменная или значение. Все это в Scala можно объявить абстрактным.

21 Неявные преобразования и параметры

Между вашим кодом и библиотеками других разработчиков существует принципиальная разница: свой код при желании можно изменить или расширить, но если нужно воспользоваться чьими-либо библиотеками, то зачастую приходится принимать их такими, какие они есть. Чтобы облегчить решение этой проблемы, в языках программирования появился ряд конструкций. В Ruby есть модули, а Smalltalk позволяет пакетам добавлять в классы друг друга. Конечно, это очень эффективно, но и довольно опасно, поскольку дает возможность изменять поведение класса для всего приложения, о тех или иных частях которого вы можете ничего не знать. В C# 3.0 имеются статически расширяемые методы, имеющие более локальную, но и более ограниченную природу, позволяющую добавить к классу только методы, но не поля и не позволяющую реализовывать в классе новые интерфейсы.

В Scala в ответ на это были реализованы неявные преобразования и параметры. Они позволяют сделать существующие библиотеки гораздо более приятными для работы, давая возможность не указывать скучные, очевидные подробности, затмевающие интересные части вашего кода. При разумном использовании этого свойства получается код, сфокусированный на оригинальных, нетривиальных частях вашей программы. В этой главе мы покажем, как работают неявные преобразования и параметры, и рассмотрим некоторые наиболее распространенные способы их применения.

21.1. Неявные преобразования

Прежде чем углубиться в подробности неявных преобразований, рассмотрим типичный пример их применения. Они зачастую полезны при работе с двумя большими частями программного средства, которые разрабатывались без учета друг друга. В каждой библиотеке имеются собственные способы программирования некоего замысла, являющиеся, по сути, одним и тем же. Неявные преобразования

помогают сократить количество необходимых явных преобразований одного типа в другой.

В целях реализации кросс-платформенных пользовательских интерфейсов в Java включена библиотека под названием Swing. Одна из выполняемых ею задач заключается в обработке событий от операционной системы, преобразовании их в независимые от используемой платформы объекты событий и передаче последних тем частям приложения, которые называются *слушателями событий* (event listeners).

Если бы Swing создавалась с прицелом на Scala, то слушатели событий, вероятно, были бы представлены функциональным типом. Тогда вызывающие объекты могли бы использовать синтаксис функционального литерала в качестве облегченного способа, призванного указать то, что должно произойти для определенного класса событий. Поскольку в Java отсутствуют функциональные литералы, то в Swing применяется следующая наилучшая альтернатива — внутренний класс, который реализует интерфейс, состоящий из одного метода. В случае со слушателями действий (action listeners) интерфейс называется `ActionListener`.

Если программа на Scala, которая применяет Swing, не применяет неявное преобразование, то должна, подобно программе на Java, воспользоваться внутренними классами. Рассмотрим пример, создающий кнопку и «вешающий» на нее слушатель действия. Этот слушатель вызывается при нажатии кнопки, после чего выводится строка `"pressed!"`:

```
val button = new JButton
button.addActionListener(
  new ActionListener {
    def actionPerformed(event: ActionEvent) = {
      println("pressed!")
    }
  }
)
```

Представленный код очень невыразительный и неинформативный. То, что данный слушатель, — это `ActionListener`, метод обратного вызова называется `actionPerformed` и аргумент — это `ActionEvent`, подразумевается для любого аргумента `addActionListener`. Единственная новая информация здесь — выполняемый код, а именно вызов функции `println`. Эта новая информация просто глушится шаблонным кодом. Читатели кода должны обладать пронзительным орлиным взглядом, чтобы пробиться сквозь шум и обнаружить информативную часть.

Более подходящая для Scala версия получит в качестве аргумента функцию, существенно сокращая объем шаблонного кода:

```
button.addActionListener( // Несоответствие типов!
  (_: ActionEvent) => println("pressed!")
)
```

В данном виде этот код работать не будет[1]. Метод `addActionListener` требует слушателя действий, а получает функцию. Но используя неявное преобразование, этот код можно привести в рабочее состояние.

Сначала нужно создать неявное преобразование между двумя типами. Это преобразование из функций в слушатели действий выглядит так:

```
implicit def function2ActionListener(f: ActionEvent => Unit) =
  new ActionListener {
    def actionPerformed(event: ActionEvent) = f(event)
  }
```

Это метод с одним аргументом, получающий функцию и возвращающий слушатель действий. Как и любой другой метод с одним аргументом, его можно вызвать напрямую, а его результат — передать другому выражению:

```
button.addActionListener(
  function2ActionListener(
    (_: ActionEvent) => println("pressed!")
  )
)
```

Этот фрагмент уже лучше версии с внутренним классом. Обратите внимание, как необоснованно большой объем шаблонного кода был сведен к замене функциональным литералом и вызову метода. Улучшить код удалось благодаря использованию неявного преобразования. Поскольку метод `function2ActionListener` помечен ключевым словом `implicit`, то может быть опущен, и компилятор вставит его автоматически. В результате получится следующий код:

```
// Теперь все работает
button.addActionListener(
  (_: ActionEvent) => println("pressed!")
)
```

Этот код работает благодаря тому, что компилятор сначала пробует скомпилировать все как есть, но видит ошибку типа. Прежде чем окончательно сдаться, он ищет неявное преобразование, которое смогло бы исправить ситуацию. В данном случае находит `function2ActionListener`, пробует применить данный метод преобразования, видит, что тот работает, и движется дальше. Здесь компилятор старается сделать так, чтобы разработчик мог проигнорировать еще одну волокитную деталь. Что попробовать: слушатель действий или функцию события? Либо одно, либо другое сработает, и компилятор использует то, что подходит больше.

В данном разделе мы продемонстрировали силу неявных преобразований и то, как они позволяют вам приукрасить существующие библиотеки. В следующих разделах мы рассмотрим правила, которые определяют, когда выполняются неявные преобразования и как они будут найдены.

---

[1] В разделе 31.5 мы объясним, что этот код будет работать в Scala 2.12.

21.2. Правила для неявных преобразований

К неявным определениям относятся те, которые компилятору разрешено вставлять в программу для устранения любой из ее ошибок типов. Например, если `x + y` не проходит проверку типов, то компилятор может изменить этот код, чтобы получилось `convert(x) + y`, где под `convert` понимается некое доступное неявное преобразование. Если `convert` переделывает `x` в нечто имеющее метод `+`, то это изменение может исправить программу, чтобы она прошла проверку типов и выполнялась корректно. Если же `convert` — всего лишь простая функция преобразования, то избавление от нее исходного кода может сделать его более понятным.

Неявные преобразования регулируются общими правилами, представленными ниже.

- **Правило маркировки: доступны только определения с пометкой implicit.** Ключевое слово `implicit` используется в целях маркировки того объявления, которое компилятор может применять в качестве неявного. Им можно помечать любое объявление переменной, функции или объекта. Пример объявления неявной функции выглядит следующим образом[1]:

  ```
  implicit def intToString(x: Int) = x.toString
  ```

  Если `convert` имеет пометку `implicit`, то компилятор всего лишь превратит `x + y` в `convert(x) + y`. Тем самым устраняется путаница, которая может возникнуть при выборе компилятором случайных функций, оказавшихся в области видимости, и вставке их в качестве преобразований. Свой выбор компилятор будет делать, исходя только лишь из определений, имеющих явную пометку `implicit`.

- **Правило области видимости: вставляемое неявное преобразование должно находиться в области видимости в качестве простого идентификатора или быть связанным с исходным или целевым типом преобразования.** Компилятор Scala будет рассматривать только те неявные преобразования, которые находятся в области видимости. Поэтому, чтобы обеспечить доступность неявного преобразования, нужно неким образом поместить его в область видимости. Более того, за единственным исключением, неявное преобразование должно находиться в области видимости в качестве *простого идентификатора*. Компилятор не станет вставлять преобразование в форме `someVariable.convert`. Например, он не станет расширять `x + y` в `someVariable.convert(x) + y`. Если нужно сделать идентификатор `someVariable.convert` доступным в качестве неявного преобразования, то его нужно будет импортировать, что сделает его доступным в качестве простого идентификатора. После импортирования

[1] В качестве неявных параметров могут использоваться переменные и объекты-одиночки с пометкой implicit. Этот вариант использования мы рассмотрим в данной главе чуть позже.

компилятор сможет свободно применить его как `convert(x) + y`. Фактически в библиотеки зачастую включают объект `Preamble`, содержащий ряд полезных неявных преобразований. После этого в коде, который задействует библиотеку, можно будет для обращения к имеющимся в библиотеке неявным преобразованиям однократно применить выражение `import Preamble._`.

В правиле простого идентификатора есть одно исключение. Компилятор будет искать неявные определения в объекте-компаньоне исходного или ожидаемого целевого типа преобразования. Например, при попытке передать объект типа `Dollar` методу, получающему `Euro`, исходным типом является `Dollar`, а целевым типом — `Euro`. Поэтому можно запаковать неявное преобразование из `Dollar` в `Euro` в объект-компаньон любого класса, `Dollar` или `Euro`.

Пример, в котором неявное преобразование помещено в объект-компаньон класса `Dollar`, выглядит следующим образом:

```
object Dollar {
  implicit def dollarToEuro(x: Dollar): Euro = ...
}
class Dollar { ... }
```

В данном случае преобразование `dollarToEuro` считается *ассоциированным* с типом `Dollar`. Компилятор найдет такое ассоциированное преобразование всякий раз, когда возникает необходимость выполнить преобразование из экземпляра типа `Dollar`. Отдельно импортировать преобразование в вашу программу нет никакой необходимости.

Правило области видимости помогает рассуждать модульно. Когда вы читаете код файла, все, что вам нужно для понимания других файлов, либо должно быть импортировано, либо на это должна быть явная ссылка с полным указанием имени. Это, по крайней мере, также важно для неявных преобразований, как для явно написанного кода. Если бы неявные преобразования распространялись на всю систему, то, чтобы понять код файла, вам нужно было бы знать о каждом неявном преобразовании, используемом в программе!

- **Правило «по одному»: может быть вставлено только одно неявное преобразование.** Компилятор никогда не станет преобразовывать `x + y` в `convert1(convert2(x)) + y`. Подобные действия приведут к резкому увеличению времени компиляции ошибочного кода и увеличат разницу между тем, что пишет программист, и тем, что фактически делает программа. Чтобы сохранить здравый смысл, компилятор не вставляет дополнительные неявные преобразования, будучи уже на полпути к применению другого неявного преобразования. Но это ограничение можно обойти за счет неявных параметров неявного преобразования, речь о которых пойдет чуть позже.
- **Правило «сначала явное»: когда код в том виде, в котором он записан, проходит проверку типов, не делаются попытки применения неявных преобразований.** Компилятор не станет изменять уже работающий код. Следствие данного правила — возможность заменить неявное преобразование явно указанным.

Это удлиняет код, но устраняет его очевидную двусмысленность. В каждом отдельно взятом случае можно менять один вариант на другой. Если код становится повторяющимся и многословным, то от однообразия поможет избавиться неявное преобразование. Если же краткость кода делает его непонятным, то преобразования можно указать явно. Количество неявных преобразований, которые вы оставляете компилятору, — вопрос стиля.

Названия неявных преобразований

Неявные преобразования могут носить произвольные имена. Имя неявного преобразования играет роль только в двух ситуациях: если вы хотите указать его явно в применении метода и если нужно определить, какие неявные преобразования доступны в том или ином месте программы. Чтобы проиллюстрировать вторую ситуацию, предположим, что есть объект с двумя неявными преобразованиями:

```
object MyConversions {
  implicit def stringWrapper(s: String):
      IndexedSeq[Char] = ...
  implicit def intToString(x: Int): String = ...
}
```

В создаваемом приложении вам следует воспользоваться преобразованием `stringWrapper` и совсем не нужно, чтобы целочисленные значения автоматически конвертировались в строки с помощью преобразования `intToString`. Этого можно добиться импортированием только одного преобразования из двух:

```
import MyConversions.stringWrapper
... // код воспользуется stringWrapper
```

В данном примере было важно, чтобы у неявных преобразований были имена, поскольку это единственный способ выполнить импортирование выборочно, применив только одно из двух преобразований.

Где применяются неявные преобразования

Неявные преобразования применяются в языке в трех местах: в преобразованиях в ожидаемый тип, в преобразованиях получателя при выборе и в неявных параметрах. Неявные преобразования в ожидаемый тип позволяют использовать один тип в том контексте, в котором ожидается другой тип. Например, можно располагать значением типа `String` и иметь желание передать его методу, требующему значения типа `IndexedSeq[Char]`. Преобразования получателя позволяют адаптировать получатель вызова метода (то есть объект, в отношении которого метод был вызван), если метод непригоден для исходного типа. Примером может послужить выражение `"abc".exists`, которое в результате преобразования приобретает вид `stringWrapper("abc").exists`, поскольку метод `exists` недоступен в классе `Strings`,

но доступен в классе `IndexedSeqs`. Что же касается неявных параметров, то они обычно служат в целях предоставления вызываемой функции более подробной информации о том, что именно нужно вызывающему объекту. Неявные параметры особенно хорошо подходят для использования с обобщенными функциями, где вызываемая функция порой вообще ничего не знает о типе одного или нескольких аргументов. Все эти разновидности неявных преобразований мы рассмотрим в следующих разделах.

21.3. Неявное преобразование в ожидаемый тип

Неявное преобразование в ожидаемый тип — первое место, в котором компилятор будет использовать неявные преобразования. Правило простое. Там, где компилятор видит *X*, а нужен *Y*, он станет искать неявную функцию, выполняющую преобразование *X* в *Y*. Например, обычно значение типа `Double` не может применяться в качестве целочисленного значения, поскольку будет потеряна точность:

```
scala> val i: Int = 3.5
<console>:7: error: type mismatch;
 found   : Double(3.5)
 required: Int
       val i: Int = 3.5
                    ^
```

Но чтобы сгладить ситуацию, можно определить неявное преобразование:

```
scala> implicit def doubleToInt(x: Double) = x.toInt
doubleToInt: (x: Double)Int

scala> val i: Int = 3.5
i: Int = 3
```

Компилятор видит значение типа `Double`, а именно `3.5`, в контексте, требующем `Int`. Прежде компилятор усмотрел бы в этом обычную ошибку несоответствия типов. Но теперь он не сдается сразу, а ищет средство реализовать неявное преобразование `Double` в `Int`. В данном случае он находит такое средство `doubleToInt`, поскольку оно находится в области видимости в качестве простого идентификатора. (Вне интерпретатора `doubleToInt` можно ввести в область видимости с помощью импортирования или, возможно, наследования.) Затем компилятор автоматически вставляет вызов `doubleToInt`. Скрытым образом код приобретает следующий вид:

```
val i: Int = doubleToInt(3.5)
```

Это действительно *неявное* преобразование. Оно не запрашивается в явном виде. Вместо этого `doubleToInt` помечается в качестве доступного неявного преобразования с помощью ввода его в область видимости в качестве простого идентификатора, а затем компилятор автоматически использует его при необходимости конвертировать из `Double` в `Int`.

Преобразование `Double`-значений в `Int`-значения может вызвать некоторое недоумение, поскольку сомнительна сама идея наличия чего-то, что скрыто вызывает потерю точности. То есть на самом-то деле мы не рекомендуем использовать подобное преобразование. Куда больше смысла можно усмотреть в обратном — в переходе от более ограниченного типа к более общему. Например, `Int`-значение можно преобразовать без потери точности в `Double`-значение, следовательно, в преобразовании `Int` в `Double` есть смысл. Фактически именно так и происходит. В объекте `scala.Predef`, который неявно импортируется в каждую программу на Scala, определено неявное преобразование, выполняющее конвертацию более «мелкого» типа в более «крупный». Например, в `Predef` можно найти такое преобразование:

```
implicit def int2double(x: Int): Double = x.toDouble
```

Вот почему в Scala `Int`-значения могут сохраняться в переменных типа `Double`. В системе типов специальных правил для этого нет, это просто неявное преобразование[1].

21.4. Преобразование получателя

Неявное преобразование применяется также к получателю вызова метода — объекту, в отношении которого вызывается метод. Этот вид неявного преобразования используется в двух основных вариантах. Первый заключается в том, что преобразования получателя позволяют более гладко интегрировать новый класс в существующую иерархию классов. А второй состоит в поддержке возможности написания внутри базового языка предметно-ориентированных языков (domain-specific languages, DSL).

Чтобы посмотреть все это в действии, предположим, что вы записали код `obj.doIt`, а у `obj` нет элемента по имени `doIt`. Прежде чем сдаться, компилятор попытается вставить преобразования. В данном случае преобразование следует применить к получателю, `obj`. Компилятор будет действовать так, словно ожидаемый тип `obj` включает в себя свойство «содержит член по имени `doIt`». Данный тип «содержит `doIt`» не есть обычный тип Scala, но концептуально является типом, и поэтому в данном случае компилятор вставит неявное преобразование.

Взаимодействие с новыми типами

Как уже упоминалось, один из основных вариантов применения преобразования получателя заключается в разрешении более гладкой интеграции новых типов с уже существующими. В частности, подобные преобразования позволяют разрешить создателям клиентских программ воспользоваться экземплярами

[1] Но внутренний код компилятора Scala станет рассматривать преобразование особым образом, переводя его в специальный байт-код i2d. Следовательно, скомпилированный образ будет таким же, как и в Java.

существующих типов, словно те являются экземплярами ваших новых типов. Возьмем, к примеру, класс `Rational`, показанный в листинге 6.5 (см. выше). Рассмотрим еще раз фрагмент этого класса:

```
class Rational(n: Int, d: Int) {
  ...
  def + (that: Rational): Rational = ...
  def + (that: Int): Rational = ...
}
```

В классе `Rational` имеются два перегруженных варианта метода `+`, получающих в качестве аргументов соответственно `Rational`- и `Int`-значения. Следовательно, можно сложить либо два рациональных числа, либо рациональное число с целым:

```
scala> val oneHalf = new Rational(1, 2)
oneHalf: Rational = 1/2

scala> oneHalf + oneHalf
res0: Rational = 1/1

scala> oneHalf + 1
res1: Rational = 3/2
```

А как будут обстоять дела с выражением наподобие `1 + oneHalf`? Это довольно каверзное выражение, поскольку у получателя, `1`, нет подходящего метода `+`. Следовательно, при выполнении следующего кода возникнет ошибка:

```
scala> 1 + oneHalf
<console>:6: error: overloaded method value + with
alternatives (Double)Double <and> ... cannot be applied
to (Rational)
     1 + oneHalf
       ^
```

Чтобы разрешить применение подобной смешанной арифметики, нужно определить неявное преобразование из `Int` в `Rational`:

```
scala> implicit def intToRational(x: Int) =
         new Rational(x, 1)
intToRational: (x: Int)Rational
```

При использовании этого преобразования получатель проделает следующий трюк:

```
scala> 1 + oneHalf
res2: Rational = 3/2
```

Здесь компилятор Scala скрыто сначала пытается проверить соответствие типов в отношении выражения `1 + oneHalf` в его неизменном виде. Проверка проходит неудачно, поскольку в `Int` есть несколько методов `+`, однако ни один из них не принимает аргумент типа `Rational`. Затем компилятор ищет неявное преобразование из `Int` в другой тип, у которого есть метод `+`, применимый к значениям типа `Rational`.

Он находит ваше преобразование и применяет его, в результате чего получается следующее:

```
intToRational(1) + oneHalf
```

В данном случае компилятор находит функцию неявного преобразования, поскольку ее определение было введено в интерпретатор, который поместил это определение в область видимости для оставшегося сеанса работы с интерпретатором.

Имитация нового синтаксиса

Другой основной вариант использования неявных преобразований — имитация добавления нового синтаксиса. Вспомним, что отображение типа `Map` можно создать с помощью следующего синтаксиса:

```
Map(1 -> "one", 2 -> "two", 3 -> "three")
```

А вас не интересовало, как поддерживается `->`? Это ведь не синтаксис! Элемент вида `->` — метод класса `ArrowAssoc`, который определен внутри стандартной преамбулы Scala (`scala.Predef`). В ней также определяется неявное преобразование из `Any` в `ArrowAssoc`. При использовании кода `1 -> "one"` компилятор вставляет преобразование из `1` в `ArrowAssoc`, чтобы можно было найти метод `->`. Соответствующие определения выглядят так:

```
package scala
object Predef {
  class ArrowAssoc[A](x: A) {
    def -> [B](y: B): Tuple2[A, B] = Tuple2(x, y)
  }
  implicit def any2ArrowAssoc[A](x: A): ArrowAssoc[A] =
    new ArrowAssoc(x)
  ...
}
```

Такая схема обогащающих оболочек часто встречается в библиотеках, которые предоставляют языку расширения, похожие на синтаксис. Поэтому, как только встретится эта схема, нужно быть готовыми ее распознать. Момент обнаружения того, что некий объект вызывает методы, не существующие в классе-получателе, означает, скорее всего, использование неявных преобразований. Аналогично если встретится класс, имя которого означает обогащение чего-либо — `RichSomething` (например, `RichInt` или `RichBoolean`), то в нем, вероятно, к типу `Something` добавляются методы, похожие на синтаксис.

Здесь эти схемы обогащающих оболочек для базовых типов, рассмотренных в главе 5, уже встречались вам. Теперь можно увидеть, что обогащающие оболочки применяются более широко, зачастую позволяя использовать внутренние DSL-языки, определяемые в виде библиотеки, в то время как программистам на других языках может понадобиться разработка внешних DSL-языков.

Неявные классы

Неявные классы были добавлены в Scala 2.10 с целью облегчить создание обогащающих классов-оболочек. Неявным называется класс, перед определением которого ставится ключевое слово `implicit`. Для любого такого класса компилятор создает неявное преобразование из параметра конструктора класса в сам класс. Если планируется использовать класс для паттерна обогащающей оболочки, то такое преобразование — именно то, что нужно.

Предположим, к примеру, что имеется класс по имени `Rectangle` для представления ширины и высоты прямоугольника на экране:

```
case class Rectangle(width: Int, height: Int)
```

Может возникнуть желание прибегнуть к паттерну обогащающей оболочки, чтобы упростить представление при весьма частом использовании этого класса. Один из способов решения данной задачи выглядит так:

```
implicit class RectangleMaker(width: Int) {
  def x(height: Int) = Rectangle(width, height)
}
```

В показанном ранее коде в необычной манере определяется класс `RectangleMaker`. Вдобавок ко всему он вызывает автоматическое создание следующего преобразования:

```
// Создается автоматически
implicit def RectangleMaker(width: Int) =
  new RectangleMaker(width)
```

В результате можно создавать точки, помещая `x` между двумя целочисленными значениями:

```
scala> val myRectangle = 3 x 4
  myRectangle: Rectangle = Rectangle(3,4)
```

А вот как это работает: поскольку в типе `Int` нет метода по имени `x`, то компилятор станет искать неявное преобразование из `Int` в нечто имеющее данный метод. Он найдет созданное преобразование `RectangleMaker`, а в `RectangleMaker` имеется метод по имени `x`. Компилятор вставляет вызов этого преобразования, затем вызывает в отношении `x` проверку соответствия типов и делает то, что нужно.

Авантюристам следует иметь в виду: не стоит обольщаться насчет того, что указать ключевое слово `implicit` можно перед определением любого класса. Это не так. В качестве неявного нельзя использовать `case`-класс, и его конструктор может иметь только один параметр. Кроме того, неявный класс должен размещаться внутри какого-либо другого объекта — класса или трейта. На практике, поскольку неявные классы применяются в качестве обогащающих оболочек для добавления новых методов в существующий класс, эти ограничения не имеют значения.

21.5. Неявные параметры

Осталось еще одно место, куда компилятор помещает неявные преобразования, — это список аргументов. Компилятор иногда заменяет код вида `someCall(a)` кодом вида `someCall(a)(b)` или код `SomeClass(a)` — кодом `new SomeClass(a)(b)`, добавляя тем самым пропущенный список параметров, чтобы завершить вызов функции. Предоставляется не просто последний параметр, а весь последний каррированный список параметров. Например, если `someCall` с пропущенным списком параметров получает три параметра, то компилятор может вместо `someCall(a)` вставить `someCall(a)(b, c, d)`. Чтобы воспользоваться этой возможностью, пометку `implicit` при своем определении должны иметь не только вставляемые идентификаторы, такие как `b`, `c` и `d` в `(b, c, d)`, но и последний список параметров в `someCall` или определение `someClass`.

Рассмотрим простой пример. Предположим, имеется класс `PreferredPrompt`, в котором инкапсулирована строка приглашения к вводу, используемая в оболочке (такая как, скажем, `"$ "` или `"> "`), предпочитаемая пользователем:

```
class PreferredPrompt(val preference: String)
```

Предположим вдобавок, что имеется объект `Greeter` с методом `greet`, принимающим два списка параметров. Первый список параметров получает строку с именем пользователя, а второй — `PreferredPrompt`:

```
object Greeter {
  def greet(name: String)(implicit prompt: PreferredPrompt) = {
    println("Welcome, " + name + ". The system is ready.")
    println(prompt.preference)
  }
}
```

Последний список параметров имеет метку `implicit`, означающую, что он может предоставляться в неявном режиме. Но при этом по-прежнему сохраняется возможность явно указать символ приглашения к вводу с помощью такого кода:

```
scala> val bobsPrompt = new PreferredPrompt("relax> ")
bobsPrompt: PreferredPrompt = PreferredPrompt@714d36d6

scala> Greeter.greet("Bob")(bobsPrompt)
Welcome, Bob. The system is ready.
relax>
```

Чтобы позволить компилятору предоставить параметр в неявном режиме, сначала нужно определить переменную ожидаемого типа, которым в данном случае будет `PreferredPrompt`. Сделать это можно, к примеру, в объекте предпочтений:

```
object JoesPrefs {
  implicit val prompt = new PreferredPrompt("Yes, master> ")
}
```

Заметьте, что у самой val-переменной есть метка implicit. Не будь ее, компилятор не стал бы использовать эту переменную для предоставления пропущенного списка параметров. Он также не стал бы ее задействовать, если бы она, как показано в следующем примере, отсутствовала в области видимости в качестве простого идентификатора:

```
scala> Greeter.greet("Joe")
<console>:13: error: could not find implicit value for
parameter prompt: PreferredPrompt
              Greeter.greet("Joe")
                            ^
```

Но после ее введения в область видимости с помощью ключевого слова import компилятор воспользуется ею, чтобы предоставить пропущенный список параметров:

```
scala> import JoesPrefs._
import JoesPrefs._

scala> Greeter.greet("Joe")
Welcome, Joe. The system is ready.
Yes, master>
```

Обратите внимание на то, что ключевое слово implicit применяется ко всему списку параметров, а не к отдельным параметрам. В листинге 21.1 показан пример, в котором последний параметр списка метода greet, принадлежащего объекту Greeter, к тому же имеющему метку implicit, содержит два параметра: prompt типа PreferredPrompt и drink типа PreferredDrink.

Листинг 21.1. Неявный список параметров с несколькими параметрами

```
class PreferredPrompt(val preference: String)
class PreferredDrink(val preference: String)

object Greeter {
  def greet(name: String)(implicit prompt: PreferredPrompt,
      drink: PreferredDrink) = {

    println("Welcome, " + name + ". The system is ready.")
    print("But while you work, ")
    println("why not enjoy a cup of " + drink.preference + "?")
    println(prompt.preference)
  }
}

object JoesPrefs {
  implicit val prompt = new PreferredPrompt("Yes, master> ")
  implicit val drink = new PreferredDrink("tea")
}
```

В объекте-одиночке JoesPrefs объявляются две неявные val-переменные: prompt типа PreferredPrompt и drink, типа PreferredDrink. Но, как и прежде,

поскольку их нет в области видимости в качестве простых идентификаторов, то они не будут использоваться для заполнения пропущенного списка параметров для `greet`:

```
scala> Greeter.greet("Joe")
<console>:19: error: could not find implicit value for
parameter prompt: PreferredPrompt
              Greeter.greet("Joe")
                            ^
```

Ввести обе неявные `val`-переменные в область видимости можно, воспользовавшись ключевым словом `import`:

```
scala> import JoesPrefs._
import JoesPrefs._
```

Теперь и `prompt`, и `drink` находятся в области видимости в форме простых идентификаторов, поэтому их можно использовать для предоставления в качестве неявного списка параметров:

```
scala> Greeter.greet("Joe")(prompt, drink)
Welcome, Joe. The system is ready.
But while you work, why not enjoy a cup of tea?
Yes, master>
```

А поскольку теперь соблюдены все правила использования неявных параметров, то можно задействовать альтернативный вариант и позволить компилятору Scala предоставить вам `prompt` и `drink`, пропустив для этого указание последнего списка параметров:

```
scala> Greeter.greet("Joe")
Welcome, Joe. The system is ready.
But while you work, why not enjoy a cup of tea?
Yes, master>
```

По поводу предыдущего примера следует заметить: в нем `String` в качестве типа переменной `prompt` или переменной `drink` не используется, даже притом, что в конечном итоге каждая из них через свои поля `preference` предоставляет значение типа `String`. Поскольку компилятор выбирает параметры путем поиска совпадения типов параметров с типом значений в области видимости, то неявные параметры имеют редкие или специальные типы, для которых случайные совпадения маловероятны. Например, типы `PreferredPrompt` и `PreferredDrink` в представленном выше листинге 21.1 были определены исключительно для того, чтобы послужить в качестве типов неявных параметров. Поэтому вряд ли неявные переменные данных типов находились бы в области видимости, если бы не предполагалось их использование в качестве неявных параметров для `Greeter.greet`.

У неявных параметров есть еще один нюанс: они, пожалуй, наиболее часто применяются для предоставления информации о типе, упомянутом *явным* образом в предшествующем списке параметров, как и классы типа языка Haskell.

Рассмотрим в качестве примера функцию `maxListOrdering`, показанную в листинге 21.2, которая возвращает максимальный элемент из переданного списка.

Листинг 21.2. Функция с верхним ограничителем

```
def maxListOrdering[T](elements: List[T])
      (ordering: Ordering[T]): T =
  elements match {
    case List() =>
      throw new IllegalArgumentException("empty list!")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxListOrdering(rest)(ordering)
      if (ordering.gt(x, maxRest)) x
      else maxRest
}
```

Сигнатура `maxListOrdering` похожа на сигнатуру `orderedMergeSort`, показанную в листинге 19.12 (см. выше): в качестве своих аргументов функция получает `List[T]` и теперь получает дополнительный аргумент типа `Ordering[T]`. Этот дополнительный аргумент указывает, какой порядок следует задействовать при сравнении элементов типа `T`. Таким образом, данная версия может применяться для типов, не имеющих встроенного механизма выстраивания по порядку. Кроме того, она может использоваться для типов, имеющих встроенное упорядочение, но время от времени *требующих* некоего другого варианта упорядочения.

Это более универсальная, но и более громоздкая в использовании версия. Теперь вызывающий объект должен указать явное упорядочение, даже если в качестве `T` выступает нечто относящееся к типам `String` или `Int`, явно имеющим исходные механизмы упорядочения. Чтобы сделать новый метод более удобным в применении, неплохо было бы превратить второй аргумент в неявный. Данный подход показан в листинге 21.3.

Листинг 21.3. Функция с неявным параметром

```
def maxListImpParm[T](elements: List[T])
      (implicit ordering: Ordering[T]): T =

  elements match {
    case List() =>
      throw new IllegalArgumentException("empty list!")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxListImpParm(rest)(ordering)
      if (ordering.gt(x, maxRest)) x
      else maxRest
  }
```

В данном примере параметр `ordering` служит для описания упорядочения значений типа `T`. В теле `maxListImpParm` это упорядочение используется в двух местах: в рекурсивном вызове `maxListImpParm` и в выражении `if`, которое проверяет, больше ли элемент `head` списка, чем максимальный элемент всего остального списка.

Функция `maxListImpParm` — пример использования неявного параметра в целях предоставления дополнительной информации о типе, упомянутом в явном виде в предшествующем списке параметров. Если говорить точнее, то подразумеваемый параметр `ordering`, относящийся к типу `Ordering[T]`, предоставляет больше информации о типе `T` — в данном случае о том, как следует выстраивать по порядку значения, относящиеся к данному типу. Он упомянут в `List[T]` — типе элементов параметра, который появляется в предшествующем списке параметров. Элементы при любом вызове `maxListImpParm` всегда должны предоставляться в явном виде, поэтому компилятор к моменту компиляции будет знать, что такое `T`, и сможет определить, доступно ли неявное определение типа `Ordering[T]`. Если да, то компилятор сможет передать второй список параметров, `ordering`, воспользовавшись механизмом предоставления неявных параметров.

Данная схема получила настолько широкое распространение, что стандартная библиотека Scala предоставляет неявные методы упорядочения для множества самых востребованных типов. Благодаря этому метод `maxListImpParm` можно использовать при работе с различными типами:

```
scala> maxListImpParm(List(1,5,10,3))
res9: Int = 10

scala> maxListImpParm(List(1.5, 5.2, 10.7, 3.14159))
res10: Double = 10.7

scala> maxListImpParm(List("one", "two", "three"))
res11: String = two
```

В первом случае компилятор вставит упорядочение для `Int`-значений, во втором — для `Double`-значений, а в третьем — для `String`-значений.

Правило стиля для неявных параметров

Согласно правилу стиля для типов неявных параметров лучше воспользоваться типом, названным особым образом. Например, переменные `prompt` и `drink` в предыдущем примере относились не к типу `String`, а к типам `PreferredPrompt` и `PreferredDrink` соответственно. В качестве контрпримера рассмотрим возможность того, что функцию `maxListImpParm` можно записать с помощью такой сигнатуры типа:

```
def maxListPoorStyle[T](elements: List[T])
      (implicit orderer: (T, T) => Boolean): T
```

Но чтобы воспользоваться данной версией функции, вызывающему объекту придется предоставить параметр `orderer`, который относится к типу `(T, T) => Boolean`. Это довольно общий тип, включающий любую функцию, выдающую на основе двух значений типа `T` результат типа `Boolean`. Из него абсолютно неясно предназначение данного типа — это может быть проверка на равенство, на то, что одно значение больше или меньше другого, или же нечто иное.

В настоящем коде для `maxListImpParm`, который приведен в листинге 21.3 (см. выше), показан более удачный стиль. В нем используется параметр `ordering`, относящийся к типу `Ordering[T]`. Слово `Ordering` в наименовании типа явно указывает на цель применения неявного параметра: он предназначен для упорядочения элементов типа `T`. Поскольку этот тип упорядочения указан явно, то добавление в стандартную библиотеку неявных предоставителей этого типа не вызовет никаких проблем. Но представьте хаос, который возникнет при добавлении в стандартную библиотеку неявного преобразования типа `(T, T) => Boolean`, когда компилятор начнет разбрасывать его в чьем-то коде. В результате будет получен код, который пройдет компиляцию и сможет запускаться на выполнение, но проверки в отношении пар элементов станут выполняться беспорядочно! Стало быть, нужно придерживаться следующего правила стиля: использовать в типе неявного параметра хотя бы одно имя, определяющее его роль.

21.6. Контекстные ограничители

В предыдущем примере возможность применения неявного преобразования была показана, но не реализована. Следует заметить, что при использовании неявного преобразования в качестве параметра компилятор не только попытается *предоставить* этот параметр с неявным значением, но и *воспользуется* им в качестве доступного неявного элемента в теле метода (листинг 21.4)! Таким образом, первое использование `ordering` внутри тела метода можно опустить.

Листинг 21.4. Функция, использующая неявный параметр внутри себя

```
def maxList[T](elements: List[T])
      (implicit ordering: Ordering[T]): T =

  elements match {
    case List() =>
      throw new IllegalArgumentException("empty list!")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxList(rest)     // Здесь (ordering) подразумевается
      if (ordering.gt(x, maxRest)) x  // А этот ordering по-прежнему
      else maxRest                    // указан явно
  }
```

Начав анализировать код, показанный в листинге 21.4, компилятор увидит, что типы не совпадают. Выражению `maxList(rest)` предоставлен только один список параметров, а `maxList` требует два списка. Второй список параметров является неявным, и потому компилятор не спешит сдаваться на проверке соответствия типов. Вместо этого он ищет неявный параметр подходящего типа, в данном случае `Ordering[T]`. Что касается нашего кода, то компилятор находит такой параметр и перезаписывает вызов в `maxList(rest)(ordering)`, после чего код успешно проходит проверку соответствия типов.

Существует также способ исключить второе использование `ordering`. Для него применяется следующий метод, определение которого находится в стандартной библиотеке:

```
def implicitly[T](implicit t: T) = t
```

Эффект от вызова `implicitly[Foo]` заключается в том, что компилятор станет искать неявное определение типа `Foo`. Затем он вызывает в отношении этого объекта неявный метод, который, в свою очередь, возвращает объект обратно. Таким образом, можно воспользоваться кодом `implicitly[Foo]` везде, где нужно найти неявный объект типа `Foo` в текущей области видимости. Например, в листинге 21.5 показано применение `implicitly[Ordering[T]]` в целях извлечения параметра `ordering` по его типу.

Листинг 21.5. Функция, используемая неявно

```
  def maxList[T](elements: List[T])
        (implicit ordering: Ordering[T]): T =

    elements match {
      case List() =>
        throw new IllegalArgumentException("empty list!")
      case List(x) => x
      case x :: rest =>
        val maxRest = maxList(rest)
        if (implicitly[Ordering[T]].gt(x, maxRest)) x
        else maxRest
    }
```

Присмотритесь к последней версии `maxList` повнимательнее. В ней в тексте метода нет ни одного упоминания о параметре `ordering`. Второй параметр можно также назвать `comparator`.

```
def maxList[T](elements: List[T])
      (implicit comparator: Ordering[T]): T = // то же самое тело...
```

По той же причине работает и эта версия:

```
def maxList[T](elements: List[T])
      (implicit iceCream: Ordering[T]): T = // то же самое тело...
```

Поскольку данная схема получила широкое распространение, в Scala допускается не указывать имя этого параметра и сократить заголовок метода, используя *контекстные ограничители*. В этом случае можно будет записывать сигнатуру `maxList`, как показано в листинге 21.6. Синтаксис `[T : Ordering]` является контекстным ограничителем с двойным назначением. Во-первых, он вводит в качестве обычного параметр типа `T`. Во-вторых, добавляет неявный параметр типа `Ordering[T]`. В предыдущих версиях `maxList` этот параметр назывался `ordering`, но применение контекстного ограничителя не позволяет узнать, какой именно параметр будет вызываться. Как было показано ранее, зачастую знать это и не нужно.

Листинг 21.6. Функция с контекстным ограничителем

```
def maxList[T : Ordering](elements: List[T]): T =
  elements match {
    case List() =>
      throw new IllegalArgumentException("empty list!")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxList(rest)
      if (implicitly[Ordering[T]].gt(x, maxRest)) x
      else maxRest
  }
```

На интуитивном уровне контекстный ограничитель можно воспринимать как высказывание о параметре типа. В момент записи `[T <: Ordered[T]]` говорится, что `T` — тип `Ordered[T]`. В отличие от этого при записи `[T : Ordering]` о том, что такое `T`, столь подробно не сообщается — говорится о наличии некой формы упорядочения, связанной с `T`. Таким образом, контекстный ограничитель обладает достаточной гибкостью. Он позволяет использовать код, требующий упорядочения, или любое другое свойство типа, не прибегая к необходимости изменять определение данного типа.

21.7. Когда применяются множественные преобразования

Может случиться так, что в области видимости будут находиться сразу несколько неявных преобразований, каждое из которых должно работать. В большинстве случаев Scala в подобной ситуации отказывается вставлять преобразование. Неявные преобразования хорошо работают, когда преобразование остается абсолютно очевидным и схематически чистым. Если применяется сразу несколько преобразований, то выбор не столь очевиден.

Рассмотрим простой пример. В нем есть метод, получающий последовательность, и два преобразования: одно превращает целочисленное значение в диапазон, второе превращает целочисленное значение в список цифр:

```
scala> def printLength(seq: Seq[Int]) = println(seq.length)
printLength: (seq: Seq[Int])Unit

scala> implicit def intToRange(i: Int) = 1 to i
intToRange: (i:
Int)scala.collection.immutable.Range.Inclusive

scala> implicit def intToDigits(i: Int) =
         i.toString.toList.map(_.toInt)
intToDigits: (i: Int)List[Int]

scala> printLength(12)
```

```
<console>:26: error: type mismatch;
 found : Int(12)
 required: Seq[Int]
Note that implicit conversions are not applicable because
they are ambiguous:
 both method intToRange of type (i:
Int)scala.collection.immutable.Range.Inclusive
 and method intToDigits of type (i: Int)List[Int]
 are possible conversion functions from Int(12) to Seq[Int]
              printLength(12)
                          ^
```

Здесь, несомненно, есть неоднозначность. Преобразование целочисленного значения в последовательность цифр коренным образом отличается от его преобразования в диапазон. В данном случае программист должен указать, какому из преобразований предназначено стать неявным. Вплоть до выхода Scala 2.7 на этом вся история и заканчивалась. В случае применения сразу нескольких неявных преобразований компилятор отказывался делать выбор между ними. Возникала такая же ситуация, как и с перегрузкой методов. При попытке вызова `foo(null)` и наличии двух различных перегруженных методов `foo`, допускающих `null`, компилятор отвергал код. Он говорил, что цель вызова метода неоднозначна.

В Scala 2.8 это правило было смягчено. Если одно из доступных преобразований *конкретнее* других, то компилятор выберет его. Замысел таков: при наличии достаточных оснований полагать, что программист всегда выбрал бы только одно из имеющихся преобразований, ему не требуется записывать это преобразование в явном виде. Ведь у правила перегрузки методов имеется точно такое же послабление. Можем продолжить рассмотрение предыдущего примера. При условии, что один из доступных методов `foo` получает значение типа `String`, а другие получают значение типа `Any`, выбор будет за версией, получающей `String`. Вполне очевидно, что этот вариант более конкретен.

Уточним сказанное. Одно неявное преобразование будет *конкретнее* другого, если к нему применимо одно из следующих утверждений:

- тип аргумента первого преобразования — подтип второго;
- оба преобразования являются методами, и класс, в который входит первое, расширяет класс, в который входит второе.

Мотивом для возвращения к этому вопросу и пересмотра правила было улучшение взаимодействия между коллекциями Java, коллекциями Scala и строками.

Рассмотрим простой пример:

```
val cba = "abc".reverse
```

Какой тип будет выведен для `cba`? Интуитивно понятно, что тип должен быть `String`. Реверсирование строки должно порождать другую строку, верно? Но в Scala 2.7 получалось так, что строка `"abc"` превращалась в коллекцию Scala. Реверсирование коллекции Scala порождало коллекцию Scala, следовательно, типом `cba` будет коллекция. Вдобавок там происходило неявное преобразование

обратно в строку, но это не решало все проблемы. К примеру, в версии, предшествующей Scala 2.8, выражение `"abc" == "abc".reverse.reverse` вычислялось в `false`!

При использовании Scala 2.8 типом `cba` является `String`. Старое неявное преобразование в коллекцию Scala (которое теперь называется `WrappedString`) сохраняется, но предоставляет более конкретное преобразование из `String` в новый тип под названием `StringOps`. В данном типе имеется множество таких методов, как `reverse`, но вместо коллекции они возвращают значение типа `String`. Преобразование в `StringOps` определено непосредственно в `Predef`, а преобразование в коллекцию Scala — в новом классе `LowPriorityImplicits`, который расширяется `Predef`. Там, где есть выбор из этих двух преобразований, компилятор выбирает преобразование в `StringOps`, поскольку оно определено в подклассе того класса, где определено другое преобразование.

21.8. Отладка неявных преобразований

Неявные преобразования — довольно эффективный инструмент языка Scala, но иногда с ним трудно справиться должным образом. В этом разделе содержится ряд советов по отладке кода, использующего неявные элементы.

Временами можно удивиться, почему компилятор не находит неявноео преобразование, которое, на ваш взгляд, должен применить. В таком случае помогает явное указание преобразования. Если компилятор к тому же выдает сообщение об ошибке, то причина, по которой компилятор не может применить ваше неявное преобразование, известна.

Предположим, например, что `wrapString` была ошибочно выбрана для преобразования из `String`- в `List`-значения, а не в `IndexedSeq`-значения. Тогда может возникнуть вопрос, почему не работает следующий код:

```
scala> val chars: List[Char] = "xyz"
<console>:24: error: type mismatch;
 found   : String("xyz")
 required: List[Char]
       val chars: List[Char] = "xyz"
                               ^
```

И опять понять, что именно пошло не так, поможет явное указание преобразования `wrapString`:

```
scala> val chars: List[Char] = wrapString("xyz")
<console>:24: error: type mismatch;
 found : scala.collection.immutable.WrappedString
 required: List[Char]
       val chars: List[Char] = wrapString("xyz")
                                          ^
```

Его использование позволит выяснить причину ошибки: у `wrapString` неверный тип возвращаемого значения. Возможно также, что вставка явного указания

на преобразование приведет к устранению ошибки. В этом случае становится известно, что применению неявного преобразования помешало нарушение одного из дополнительных правил (правило области видимости).

Иногда при отладке программы вам может быть полезно видеть, какие именно неявные преобразования вставляет компилятор. Для этого пригодится ключ запуска компилятора `-Xprint:typer`. При запуске `scalac` с этим ключом компилятор покажет, как выглядит код после добавления механизмом проверки соответствия типов всех неявных преобразований. Пример показан в листингах 21.7 и 21.8. Если посмотреть на последнюю инструкцию каждого из этих листингов, то можно увидеть, что второй список параметров для `enjoy`, который был пропущен в листинге 21.7, `enjoy("reader")`, был вставлен компилятором, что и показано в листинге 21.8:

```
Mocha.this.enjoy("reader")(Mocha.this.pref)
```

Листинг 21.7. Пример кода, использующего неявный параметр

```
object Mocha extends App {

  class PreferredDrink(val preference: String)

  implicit val pref = new PreferredDrink("mocha")

  def enjoy(name: String)(implicit drink: PreferredDrink) = {
    print("Welcome, " + name)
    print(". Enjoy a ")
    print(drink.preference)
    println("!")
  }

  enjoy("reader")
}
```

Листинг 21.8. Пример кода после проверки соответствия типов и вставки неявных преобразований

```
$ scalac -Xprint:typer mocha.scala
[[syntax trees at end of typer]]
// Исходный код на Scala source: mocha.scala
package <empty> {
  final object Mocha extends java.lang.Object with Application
      with ScalaObject {
    // ...

    private[this] val pref: Mocha.PreferredDrink =
      new Mocha.this.PreferredDrink("mocha");
    implicit <stable> <accessor>
      def pref: Mocha.PreferredDrink = Mocha.this.pref;
    def enjoy(name: String)
        (implicit drink: Mocha.PreferredDrink): Unit = {
      scala.this.Predef.print("Welcome, ".+(name));
      scala.this.Predef.print(". Enjoy a ");
```

```
      scala.this.Predef.print(drink.preference);
      scala.this.Predef.println("!")
    };
    Mocha.this.enjoy("reader")(Mocha.this.pref)
  }
}
```

Можете отважиться и на попытку применить команду `scala -Xprint:typer`, чтобы получить интерактивную оболочку, которая выводит исходный код, используемый внутри интерпретатора, после набора текста.

Резюме

Неявные преобразования — довольно эффективный инструмент Scala, значительно сокращающий объем исходного кода. В этой главе мы рассмотрели действующие в Scala правила, которые относятся к неявным преобразованиям, и ряд наиболее распространенных в программировании ситуаций, когда, используя неявные преобразования, можно получить весомые преимущества.

В качестве предостережения следует заметить: неявные преобразования при слишком частом использовании могут запутать код. Поэтому, прежде чем добавлять новое неявное преобразование, задайтесь вопросом: нельзя ли получить аналогичный эффект с помощью других средств, таких как наследование, композиция примесей или перегрузка методов? Если ни одно из них не подойдет и у вас возникнет ощущение, что ваш код все еще загроможден и избыточен, то улучшить ситуацию сможет применение неявных преобразований.

22 Реализация списков

Списки в данной книге встречались повсеместно. Класс `List`, вероятно, — один из самых востребованных в Scala типов структурированных данных. Использование списков мы показали в главе 16. А в текущей главе мы сорвем покровы и объясним саму суть реализации списков в Scala.

Разбираться во внутренней работе класса `List` полезно по нескольким причинам. Так вы более четко представляете относительную эффективность операций со списками, и это помогает быстро создавать компактный код, использующий списки. Кроме того, освоите набор методов, который можно применить при разработке собственных библиотек. И наконец, класс `List` — это пример искусного применения имеющихся в Scala системы типов в целом и концепции универсального типа в частности. Следовательно, изучение класса `List` позволит углубить ваши знания в данных областях.

22.1. Принципиальный взгляд на класс List

Списки не встроены в Scala — они определяются с помощью абстрактного класса `List` из пакета `scala.collection.immutable` с двумя подклассами для `::` и `Nil`. В этой главе мы представим беглый обзор класса `List`. В данном разделе приведем несколько упрощенную версию класса по сравнению с его реальной реализацией в стандартной библиотеке Scala, рассматриваемой ниже, в разделе 22.3:

```
package scala.collection.immutable
sealed abstract class List[+A] {
```

Класс `List` — абстрактный, следовательно, определить элементы путем вызова пустого конструктора `List` невозможно. Например, выражение `new List` будет запрещенным. У класса есть параметр типа `A`. Знак `+` перед указанием этого параметра типа выступает признаком ковариантности списков, рассмотренным в главе 19.

Благодаря этому свойству переменной типа `List[Any]` можно присвоить значение типа `List[Int]`:

```
scala> val xs = List(1, 2, 3)
xs: List[Int] = List(1, 2, 3)

scala> var ys: List[Any] = xs
ys: List[Any] = List(1, 2, 3)
```

Все операции со списками можно определить в понятиях трех основных методов:

```
def isEmpty: Boolean
def head: A
def tail: List[A]
```

Все эти три метода являются абстрактными методами, принадлежащими классу `List`. Они определены в подчиненном объекте `Nil` и в подклассе `::`. Иерархия для `List` показана на рис. 22.1.

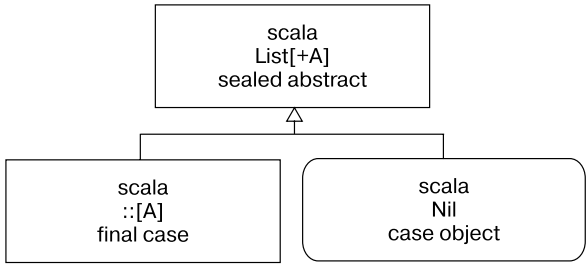


Рис. 22.1. Иерархия классов для списков Scala

Объект Nil

Объект `Nil` определяет пустой список. Его определение показано в листинге 22.1. Данный объект — наследник типа `List[Nothing]`. Благодаря ковариантности это означает, что `Nil` совместим с каждым экземпляром типа `List`.

Листинг 22.1. Определение объекта-одиночки Nil

```
case object Nil extends List[Nothing] {
  override def isEmpty = true
  def head: Nothing =
    throw new NoSuchElementException("head of empty list")
  def tail: List[Nothing] =
    throw new NoSuchElementException("tail of empty list")
}
```

В объекте `Nil` очень просто реализуются три абстрактных метода класса `List`: метод `isEmpty` возвращает `true`, а оба метода, `head` и `tail`, генерируют исключения.

Обратите внимание: генерация исключения — не только обоснованное, но и единственно возможное действие для метода `head`: поскольку `Nil` — список `List` из ничего, `Nothing`, то результирующим типом `head` должен быть `Nothing`. Значения подобного типа не существует, значит, метод `head` не может вернуть обычное значение. Ему приходится выполнять возвращение ненормальным образом — через генерацию исключения[1].

Класс ::

Класс `::` (произносится «конс», от слова construct — «конструировать») представляет непустые списки. Он назван так, чтобы поддержать сопоставление с образцом с инфиксом `::`. В разделе 16.5 было показано: каждая инфиксная операция в паттерне рассматривается как применение конструктора инфиксного оператора к его аргументам. Следовательно, схема `x :: xs` рассматривается как `::(x, xs)`, где `::` является `case`-классом.

Определение `case`-класса `::` выглядит так:

```
final case class ::[A](hd: A, tl: List[A]) extends List[A] {
  def head = hd
  def tail = tl
  override def isEmpty: Boolean = false
}
```

Реализация класса `::` весьма проста. Он получает два конструируемых параметра, `hd` и `tl`, представляющих «голову» и «хвост» создаваемого списка.

Определения методов `head` и `tail` просто возвращают соответствующий параметр. Фактически эту схему можно сократить за счет разрешения параметрам непосредственно реализовывать методы `head` и `tail` суперкласса `List`, как в следующем эквивалентном, но более лаконичном определении класса `::`:

```
final case class ::[A](head: A, tail: List[A])
    extends List[A] {

  override def isEmpty: Boolean = false
}
```

Данный код работает благодаря тому, что каждый параметр `case`-класса является еще и полем класса (это похоже на объявление параметра с префиксом `val`). Вспомним, в разделе 20.3 мы говорили: Scala позволяет выполнять реализацию абстрактных методов, не имеющих параметров, таких как `head` или `tail`, с помощью поля. Следовательно, показанный ранее код напрямую использует параметры `head` и `tail` в качестве реализаций абстрактных методов `head` и `tail`, унаследованных из класса `List`.

---

[1] Точнее говоря, типы также позволяли бы методу head всегда вместо генерации исключения входить в бесконечный цикл, но это явно не то, что нужно.

Еще несколько методов

Все остальные методы класса `List` могут быть написаны с использованием трех основных методов, например:

```
def length: Int =
  if (isEmpty) 0 else 1 + tail.length
```

или

```
def drop(n: Int): List[A] =
  if (isEmpty) Nil
  else if (n <= 0) this
  else tail.drop(n - 1)
```

или

```
def map[B](f: A => B): List[B] =
  if (isEmpty) Nil
  else f(head) :: tail.map(f)
```

Создание списка

Методы создания списка `::` и `:::` имеют особое свойство. Их имена заканчиваются двоеточием, поэтому они привязаны к своему правому операнду. То есть такая операция, как `x :: xs`, рассматривается как вызов метода `xs.::(x)`, а не как вызов метода `x.::(xs)`. Фактически выражение `x.::(xs)` не имело бы никакого смысла, поскольку `x` относится к типу элементов списка, который может быть каким угодно, поэтому предположить, что в данном типе будет реализован метод `::`, невозможно.

Поэтому методу `::` нужно получить значение элемента и выдать новый список. Каков же обязательный тип значения элемента? Возможно, вам захочется сказать в ответ, что он должен быть таким же, как и тип элементов списка, однако на самом деле он имеет более ограниченную природу, чем требуется.

Чтобы понять причину этого, рассмотрим иерархию следующего класса:

```
abstract class Fruit
class Apple extends Fruit
class Orange extends Fruit
```

В листинге 22.2 показано, что произойдет при создании списка фруктов (`fruits`).

Листинг 22.2. Добавление элемента супертипа в список подтипов

```
scala> val apples = new Apple :: Nil
apples: List[Apple] = List(Apple@e885c6a)

scala> val fruits = new Orange :: apples
fruits: List[Fruit] = List(Orange@3f51b349, Apple@e885c6a)
```

Значение `apples` (яблоки) согласно ожиданиям рассматривается как список, состоящий из элементов типа `Apple`. Но определение фруктов показывает, что сохраняется возможность добавить к этому списку элемент другого типа. Типом элемента получающегося списка объявляется `Fruit` — наиболее точный общий супертип для типа элементов исходного списка, то есть `Apple`, и типа добавляемого элемента, то есть `Orange`. Такая гибкость достигается за счет определения метода `::` (cons), показанного в листинге 22.3.

Листинг 22.3. Определение метода :: (cons) в классе List

```
def ::[B >: A](x: B): List[B] = new scala.::(x, this)
```

Обратите внимание: сам по себе этот метод полиморфен, поскольку получает параметр типа по имени `B`. Кроме того, на `B` наложены ограничения в выражении `[B >: A]` и он должен быть супертипом того типа `A`, к которому относятся элементы списка. Требуется, чтобы добавляемый элемент относился к типу `B`, а получаемый результат — к типу `List[B]`.

С помощью формулировки `::`, показанной в листинге 22.3, можно проверить, как определение `fruits`, представленное в листинге 22.2, срабатывает в зависимости от типа: в этом определении параметр типа `B` класса `::` конкретизируется в `Fruit`. Условие по нижнему ограничителю `B` удовлетворено, поскольку список `apples` относится к типу `List[Apple]`, а `Fruit` является супертипом для типа `Apple`. Аргументом метода `::` выступает `new Orange`, который соответствует типу `Fruit`. Поэтому применение метода не приводит к нарушению соответствия типов, а его результирующим типом является `List[Fruit]`. На рис. 22.2 показана структура списков, получающихся в результате выполнения кода из листинга 22.2.

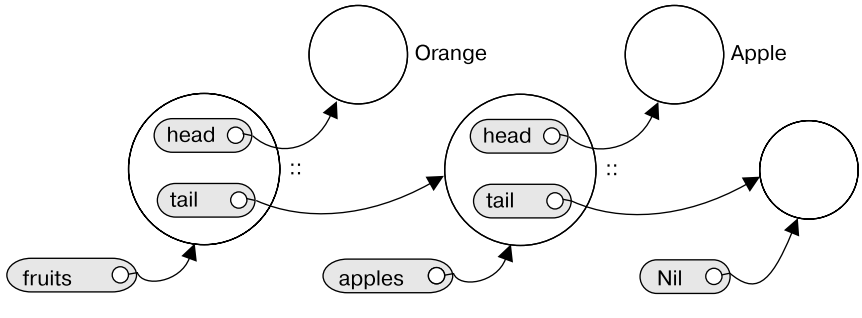


Рис. 22.2. Структура списков Scala, показанных в листинге 22.2

Фактически полиморфное определение метода `::` с нижним ограничителем `A` не только приносит определенные удобства, но и необходимо для представления определения класса `List` в корректном в отношении типов виде. Это происходит благодаря тому, что списки типа `List` определены как ковариантные.

Предположим на минуту, что у метода `::` есть следующее определение:

```
// Мысленный эксперимент (в неработоспособном виде)
def ::(x: A): List[A] = new scala.::(x, this)
```

В главе 19 мы показали, что параметры метода считаются контравариантной позицией, следовательно, элементы списка типа `A` находятся в контравариантной позиции в определении выше. Но тогда класс `List` не может быть объявлен ковариантным по `A`. Таким образом, нижний ограничитель `[B >: A]` убивает одним выстрелом двух зайцев: устраняет проблему несоответствия типов и приводит к повышению гибкости использования метода `::`. Как показано в листинге 22.4, метод конкатенации списков `:::` определен по аналогии с определением метода `::`.

Листинг 22.4. Определение метода ::: в классе List

```
def :::[B >: A](prefix: List[B]): List[B] =
  if (prefix.isEmpty) this
  else prefix.head :: prefix.tail ::: this
```

Как и метод `cons`, метод конкатенации полиморфен. Результирующий тип расширяется по мере необходимости в целях включения типов всех элементов списка. Нужно еще раз отметить, что порядок следования аргументов между инфиксной операцией и явным вызовом метода меняется на противоположный. Имена методов `:::` и `::` заканчиваются двоеточием, поэтому оба они привязаны к правому аргументу и обладают правой ассоциативностью. Например, в части `else` определения метода `:::`, показанного выше в листинге 22.4, содержатся инфиксные операции как для оператора `::`, так для и оператора `:::`.

Эти инфиксные операции можно расширить в следующие эквивалентные вызовы методов:

```
prefix.head :: prefix.tail ::: this
  равно (поскольку :: и ::: обладают правой ассоциативностью)
prefix.head :: (prefix.tail ::: this)
  равно (поскольку :: привязывается к правому операнду)
(prefix.tail ::: this).::(prefix.head)
  равно (поскольку ::: привязывается к правому операнду)
this.:::(prefix.tail).::(prefix.head)
```

22.2. Класс ListBuffer

Обычный паттерн доступа к списку является рекурсией. Например, чтобы увеличить на единицу значение каждого элемента списка, не прибегая к методу `map`, можно воспользоваться следующим кодом:

```
def incAll(xs: List[Int]): List[Int] = xs match {
  case List() => List()
  case x :: xs1 => x + 1 :: incAll(xs1)
}
```

У этого программного паттерна есть один недостаток: он не является хвостовой рекурсией. Следует заметить, что показанный ранее рекурсивный вызов `incAll` происходит внутри операции `::`. Поэтому каждый рекурсивный вызов требует нового стекового фрейма.

На современных виртуальных машинах это означает, что применять `incAll` к спискам, количество элементов которых существенно превышает диапазон от 30 000 до 50 000, невозможно. А жаль. А как бы вы написали версию `incAll`, способную работать со списками произвольного размера (насколько это позволил бы объем динамической области памяти)?

Одним из подходов может стать применение цикла:

```
for (x <- xs) // ??
```

Но что должно быть помещено в тело цикла? Ведь `incAll` создает список, помещая элементы перед результатом рекурсивного вызова, а циклу нужно добавлять новые элементы в конец итогового списка. Возможность использования оператора добавления к списку `:::` представляется слишком неэффективной:

```
var result = List[Int]() // весьма неэффективный подход
for (x <- xs) result = result ::: List(x + 1)
result
```

Это абсолютно неэффективный код. Поскольку на выполнение операции `:::` затрачивается время, пропорциональное длине ее первого операнда, то на всю операцию уйдет время, пропорциональное квадрату длины списка. А это совершенно неприемлемо.

Более удачной альтернативой будет использование буфера списка, позволяющего аккумулировать элементы списка. Для этого нужно применить такую операцию, как `buf += elem`, добавляющую элемент `elem` в конец буфера списка `buf`. Закончив добавлять элементы буфера списка, можно превратить буфер в обычный список, воспользовавшись операцией `toList`.

`ListBuffer` — класс, который находится в пакете `scala.collection.mutable`. Чтобы пользоваться только простым именем, можно импортировать `ListBuffer` из его пакета:

```
import scala.collection.mutable.ListBuffer
```

Теперь с помощью буфера списка тело `incAll` можно записать так:

```
val buf = new ListBuffer[Int]
for (x <- xs) buf += x + 1
buf.toList
```

Это весьма эффективный способ создания списков. Фактически реализация буфера списка организована таким образом, чтобы на обе операции: добавления (`+=`) и превращения в список (`toList`) — уходило постоянное, весьма незначительное время.

22.3. Класс List на практике

Реализация методов работы со списками, рассмотренная в разделе 22.1, имеет весьма лаконичный и понятный код, но страдает от проблемы переполнения так же, как и реализация `incAll` без хвостовой рекурсии. Поэтому в большинстве методов в реальной реализации класса `List` используется не рекурсия, а буферы списков. Так, в листинге 22.5 показана реальная реализация метода `map` класса `List`.

Листинг 22.5. Определение метода map класса List

```
final override def map[B](f: A => B): List[B] = {
  if (this eq Nil) Nil else {
    val h = new ::[B](f(head), Nil)
    var t: ::[B] = h
    var rest = tail
    while (rest ne Nil) {
      val nx = new ::(f(rest.head), Nil)
      t.next = nx
      t = nx
      rest = rest.tail
    }
    h
  }
}
```

Эта пересмотренная реализация проходит по элементам списка с помощью простого, но эффективного цикла. Не менее эффективной будет и реализация с использованием хвостовой рекурсии, но общая рекурсивная реализация станет работать медленнее. Обратите внимание, как список выстраивается «слева направо» (в отличие от применения обычного конс-оператора, который добавляет элементы с левой стороны списка) с помощью строки кода:

```
t.next = nx
```

Данная строка *изменяет* конец списка. Чтобы понять, как все это работает, еще раз взглянем на класс `::`, создающий непустые списки. На практике он не вполне соответствует своему идеализированному определению, которое было дано в разделе 22.1. Реальное определение больше похоже на то, которое показано в листинге 22.6. Как видите, он имеет еще одну особенность: аргумент `next` является `var`-переменной! Это значит, «хвост» после создания списка можно изменять. Но поскольку у переменной `next` имеется модификатор `private[scala]`, то к ней можно обращаться только из пакета `scala`. Клиентский код, находящийся за пределами данного пакета, не сможет читать переменную `next` и записывать в нее.

Листинг 22.6. Определение подкласса :: класса List

```
final case class :: [A](override val head: A,
    private[scala] var next: List[A]) extends List[A] {

  def isEmpty: Boolean = false
  def tail: List[A] = next
}
```

В примере, ввиду того что класс `ListBuffer` содержится в подпакете `scala.collection.mutable` пакета `scala`, `ListBuffer` может обращаться к полю `next` ячейки `cons`. Фактически элементы буфера списка представлены в виде списка, и добавление новых элементов включает изменение поля `next` последней ячейки `::` в этом списке. Таким образом, `ListBuffer` — еще один эффективный способ построить список слева направо. Начало кода класса `ListBuffer` выглядит так:

```
package scala.collection.mutable
class ListBuffer[A] extends Buffer[A] {
  private var first: List[A] = Nil
  private var last0: ::[A] = null
  private var aliased: Boolean = false

  ...
```

Здесь видны три приватных поля, характеризующих `ListBuffer`:

- `first` указывает на список всех элементов, хранящихся в буфере списка;
- `last0` указывает на последнюю ячейку `::` в этом буфере списка;
- `aliased` показывает, был ли превращен буфер списка в список с помощью `toList`.

Операция `toList` очень проста:

```
override def toList: List[A] = {
  aliased = nonEmpty
  first
}
```

Она возвращает список элементов, на который ссылается `first`, а также присваивает переменной `aliased` значение `true`, если список не пуст. Следовательно, операция `toList` весьма эффективна, поскольку не копирует список, сохраненный в `ListBuffer`. А что произойдет, если список после выполнения операции `toList` будет дополнен еще раз? Разумеется, список, возвращенный из `toList`, должен быть неизменяемым. Но добавление к элементу `last0` приведет к изменению списка, на который ссылается `first`.

Чтобы поддержать корректность операций с буфером списка, нужно вместо этого работать со свежим списком. Данная задача в реализации операции `addOne` решается в первой же строке:

```
def addOne(elem: A): this.type = {
  if (aliased) copyElems()
  val last1 = new ::[A](elem, Nil)
  if (first.isEmpty) first = last1 else last0.next = last1
  last0 = last1
  this
}
```

Как видите, операция `addOne` копирует список, на который указывает `first`, если переменная `aliased` имеет значение `true`. Следовательно, в итоге без издержек не обходится. Если понадобится перейти от списков, допускающих расширение,

к неизменяемым спискам, то без копирования ничего не получится. Но реализация `ListBuffer` выполнена таким образом, что копирование необходимо только для буферов списков, расширение которых продолжилось после того, как они превратились в списки. На практике подобное случается очень редко. В большинстве случаев использования буферов списков элементы добавляются постепенно, а затем в конце выполняется всего одна операция `toList`. Тогда в копировании нет необходимости.

22.4. Внешняя функциональность

В предыдущем разделе мы рассмотрели основные элементы реализации в Scala классов `List` и `ListBuffer`. Мы показали, что снаружи списки имеют чисто функциональную природу, но внутри у них есть императивная реализация, которая использует буферы списков. Это обычная стратегия в программировании на Scala, заключающаяся в стремлении добиться сочетания чистоты с эффективностью путем тщательного ограничения последствий нечистых операций.

У вас может возникнуть вопрос: зачем так рьяно настаивать на чистоте? Почему бы просто не открыть определение списков, сделав изменяемым поле `tail`, а может быть, и поле `head`? Недостаток подобного подхода заключается в том, что программы станут значительно менее надежными. Обратите внимание: при создании списков с помощью оператора `::` многократно используется «хвост» создаваемого списка.

Поэтому при написании кода:

```
val ys = 1 :: xs
val zs = 2 :: xs
```

«хвосты» списков `ys` и `zs` являются общими — они указывают на одну и ту же структуру данных. Это важно с точки зрения эффективности. Если бы список `xs` копировался всякий раз, когда к нему добавляется новый элемент, то работа выполнялась бы намного медленнее. Поскольку совместное использование получило весьма широкое распространение, изменение элементов списка, будь оно возможным, было бы крайне опасной операцией. К примеру, если путем написания следующего кода в показанном ранее фрагменте понадобится выполнить усечение списка `ys` до его первых двух элементов:

```
ys.drop(2).tail = Nil // Сделать это в Scala нельзя!
```

то в качестве побочного эффекта будут усечены также списки `zs` и `xs`.

Совершенно очевидно, что отслеживать места изменений крайне трудно. Именно поэтому для списков в Scala выбраны их широкомасштабное совместное применение и отказ от изменений. Если пожелаете, то класс `ListBuffer` по-прежнему позволяет создавать списки постепенно, в императивном стиле. Но поскольку буферы списков не являются списками, то типы разделяют изменяемые буферы списков и неизменяемые списки.

В дизайне классов `List` и `ListBuffer` в Scala происходят действия, очень похожие на те, которые выполняются в принадлежащей Java паре классов `String` и `StringBuffer`. Это не случайно. В обеих ситуациях разработчики стремились создать чистые неизменяемые структуры данных, но при этом представить эффективный способ постепенного создания таких структур. Для строк Java и Scala методы `StringBuffer` (или в Java 5 — `StringBuilder`) предоставляют способ постепенного создания строки. В списках Scala у вас есть выбор: можно либо создавать списки постепенно, добавляя элементы к началу списка с помощью операции `::`, либо задействовать буфер списка для добавления элементов к его концу. Какому из них отдать предпочтение, зависит от конкретной ситуации. Обычно операция `::` хорошо поддается рекурсивным алгоритмам в стиле «разделяй и властвуй». Буферы списков часто используются в более традиционном стиле, основанном на применении циклов.

Резюме

В этой главе мы показали реализацию списков в Scala. Список — наиболее востребованная структура данных в Scala, имеющая почти совершенную реализацию. Оба подкласса класса `List`, `Nil` и `::`, являются `case`-классами. Но вместо рекурсивной обработки данной структуры многие основные методы работы со списками задействуют `ListBuffer`. В свою очередь, класс `ListBuffer` реализован настолько аккуратно, что может эффективно создавать списки, не выделяя для своей работы дополнительную память. Снаружи он функциональный, но внутри использует изменяемость для ускорения обычной работы, когда буфер списка сбрасывается после вызова метода `toList`. Теперь, после изучения всех этих особенностей, классы списков вам знакомы как снаружи, так и изнутри, а кроме этого, вы освоили пару хитрых приемов их реализации.

23 Возвращение к выражениям for

В главе 16 мы показали, что функции высшего порядка, такие как `map`, `flatMap` и `filter`, предоставляют весьма эффективные конструкции для работы со списками. Но иногда уровень абстракции, требующийся этим функциям, усложняет понимание кода программы.

Рассмотрим пример. Предположим, имеется список людей, каждый элемент которого определен как экземпляр класса `Person`. У класса `Person` есть поля, показывающие имя человека, его пол: мужской или женский — и его детей.

Определение класса имеет следующий вид:

```
scala> case class Person(name: String,
                         isMale: Boolean,
                         children: Person * )
```

А вот как выглядит список людей, взятых для примера:

```
val lara = Person("Lara", false)
val bob = Person("Bob", true)
val julie = Person("Julie", false, lara, bob)
val persons = List(lara, bob, julie)
```

Теперь предположим, что в списке нужно найти имена всех пар матерей и их детей. Используя `map`, `flatMap` и `filter`, можно сформулировать следующий запрос:

```
scala> persons filter (p => !p.isMale) flatMap (p =>
           (p.children map (c => (p.name, c.name))))
res0: List[(String, String)] = List((Julie,Lara),
    (Julie,Bob))
```

Данный пример можно слегка оптимизировать, используя вместо `filter` вызов метода `withFilter`. Это позволит избавиться от создания промежуточной структуры данных для людей женского пола:

```
scala> persons withFilter (p => !p.isMale) flatMap (p =>
           (p.children map (c => (p.name, c.name))))
res1: List[(String, String)] = List((Julie,Lara),
    (Julie,Bob))
```

Несмотря на то что эти запросы со своей задачей успешно справляются, для написания и понимания они не так уж просты. А есть ли более простой способ? Да.

Помните выражения `for` из раздела 7.3? Используя выражение `for`, тот же самый пример можно записать следующим образом:

```
scala> for (p <- persons; if !p.isMale; c <- p.children)
       yield (p.name, c.name)
res2: List[(String, String)] = List((Julie,Lara),
    (Julie,Bob))
```

Результат выполнения этого выражения будет точно таким же, как и результат выполнения предыдущего. Более того, многим читателям кода выражение `for` будет понятнее предыдущего запроса, в котором использовались функции высшего порядка `map`, `flatMap` и `withFilter`.

Но может показаться, что последние два запроса отличаются друг от друга не слишком сильно. Фактически окажется, что компилятор Scala переведет второй запрос в первый. В более широком смысле компилятор переводит все выражения, выдающие результат с помощью инструкции `yield`, в сочетание вызовов функций высшего порядка `map`, `flatMap` и `withFilter`. Все циклы `for` без инструкции `yield` переводятся в меньший по объему набор функций высшего порядка, в котором присутствуют только функции `withFilter` и `foreach`.

Сначала в данной главе мы рассмотрим точные правила написания выражений `for`. Затем покажем, как с их помощью упростить решение комбинаторных задач. И наконец, вы узнаете, как выполняется перевод выражений `for` и как в результате этого выражения `for` могут способствовать развитию языка Scala в новых областях применения.

23.1. Выражения for

В общем выражение `for` выглядит так:

```
for ( seq ) yield expr
```

где *`seq`* — последовательность из *генераторов*, *определений* и *фильтров* с точками с запятой между стоящими друг за другом элементами. Выражение `for` показано в ниже:

```
for (p <- persons; n = p.name; if (n startsWith "To"))
yield n
```

Это выражение `for` содержит один генератор, одно определение и один фильтр. Как упоминалось в разделе 7.3, последовательность можно заключить вместо круглых скобок в фигурные. Тогда точки с запятой можно будет не ставить:

```
for {
  p <- persons           // генератор
  n = p.name             // определение
  if (n startsWith "To") // фильтр
} yield n
```

Генератор имеет следующую форму:

```
pat <- expr
```

Выражение *expr* обычно возвращает список, хотя чуть позже мы покажем, что это обстоятельство можно обобщить. Образец *pat* сопоставляется по порядку со всеми элементами списка. При успешной проверке переменные в образце привязываются к соответствующим частям элемента, как описано в главе 15. Но если проверка завершается неудачно, то никакая ошибка `MatchError` не выдается. Вместо этого элемент просто выбрасывается из итерации.

В наиболее распространенном варианте применения образец *pat* представлен простой переменной *x*, как в выражении *x <- expr*. В данном случае переменная *x* просто используется при последовательном переборе всех элементов, возвращаемых выражением *expr*.

Определение имеет форму:

```
pat = expr
```

Оно привязывает образец *pat* к переменной выражения *expr*. Таким образом, дает такой же эффект, что и определение `val`-переменной:

```
val x = expr
```

В наиболее распространенном варианте использования образец также представлен простой переменной *x* (например, *x = expr*). Это определяет *x* как имя для значения *expr*.

Фильтр имеет форму:

```
if expr
```

Здесь *expr* является выражением, имеющим тип `Boolean`. Фильтр выбрасывает из итерации все элементы, для которых выражение *expr* возвращает `false`.

Каждое выражение `for` начинается с генератора. Если в выражении имеется сразу несколько генераторов, то последующие генераторы изменяются быстрее предыдущих. Это легко проверить с помощью следующего простого теста:

```
scala> for (x <- List(1, 2); y <- List("one", "two"))
       yield (x, y)
res3: List[(Int, String)] =
  List((1,one), (1,two), (2,one), (2,two))
```

23.2. Задача N ферзей

Выражениям `for` в качестве подходящей области применения лучше всего подходят комбинаторные головоломки. Пример такой головоломки — задача восьми ферзей: на стандартной шахматной доске нужно расставить восемь ферзей таким образом, чтобы ни один из них не мог бить любого другого (ферзь может бить другую фигуру, если они находятся на одной вертикали, горизонтали или диагонали). Чтобы найти решение этой задачи, ее проще рассматривать в общем виде на шахматной доске произвольного размера. Тогда задача будет заключаться в размещении N ферзей на доске из $N \times N$ клеток, где N имеет произвольное значение. Начнем нумеровать клетки с единицы, чтобы верхняя левая клетка доски размером $N \times N$ имела координаты (1, 1), а нижняя правая — (N, N).

Обратите внимание: для решения задачи N ферзей нужно поставить по ферзю на каждую горизонталь. Таким образом, можно последовательно расставлять ферзей по горизонталям, каждый раз проверяя, не находится ли только что поставленная на доску фигура под ударом любых других ранее поставленных ферзей. В ходе проверки может оказаться, что ферзю, которого нужно поставить на горизонталь k, будут грозить на всех полях этой горизонтали ферзи, установленные на горизонталях от 1 до $k - 1$. В таком случае следует прекратить данную часть проверки, чтобы продолжить с другой расстановкой ферзей на вертикалях от 1 до $k - 1$.

При императивном решении этой задачи ферзей будут ставить по одному, перемещая их по доске. Но, похоже, найти схему, которая реально перепробует все возможности, нелегко. Более функциональный подход представляет собой решение непосредственно в виде значения. Решение состоит из списка координат, по одному для каждого поставленного на доску ферзя. Но следует заметить, что найти за один шаг полноценное решение не удастся. Оно должно выстраиваться постепенно, по мере расстановки ферзей по последовательным горизонталям.

Это наводит на мысль о применении рекурсивного алгоритма. Предположим, что уже сгенерированы все решения по размещению k ферзей на доске размером $N \times N$ клеток, где $k < N$. Каждое такое решение можно представить в виде списка длиной k координат строк и столбцов (`row`, `column`), где и `row`, и `column` — числа в диапазоне от 1 до N. Эти списки частных решений удобнее будет рассматривать в виде стеков, где координаты ферзя в строке k следуют первыми в списке, за ними идут координаты ферзя в строке $k - 1$ и т. д. На дне стека находятся координаты ферзя, помещенного на первую строку доски. Решение целиком представлено в виде списка списков, по одному элементу для каждого решения.

Теперь, чтобы поставить следующего ферзя на строку $k + 1$, генерируются все возможные расширения каждого предыдущего решения еще на одного ферзя. Это приводит к появлению еще одного списка, который состоит из списков решений, на этот раз длиной $k + 1$. Процесс продолжается до момента получения всех решений для шахматной доски размером $N \times N$.

Этот алгоритмический замысел воплотился в функции `placeQueens`:

```
def queens(n: Int): List[List[(Int, Int)]] = {
  def placeQueens(k: Int): List[List[(Int, Int)]] =
    if (k == 0)
      List(List())
    else
      for {
        queens <- placeQueens(k - 1)
        column <- 1 to n
        queen = (k, column)
        if isSafe(queen, queens)
     } yield queen :: queens
  placeQueens(n)
}
```

Внешняя функция `queens` в показанной ранее программе просто вызывает `placeQueens` с размером доски `n`, используемым в качестве аргумента. Задача функции-приложения `placeQueens(k)` — генерирование в списке всех частных решений

длиной k. Каждый элемент списка — одно решение, представленное списком длиной k. Следовательно, placeQueens возвращает список списков.

Если параметр k для placeQueens равен нулю, то это значит, что нужно сгенерировать все решения, помещая нуль ферзей на нуль строк. Есть только одно такое решение: не ставить ни одного ферзя. Оно представлено пустым списком. Следовательно, если параметр k равен нулю, то placeQueens возвращает List(List()) — список, который состоит из одного элемента, являющегося пустым списком. Обратите внимание: это весьма отличается от пустого списка List(). Если placeQueens возвращает List(), то это означает *отсутствие решений*, а не единственное решение, состоящее из не поставленного на доску ферзя.

В иных случаях, когда $k \neq 0$, вся работа placeQueens выполняется в выражении for. Первый генератор этого выражения for обходит все решения по расстановке на доске k - 1 ферзей. Второй генератор обходит все возможные столбцы column, на которые можно поставить k-го ферзя. Третья часть выражения for определяет позицию нового ферзя как пару, состоящую из строки k и каждого сгенерированного столбца column. Четвертая часть выражения for является фильтром, проверяющим с помощью метода isSafe, не находится ли новый ферзь под ударом предыдущих ферзей (определение isSafe будет рассмотрено чуть позже).

Если новый ферзь не находится под ударом любых других ферзей, то может быть сформирована часть частного решения, стало быть, placeQueens с помощью выражения queen :: queens генерирует новое решение. Если новый ферзь не находится в безопасной от удара позиции, то фильтр возвращает false и решение не генерируется.

Осталось только определить метод isSafe, используемый для проверки того, не находится ли рассматриваемый ферзь под ударом от любого другого элемента списка queens. Определение имеет следующий вид:

```
def isSafe(queen: (Int, Int), queens: List[(Int, Int)]) =
  queens forall (q => !inCheck(queen, q))

def inCheck(q1: (Int, Int), q2: (Int, Int)) =
  q1._1 == q2._1 || // та же строка
  q1._2 == q2._2 || // тот же столбец
  (q1._1 - q2._1).abs == (q1._2 - q2._2).abs // по диагонали
```

Метод isSafe выражает безопасность ферзя относительно какой-либо из других ферзей. Метод inCheck показывает, что ферзи q1 и q2 находятся под ударом друг друга.

Он возвращает true в одном из трех случаев.

1. Если два ферзя имеют одну и ту же координату строки.
2. Если два ферзя имеют одну и ту же координату столбца.
3. Если два ферзя стоят на одной и той же диагонали (то есть разница между их строками и разница между их столбцами одинакова).

Первый случай, когда два ферзя имеют одинаковую координату строки, в данном приложении невозможен, поскольку метод placeQueens уже позаботился о помещении каждого ферзя на другую строку. Следовательно, эту проверку можно убрать, не изменяя функциональности программы.

23.3. Выполнение запросов с помощью выражений for

Форма записи выражения `for`, по сути, эквивалентна общим операциям языков запросов к базам данных. Возьмем, к примеру, некую базу данных по имени `books`, представляющую собой список книг, в котором класс `Book` определен так:

```
case class Book(title: String, authors: String* )
```

Вот как выглядит небольшой фрагмент базы данных, представленный в виде списка в памяти:

```
val books: List[Book] =
  List(
    Book(
      "Structure and Interpretation of Computer Programs",
      "Abelson, Harold", "Sussman, Gerald J."
    ),
    Book(
      "Principles of Compiler Design",
      "Aho, Alfred", "Ullman, Jeffrey"
    ),
    Book(
      "Programming in Modula-2",
      "Wirth, Niklaus"
    ),
    Book(
      "Elements of ML Programming",
      "Ullman, Jeffrey"
    ),
    Book(
      "The Java Language Specification", "Gosling, James",
      "Joy, Bill", "Steele, Guy", "Bracha, Gilad"
    )
 )
```

Чтобы найти названия всех книг, автор которых носит фамилию Gosling, требуется следующий код:

```
scala> for (b <- books; a <- b.authors
            if a startsWith "Gosling")
       yield b.title
res4: List[String] = List(The Java Language Specification)
```

А чтобы найти названия всех книг, в которых встречается строка `Program`, требуется такой код:

```
scala> for (b <- books if (b.title indexOf "Program") >= 0)
       yield b.title
res5: List[String] = List(Structure and Interpretation of
  Computer Programs, Programming in Modula-2, Elements of ML
    Programming)
```

Чтобы в базе данных найти фамилии всех авторов, написавших хотя бы две книги, требуется следующий код:

```
scala> for (b1 <- books; b2 <- books if b1 != b2;
            a1 <- b1.authors; a2 <- b2.authors if a1 == a2)
       yield a1
res6: List[String] = List(Ullman, Jeffrey, Ullman, Jeffrey)
```

Последнее решение все же далеко от совершенства, поскольку в полученном в результате списке авторы будут фигурировать по нескольку раз. Нужно удалить повторяющихся авторов из списка. Сделать это позволяет функция, показанная ниже:

```
scala> def removeDuplicates[A](xs: List[A]): List[A] = {
         if (xs.isEmpty) xs
         else
           xs.head :: removeDuplicates(
             xs.tail filter (x => x != xs.head)
           )
      }
removeDuplicates: [A](xs: List[A])List[A]

scala> removeDuplicates(res6)
res7: List[String] = List(Ullman, Jeffrey)
```

Стоит отметить, что последнее выражение в методе `removeDuplicates` можно выразить эквивалентно с помощью `for`:

```
xs.head :: removeDuplicates(
  for (x <- xs.tail if x != xs.head) yield x
)
```

23.4. Трансляция выражений for

Каждое выражение `for` можно выразить в понятиях трех функций высшего порядка: `map`, `flatMap` и `withFilter`. В этом разделе рассматривается схема трансляции, которая также используется компилятором Scala.

Трансляция выражений for с одним генератором

Предположим сначала, что у нас есть простое выражение `for`:

`for ( x <-` $expr_1$ `) yield` $expr_2$

где `x` — переменная. Выражение транслируется в следующее выражение:

$expr_1$`.map( x => `$expr_2$ `)`

Трансляция выражений for, начинающихся с генератора и фильтра

Теперь рассмотрим выражения `for`, сочетающие в себе ведущий генератор с некоторыми другими элементами. Выражение `for` следующей формы:

```
for ( x <- expr1 if expr2 ) yield expr3
```

транслируется в выражение:

```
for ( x <- expr1 withFilter ( x => expr2 )) yield expr3
```

Эта трансляция выдает еще одно выражение `for`, которое короче исходного на один элемент, поскольку элемент `if` транслируется в применение `withFilter` к первому выражению генератора. Затем трансляция продолжается в отношении второго выражения, и в итоге получается такой код:

```
expr1 withFilter ( x => expr2 ) map ( x => expr3 )
```

Точно такая же схема применима, если есть другие элементы, следующие за фильтром. Если `seq` является произвольной последовательностью генераторов, определений и фильтров, то выражение:

```
for ( x <- expr1 if expr2; seq) yield expr3
```

транслируется в выражение:

```
for ( x <- expr1 withFilter expr2; seq) yield expr3
```

Затем процесс трансляции продолжается в отношении второго выражения, которое снова короче исходного на один элемент.

Трансляция выражений for, начинающихся с двух генераторов

Следующий случай относится к обработке выражений `for`, начинающихся с двух генераторов, как в:

```
for ( x <- expr1; y <- expr2; seq) yield expr3
```

Здесь также предположим, что *seq* — произвольная последовательность генераторов, определений и фильтров. Фактически элемент *seq* также может быть пустым, и тогда после $expr_2$ точки с запятой может не быть. Схема трансляции в любом случае будет одинаковой. Показанное ранее выражение `for` транслируется в выражение с применением метода `flatMap`:

```
expr1.flatMap( x => for ( y <- expr2; seq) yield expr3 )
```

На сей раз получается еще одно выражение `for`, которое находится в переданном методу `flatMap` функциональном значении. Это выражение `for`, которое опять проще исходного на один элемент, транслируется с применением тех же самых правил.

Трех показанных схем достаточно для трансляции всех выражений `for`, содержащих лишь генераторы и фильтры, где генераторы привязывают только простые переменные. Возьмем, к примеру, запрос из раздела 23.3 «найти фамилии всех авторов, написавших хотя бы две книги»:

```
for (b1 <- books; b2 <- books if b1 != b2;
     a1 <- b1.authors; a2 <- b2.authors if a1 == a2)
yield a1
```

Этот запрос транслируется в следующую комбинацию `map`/`flatMap`/`withFilter`:

```
books flatMap (b1 =>
  books withFilter (b2 => b1 != b2) flatMap (b2 =>
    b1.authors flatMap (a1 =>
      b2.authors withFilter (a2 => a1 == a2) map (a2 =>
        a1))))
```

Представленная до сих пор схема трансляции пока не справляется с генераторами, привязывающими не простые переменные, а целые паттерны. Она также пока не охватывает определения. Эти два аспекта будут рассмотрены в следующих двух подразделах.

Трансляция паттернов в генераторах

Если в левой части генератора находится не простая переменная, а паттерн *pat*, то схема трансляции усложняется. Тот случай, когда в выражении `for` имеется привязка кортежа переменных, обрабатывается довольно просто. При этом применяется почти такая же схема, как и при одной переменной.

Выражение `for`, имеющее следующую форму:

```
for (( x1 , ..., xn ) <- expr1 ) yield expr2
```

транслируется в выражение:

```
expr1.map { case ( x1 , ..., xn ) => expr2 }
```

Ситуация усложняется, если в левой части генератора находится не одна переменная и не кортеж переменных, а произвольный паттерн *pat*. В таком случае выражение:

```
for (pat <- expr1 ) yield expr2
```

транслируется в выражение:

```
expr1 withFilter {
  case pat => true
  case _ => false
} map {
  case pat => expr2
}
```

То есть сначала генерируемые элементы фильтруются, а затем отображаются только те из них, которые соответствуют *pat*. Тем самым гарантируется, что генератор сопоставления с образцом никогда не выдаст ошибку `MatchError`.

В представленной здесь схеме рассматриваются только случаи, когда выражение `for` содержит лишь один генератор сопоставления с образцом. Аналогичные правила применяются и при наличии в выражении `for` других генераторов, фильтров или определений. Поскольку эти дополнительные правила не помогают лучше понять тему, здесь они не рассматриваются. Любопытные читатели могут найти их в Scala Language Specification[1].

Трансляция определений

И последняя недостающая ситуация — когда в выражении `for` содержатся встроенные определения. Рассмотрим типичный вариант:

```
for (x <- expr1 ; y = expr2; seq) yield expr3
```

Предположим в очередной раз, что элемент *seq* (возможно, пустой) является последовательностью генераторов, определений и фильтров. Это выражение транслируется в следующее выражение:

```
for ((x, y) <- for (x <- expr1 ) yield (x, expr2); seq)
yield expr3
```

Как видите, $expr_2$ вычисляется при каждой генерации нового значения *x*. Необходимость проводить многократное вычисление обусловливается тем, что $expr_2$ может ссылаться на *x* и поэтому при изменении значения *x* должно быть вычислено заново. Вы как программист можете прийти к заключению, что встраивать определения в выражения `for`, которые не ссылаются на переменные, привязанные к какому-нибудь предшествующему генератору, вряд ли разумно, поскольку многократные вычисления таких выражений повлекут за собой существенные издержки. Например, вместо:

```
for (x <- 1 to 1000; y = expensiveComputationNotInvolvingX)
                        (затратное вычисление, не использующее x)
yield x * y
```

во многих случаях лучше воспользоваться следующим кодом:

```
val y = expensiveComputationNotInvolvingX
        (затратное вычисление, не использующее x)
for (x <- 1 to 1000) yield x * y
```

---

[1] *Odersky M.* The Scala Language Specification, Version 2.9. EPFL, May 2011 [Электронный ресурс] / M. Odersky. — Режим доступа: www.scalalang.org/docu/manuals.html. — Дата доступа: 20.04.2014.

Трансляция, применяемая для циклов

В предыдущих подразделах было показано, как транслируется выражение `for`, содержащее оператор `yield`. А как насчет циклов, которые просто дают побочный эффект, не возвращая вообще ничего? Их трансляция выполняется аналогично, но является более простой, чем для выражений `for`. В принципе, там, где предыдущая схема трансляции применила `map` или `flatMap`, в схеме трансляции для циклов `for` используется просто `foreach`.

Например, выражение:

```
for (x <- expr1) body
```

транслируется в:

```
expr1 foreach ( x => body)
```

Более объемным примером будет выражение:

```
for (x <- expr1; if expr2; y <- expr3) body
```

Оно транслируется в следующий код:

```
expr1 withFilter (x => expr2) foreach (x =>
  expr3 foreach (y => body))
```

Например, следующее выражение суммирует все элементы матрицы, представленной в виде списка списков:

```
var sum = 0
for (xs <- xss; x <- xs) sum += x
```

Этот цикл транслируется в два вложенных применения `foreach`:

```
var sum = 0
xss foreach (xs =>
  xs foreach (x =>
    sum += x))
```

23.5. «Мы пойдем другим путем»

В предыдущем разделе мы показали, что выражения `for` могут транслироваться в использование функций высшего порядка: `map`, `flatMap` и `withFilter`. Но можно пойти и в обратном направлении и каждое применение `map`, `flatMap` или `withFilter` представить в виде выражения `for`.

Рассмотрим реализацию этих трех методов в понятиях выражений `for`. Методы содержатся в объекте `Demo`, чтобы их можно было отличить от стандартных операций над объектами типа `List`. Точнее, все три функции получают в качестве параметра значение типа `List`, но схема трансляции будет успешно работать и с другими типами коллекций:

```
object Demo {
  def map[A, B](xs: List[A], f: A => B): List[B] =
    for (x <- xs) yield f(x)

  def flatMap[A, B](xs: List[A], f: A => List[B]): List[B] =
    for (x <- xs; y <- f(x)) yield y

  def filter[A](xs: List[A], p: A => Boolean): List[A] =
    for (x <- xs if p(x)) yield x
}
```

Думаю, у вас не вызовет удивления то, что трансляция выражения `for`, использованного в теле метода `Demo.map`, превратится в вызов `map` класса `List`. По аналогии с этим `Demo.flatMap` и `Demo.filter` будут переведены в `flatMap` и `withFilter` класса `List`. Таким образом, эта небольшая демонстрация показывает, что выражения `for` действительно эквивалентны по своей выразительности применению функций `map`, `flatMap` и `withFilter`.

23.6. Обобщение for

Поскольку трансляция выражений `for` зависит только от наличия методов `map`, `flatMap` и `withFilter`, то систему записи выражений `for` можно применить к большому классу типов данных.

Вам уже встречалось применение выражений `for` в отношении списков и массивов. Их поддержка обусловливается тем, что для списков, как и для массивов, определены операции `map`, `flatMap` и `withFilter`. Поскольку для них также определен метод `foreach`, то при работе с этими типами данных доступны и циклы `for`.

Помимо списков и массивов, в стандартной библиотеке Scala есть множество других типов, поддерживающих те же четыре метода, а следовательно, позволяющих использовать выражения `for`. В качестве примеров можно назвать диапазоны, итераторы, потоки и все реализации множеств. Кроме того, вполне возможно путем определения всех необходимых методов получить поддержку выражений `for` для ваших собственных типов данных. Поддержка всего диапазона выражений `for` и циклов `for` в качестве методов вашего типа данных возможна с помощью определения методов `map`, `flatMap`, `withFilter` и `foreach`. Но можно также определить подмножество этих методов и, соответственно, получить подмножество всех потенциальных выражений и циклов `for`.

При этом нужно придерживаться следующих строгих правил.

- Если в вашем типе определяется лишь метод `map`, то можно будет применять выражения `for`, содержащие только один генератор.
- Если наряду с `map` определяется `flatMap`, то можно будет применять выражения `for`, содержащие несколько генераторов.

- Если определяется метод `foreach`, то можно будет применять циклы `for` (как с одним, так и с несколькими генераторами).
- Если определяется метод `withFilter`, то можно будет применять выражение фильтра, где выражение `for` начинается с `if`.

Трансляция выражений `for` происходит до проверки типов. Тем самым допускается максимальная гибкость, поскольку необходимо, чтобы проверку типов прошел лишь результат расширения выражения `for`. Для самих выражений `for` правила типизации в Scala не определяются, кроме того, не требуется наличие какой-либо конкретной сигнатуры типов в методах `map`, `flatMap`, `withFilter` или `foreach`.

Тем не менее существует стандартная ситуация, отражающая наиболее распространенный замысел, вкладываемый в методы высшего порядка, в которые транслируются выражения `for`. Предположим, имеется некий параметризованный класс `C`, который, как правило, будет представлять некую разновидность коллекции. Тогда вполне естественно будет выбрать для `map`, `flatMap`, `withFilter` и `foreach` следующие сигнатуры типов:

```
abstract class C[A] {
  def map[B](f: A => B): C[B]
  def flatMap[B](f: A => C[B]): C[B]
  def withFilter(p: A => Boolean): C[A]
  def foreach(b: A => Unit): Unit
}
```

То есть функция `map` получает функцию, отображающую элементы коллекции типа `A` к некоторому другому типу `B`. Она возвращает новую коллекцию того же вида `C`, но с элементами типа `B`. Метод `flatMap` получает функцию `f` от типа `A` к некой коллекции `C`, состоящей из элементов типа `B`, и возвращает коллекцию `C` из `B`. Метод `withFilter` получает функцию-предикат от типа элементов коллекции `A` к `Boolean`. Он возвращает коллекцию того же типа, что и та, в отношении которой он был вызван. И наконец, метод `foreach` получает функцию от `A` к `Unit` и возвращает результат типа `Unit`.

В показанном ранее классе `C` метод `withFilter` возвращает новую коллекцию того же класса. Это значит, при каждом вызове `withFilter` создается новый объект типа `C`, совпадающего с тем типом, с которым будет работать фильтр. Теперь в трансляции выражений `for` за любым вызовом `withFilter` всегда будет следовать вызов одного из трех других методов. Поэтому объект, создаваемый методом `withFilter`, будет сразу же разобран одним из других методов. Если объекты класса `C` велики (например, это длинные последовательности), то может потребоваться обойти создание такого промежуточного объекта. В качестве стандартного приема можно позволить методу `withFilter` возвращать не объект `C`, а просто объект-оболочку, который помнит, что элементы перед последующей обработкой должны быть отфильтрованы.

Концентрируясь только на первых трех функциях класса `C`, нужно отметить следующие факты. В функциональном программировании существует общее понятие *«монада»*, с помощью которого можно объяснить большое количество типов с вы-

числениями, начиная от коллекций и заканчивая, если называть лишь некоторые из них, вычислениями с состоянием и вводом-выводом, обратными вычислениями и транзакциями. Функции `map`, `flatMap` и `withFilter` можно сформулировать в понятиях монад, и если сделать это, то все сведется к тому, что у них будут точно такие же типы, что и здесь.

Более того, каждую монаду можно охарактеризовать с помощью `map`, `flatMap` и `withFilter`, если прибавить блочный конструктор, создающий монаду из значения элемента. В объектно-ориентированных языках этот блочный конструктор является просто конструктором экземпляра или фабричным методом. Поэтому методы `map`, `flatMap` и `withFilter` могут рассматриваться как объектно-ориентированные версии функциональной концепции монад. Поскольку выражения `for` эквивалентны применению этих трех методов, то могут рассматриваться в качестве синтаксиса для монад.

Все вышеприведенное говорит о том, что концепция выражения `for` более универсальна, чем просто обход элементов коллекции, и это действительно так. Например, выражения `for` также играют важную роль в асинхронном вводе-выводе или могут использоваться в качестве альтернативной формы записи для опциональных значений. В библиотеках Scala следите за появлениями `map`, `flatMap` и `withFilter`: там, где они присутствуют, сами собой напрашиваются выражения `for`, которые могут стать весьма лаконичным способом работы с элементами типа.

Резюме

В этой главе мы предложили заглянуть во внутреннюю кухню выражений и циклов `for`. Вы узнали, что они транслируются в применение стандартного набора методов высшего порядка. В результате было показано, что выражения `for` фактически имеют намного более универсальное применение, чем простой обход элементов коллекций, и что вы можете создавать в целях их поддержки собственные классы.

24 Углубленное изучение коллекций

В Scala включена весьма элегантная и эффективная библиотека коллекций. На первый взгляд API коллекций представляется незаметным, однако вызванные им изменения в вашем стиле программирования могут быть весьма существенными. Зачастую это похоже на работу на высоком уровне с основными строительными блоками программы, представляющими собой скорее коллекции, чем их элементы. Этот новый стиль программирования требует некоторой адаптации. К счастью, ее облегчает ряд привлекательных свойств коллекций Scala. Они просты в использовании, лаконичны в описании, безопасны, быстры в работе и универсальны.

- **Простота использования.** Небольшого словаря, содержащего от 20 до 50 методов, вполне достаточно для решения основного набора задач всего за пару операций. Не нужно морочить голову сложными цикличными структурами или рекурсиями. Стабильные коллекции и операции без побочных эффектов означают, что вам не следует опасаться случайного повреждения существующих коллекций новыми данными. Взаимовлияние итераторов и обновления коллекций исключено.
- **Лаконичность.** То, что раньше занимало от одного до нескольких циклов, теперь можно выразить всего одним словом. Можно выполнять функциональные операции, задействуя упрощенный синтаксис, и без особых усилий комбинировать операции таким образом, чтобы в результате получалось нечто похожее на обычные алгебраические формулы.
- **Безопасность.** Чтобы разобраться в этом вопросе, нужен определенный опыт. Природа коллекций Scala с их статической типизацией и функциональностью означает, что подавляющее большинство потенциальных ошибок отлавливается еще во время компиляции. Это достигается благодаря следующему:
  - сами операции над коллекциями используются довольно широко, а следовательно, прошли проверку временем;

- использование операций над коллекциями делает ввод и вывод явным в виде параметров функций и результатов;
- эти явные входные и выходные данные являются предметом для статической проверки типов.

Суть заключается в том, что большинство случаев неверного использования кода проявится в виде ошибок типа. И вовсе не редкость, когда программы, состоящие из нескольких сотен строк кода, запускаются с первой же попытки.

- **Скорость.** Операции с коллекциями, имеющиеся в библиотеках, уже настроены и оптимизированы. В результате этого использование коллекций обычно отличается высокой эффективностью. Тщательно настроенные структуры данных и операции могут улучшить ситуацию, но в то же время, принимая решения, далекие от оптимальных, можно ее значительно ухудшить. Более того, коллекции были адаптированы под параллельную обработку на многоядерных системах. Параллельно обрабатываемые коллекции поддерживают те же самые операции, что и последовательно обрабатываемые, поэтому изучать новые операции и переписывать код не нужно. Последовательно обрабатываемые коллекции можно превратить в параллельно обрабатываемые, просто вызвав метод `par`.
- **Универсальность.** Коллекции реализуют одни и те же операции над любым типом там, где в этих операциях есть определенный смысл. Следовательно, используя сравнительно небольшой словарь операций, вы приобретаете множество возможностей. Например, концептуально строка является последовательностью символов. Следовательно, в коллекциях Scala строки поддерживают все операции с последовательностями. То же самое справедливо и для массивов.

В этой главе мы даем углубленное описание API имеющихся в Scala классов коллекций с точки зрения их использования. Краткий тур по библиотекам коллекций мы совершили в главе 17. В этой главе нам предстоит поучаствовать в более подробном путешествии, в ходе которого мы покажем все классы коллекций и все определенные в них методы, то есть представим все сведения, необходимые для использования коллекций Scala. Забегая вперед: для тех, кто собирается заниматься реализацией новых типов коллекций, в главе 25 основное внимание мы уделим вопросам архитектуры библиотек и возможностям их расширения.

24.1. Изменяемые и неизменяемые коллекции

Как вам уже известно, в целом коллекции в Scala подразделяются на изменяемые и неизменяемые. Изменяемые могут обновляться или расширяться на месте. Это значит, что вы можете изменять, добавлять или удалять элементы коллекции

как побочный эффект. Неизменяемые коллекции, напротив, всегда сохраняют неизменный вид. Но все же есть операции, имитирующие добавление, удаление или обновление, но каждая из таких операций возвращает новую коллекцию, оставляя старую без изменений.

Все классы коллекций находятся в пакете `scala.collection` или в одном из его подпакетов: `mutable`, `immutable` и `generic`. Большинство классов коллекций, востребованных в клиентском коде, существуют в трех вариантах, которые имеют разные характеристики в смысле возможности изменения. Эти три варианта находятся в пакетах `scala.collection`, `scala.collection.immutable` и `scala.collection.mutable`.

Коллекция в пакете `scala.collection.immutable` гарантированно неизменяемая для всех. Такая коллекция после создания всегда остается неизменной. Поэтому можно полагаться на то, что многократные обращения к одному и тому же значению коллекции в разные моменты времени всегда будут давать коллекцию с теми же элементами.

Коллекция в пакете `scala.collection.mutable` известна тем, что имеет ряд операций, изменяющих коллекцию на месте. Эти операции позволяют вам создавать код для самостоятельного изменения коллекции. Но при этом нужно четко понимать, что существует вероятность обновления из других частей исходного кода, и защищаться от нее.

Коллекции в пакете `scala.collection` могут быть как изменяемыми, так и неизменяемыми. Например, `scala.collection.IndexedSeq[T]` является супертрейтом как для `scala.collection.immutable.IndexedSeq[T]`, так и для его изменяемого брата `scala.collection.mutable.IndexedSeq[T]`. В целом корневые коллекции в пакете `scala.collection` поддерживают трансформирующие операции, которые влияют на целую коллекцию, такие как `map` и `filter`. Неизменяемые коллекции в пакете `scala.collection.immutable` обычно дополняют операциями добавления и удаления отдельных значений, а изменяемые коллекции в пакете `scala.collection.mutable` дополняются некоторыми модифицирующими операциями с побочными эффектами.

Между корневыми и неизменяемыми коллекциями есть еще одно различие: клиенты неизменяемых коллекций получают гарантии того, что никто не сможет внести в коллекцию изменения, а клиенты корневых коллекций знают только то, что не могут изменить коллекцию самостоятельно. Даже если статический тип такой коллекции не предоставляет операций для ее изменения, все же существует вероятность того, что в процессе выполнения программы она получит тип изменяемой коллекции, предоставив другим клиентам возможность вносить в нее изменения.

По умолчанию в Scala всегда выбираются неизменяемые коллекции. Например, если просто написать `Set` без какого-либо префикса или ничего не импортируя, то будет получено неизменяемое множество, а если написать `Iterable` — то неизменяемый объект с возможностью обхода его элементов, поскольку имеются привязки по умолчанию, импортируемые из пакета `scala`. Чтобы получить изменяемые

версии по умолчанию, следует явно указать `collection.mutable.Set` или `collection.mutable.Iterable`.

Последний пакет в иерархии коллекций — `collection.generic`. В нем содержатся строительные блоки для абстрагирования поверх конкретных коллекций. Впрочем, постоянные пользователи коллекций должны ссылаться на классы в пакете `generic` только в крайних случаях.

24.2. Согласованность коллекций

Наиболее важные классы коллекций показаны на рис. 24.1.

```
Iterable
    Seq
        IndexedSeq
            ArraySeq
            Vector
            ArrayDeque (mutable)
            Queue (mutable)
            Stack (mutable)
            Range
            NumericRange
        LinearSeq
            List
            LazyList
            Queue (immutable)
        Buffer
            ListBuffer
            ArrayBuffer
    Set
        SortedSet
            TreeSet
        HashSet (mutable)
        LinkedHashSet
        HashSet (immutable)
        BitSet
        EmptySet, Set1, Set2, Set3, Set4
    Map
        SortedMap
            TreeMap
        HashMap (mutable)
        LinkedHashMap (mutable)
        HashMap (immutable)
        VectorMap (immutable)
        EmptyMap, Map1, Map2, Map3, Map4
```

Рис. 24.1. Иерархия коллекций

Эти классы имеют много общего. Например, каждая разновидность коллекции может быть создана с использованием единообразного синтаксиса, заключающегося в записи названия класса коллекции, за которым стоят элементы данной коллекции:

```
Iterable("x", "y", "z")
Map("x" -> 24, "y" -> 25, "z" -> 26)
Set(Color.Red, Color.Green, Color.Blue)
SortedSet("hello", "world")
Buffer(x, y, z)
IndexedSeq(1.0, 2.0)
LinearSeq(a, b, c)
```

Тот же принцип применяется и к конкретным реализациям коллекций:

```
List(1, 2, 3)
HashMap("x" -> 24, "y" -> 25, "z" -> 26)
```

Методы `toString` для всех коллекций выводят информацию, аналогичную показанной ранее, с именем типа, за которым следуют значения элементов коллекции, заключенные в круглые скобки. Все коллекции поддерживают API, предоставляемый `Iterable`, но все их методы возвращают собственный класс, а не корневой класс `Iterable`. Например, у метода `map` класса `List` тип возвращаемого значения `List`, у метода `map` класса `Set` тип возвращаемого значения `Set`. То есть статический возвращаемый тип этих методов абсолютно точен:

```
scala> List(1, 2, 3) map (_ + 1)
res0: List[Int] = List(2, 3, 4)

scala> Set(1, 2, 3) map (_ * 2)
res1: scala.collection.immutable.Set[Int] = Set(2, 4, 6)
```

Эквивалентность для всех классов коллекций также организована единообразно: более подробно этот вопрос рассматривается в разделе 24.12.

Большинство классов, показанных на рис. 24.1 (см. выше), существует в трех вариантах: корневом, изменяемом и неизменяемом (root, mutable и immutable). Единственное исключение — трейт `Buffer`, существующий только в виде изменяемой коллекции.

Далее в главе все эти классы мы рассмотрим поочередно.

24.3. Трейт Iterable

На вершине иерархии коллекций находится трейт `Iterable[A]`, где `A` — тип элементов коллекции. Все методы в этом трейте определены в терминах абстрактного метода `iterator`, который возвращает элементы коллекции один за другим.

```
def iterator: Iterator[A]
```

Классам коллекций, реализующим `Iterable`, нужно определить только этот метод, а все остальные методы могут быть унаследованы из `Iterable`.

В `Iterable` также определяется много конкретных методов (перечислены ниже, в табл. 24.1). Их можно разбить на следующие категории.

- **Операции итерирования** — `foreach`, `grouped`, `sliding` — перебирают все элементы коллекции в порядке, который определяет ее итератор. Методы `grouped` и `sliding` возвращают итераторы, которые, в свою очередь, возвращают не отдельные элементы, а, скорее, подпоследовательности элементов исходной коллекции. Максимальный размер этих подпоследовательностей указывается в качестве аргумента для этих методов. Метод `grouped` делит полученные элементы на инкременты, а `sliding` формирует по ним скользящее окно. Разницу между ними должен наглядно продемонстрировать следующий сеанс интерпретатора:

```
scala> val xs = List(1, 2, 3, 4, 5)
xs: List[Int] = List(1, 2, 3, 4, 5)

scala> val git = xs grouped 3
git: Iterator[List[Int]] = non-empty iterator

scala> git.next()
res2: List[Int] = List(1, 2, 3)

scala> git.next()
res3: List[Int] = List(4, 5)

scala> val sit = xs sliding 3
sit: Iterator[List[Int]] = non-empty iterator

scala> sit.next()
res4: List[Int] = List(1, 2, 3)

scala> sit.next()
res5: List[Int] = List(2, 3, 4)

scala> sit.next()
res6: List[Int] = List(3, 4, 5)
```

- **Сложение** — `++` (псевдоним: `concat`) — складывает две коллекции вместе или добавляет все элементы итератора в коллекцию.
- **Операции отображения** — `map`, `flatMap` и `collect` создают новую коллекцию путем применения некой функции к элементам коллекции.
- **Операции преобразования** — `toIndexedSeq`, `toIterable`, `toList`, `toMap`, `toSeq`, `toSet` и `toVector` — превращают коллекцию `Iterable` в неизменяемую. Все эти преобразования возвращают объект-получатель, если он уже соответствует требуемому типу коллекции. Например, применение `toList` к списку выдаст сам

список. Методы `toArray` и `toBuffer` возвращают новую изменяемую коллекцию, даже если объект-получатель соответствует. Метод `to` можно использовать для преобразования в любую другую коллекцию.

- **Операция копирования** — `copyToArray`. Как следует из названия, копирует элементы коллекции в массив.
- **Операции с размером** — `isEmpty`, `nonEmpty`, `size`, `knownSize`, `sizeCompare` и `sizeIs` — имеют отношение к размеру коллекции. Чтобы вычислить количество элементов коллекции, в некоторых случаях может потребоваться совершить обход, например, в случае `List`. В других случаях коллекция может содержать бесконечное количество элементов, например `LazyList.from(0)`. Методы `knownSize`, `sizeCompare` и `sizeIs` предоставляют информацию о количестве элементов, обходя как можно меньшее количество элементов.
- **Операции извлечения элементов** — `head`, `last`, `headOption`, `lastOption` и `find` — выбирают первый или последний элемент коллекции или же первый элемент, соответствующий условию. Однако следует заметить, что не все коллекции имеют четко определенное значение первого и последнего. Например, `HashSet` может хранить элементы в соответствии с их хеш-ключами, и порядок их следования от запуска к запуску может изменяться. В таком случае для разных запусков программы первый элемент `HashSet` также может быть разным. Коллекция считается *упорядоченной*, если всегда выдает свои элементы в одном и том же порядке. Большинство коллекций упорядочены, однако некоторые, такие как `HashSet`, таковыми не являются, отказ от упорядочения позволяет им быть более эффективными. Упорядочение зачастую необходимо, чтобы получать воспроизводимые тесты и способствовать отладке. Поэтому коллекции Scala предоставляют упорядоченные альтернативы для всех типов коллекций. Например, упорядоченная альтернатива для `HashSet` — `LinkedHashSet`.
- **Операции извлечения подколлекций** — `takeWhile`, `tail`, `init`, `slice`, `take`, `drop`, `filter`, `dropWhile`, `filterNot` и `withFilter` — возвращают подколлекцию, определяемую диапазоном индексов или предикатом.
- **Операции подразделения** — `groupBy`, `groupMap`, `groupMapReduce`, `splitAt`, `span`, `partition` и `partitionMap` — разбивают элементы переданной коллекции на несколько подколлекций.
- **Операции тестирования элементов** — `exists`, `forall` и `count` — тестируют элементы коллекции на соответствие заданному предикату.
- **Операции свертки** — `foldLeft`, `foldRight`, `reduceLeft`, `reduceRight` — применяют бинарные операции к следующим друг за другом элементам.
- **Операции специализированных сверток** — `sum`, `product`, `min` и `max` — работают с коллекциями конкретных типов (numeric или comparable).

- **Операции со строками** — `mkString` и `addString` — предоставляют альтернативные способы преобразования коллекции в строку.
- **Операция представления** — `view` — это коллекция, которая вычисляется лениво. Больше о представлениях вы узнаете из раздела 24.13.

Таблица 24.1. Операции в трейте Iterable

Что собой представляет	Результат работы
Абстрактный метод	
xs.iterator	Итератор, который возвращает каждый элемент в xs
Итерирование	
xs foreach f	Применяет функцию f к каждому элементу xs. Вызов f нужен только для получения побочных эффектов; по сути, foreach отбрасывает любой результат функции f
xs grouped size	Итератор, который возвращает блоки коллекции фиксированного размера
xs sliding size	Итератор, который возвращает скользящее окно по элементам этой коллекции
Сложение	
xs ++ ys (или xs concat ys)	Коллекция, состоящая из элементов xs и ys. Элемент ys — это коллекция IterableOnce, то есть Iterable или Iterator
Отображение	
xs map f	Коллекция, получающаяся в результате применения функции f к каждому элементу в xs
xs flatMap f	Коллекция, получающаяся в результате применения функции f, результатом которой является коллекция, к каждому элементу в xs, и объединяющая результаты
xs collect f	Коллекция, получающаяся в результате применения частично примененной функции f к каждому элементу в xs, для которого она определена, и объединяющая результаты
Преобразование	
xs.toArray	Превращает коллекцию в массив
xs.toList	Превращает коллекцию в список
xs.toIterable	Превращает коллекцию в итерируемую коллекцию
xs.toSeq	Превращает коллекцию в последовательность

Продолжение

Таблица 24.1 *(продолжение)*

Что собой представляет	Результат работы
xs.toIndexedSeq	Превращает коллекцию в проиндексированную последовательность
xs.toSet	Превращает коллекцию во множество
xs.toMap	Превращает коллекцию пар «ключ — значение» в отображение
xs to SortedSet	Обобщенная операция преобразования, которая принимает в качестве параметра фабрику коллекций
Копирование	
xs copyToArray(arr, s, len)	Копирует максимум len элементов в массиве arr, начиная с индекса s. Последние два аргумента необязательны
Получение информации о размере	
xs.isEmpty	Проверяет, является ли коллекция пустой
xs.nonEmpty	Проверяет, содержит ли коллекция элементы
xs.size	Количество элементов в коллекции
xs.knownSize	Количество элементов, если его можно вычислить за постоянное время. В противном случае –1
xs sizeCompare ys	Возвращает отрицательное значение, если коллекция xs короче ys, положительное, если длиннее, и 0, если обе коллекции имеют одинаковый размер. Работает даже для бесконечных коллекций
xs.sizeIs < 42, xs.sizeIs != 42 и т. д.	Сравнивает размер коллекции с заданным значением, перебирая как можно меньше элементов
Извлечение элементов	
xs.head	Первый элемент коллекции (или какой-нибудь элемент, если порядок следования элементов не определен)
xs.headOption	Первый элемент xs в Option-значении или None, если xs пуст
xs.last	Последний элемент коллекции (или какой-нибудь элемент, если порядок следования элементов не определен)
xs.lastOption	Последний элемент xs в Option-значении параметра или None, если xs пуст
xs find p	Option-значение, содержащее первый элемент в xs, удовлетворяющий условию p, или None, если такого элемента нет
Создание подколлекций	
xs.tail	Вся коллекция, за исключением xs.head
xs.init	Вся коллекция, за исключением xs.last
xs.slice(from, to)	Коллекция, состоящая из элементов в некотором диапазоне индексов xs (от from включительно до to не включительно)

Что собой представляет	Результат работы
xs take n	Коллекция, состоящая из первых n элементов xs (или n каких-либо произвольных элементов, если порядок следования элементов не определен)
xs drop n	Вся коллекция xs за вычетом тех элементов, которые получаются при использовании выражения xs take n
xs takeWhile p	Самый длинный префикс элементов в коллекции, каждый из которых соответствует условию p
xs dropWhile p	Коллекция без самого длинного префикса элементов, каждый из которых соответствует условию p
xs takeRight n	Коллекция, состоящая из заключительных n элементов xs (или произвольных n элементов, если порядок не определен)
xs dropRight n	Вся коллекция, кроме xs takeRight n
xs filter p	Коллекция, состоящая только из тех элементов xs, которые соответствуют условию p
xs withFilter p	Нестрогий фильтр коллекции. Все операции в отношении фильтруемых элементов будут применяться только к тем элементам xs, для которых условие p вычисляется в true
xs filterNot p	Коллекция, состоящая из тех элементов xs, которые не соответствуют условию p
Слияние	
xs zip ys	Итерируемые пары соответствующих элементов из xs и ys
xs lazyZip ys	Значение, предоставляющее методы для поэлементной работы с коллекциями xs и ys. См. раздел 16.9
xs.zipAll(ys, x, y)	Итерируемые пары соответствующих элементов из xs и ys; если одна из последовательностей короче, то расширяется за счет добавления элементов x или y
xs.zipWithIndex	Итерируемые пары элементов из xs с их индексами
Разбиение	
xs splitAt n	Разбивает xs по позиции на пару коллекций (xs take n, xs drop n)
xs span p	Разбивает xs в соответствии с предикатом на пару коллекций (xs takeWhile p, xs.dropWhile p)
xs partition p	Разбивает xs на пару коллекций, в одной из которых находятся все элементы, удовлетворяющие условию p, а в другой — все элементы, не удовлетворяющие этому условию (xs filter p, xs.filterNot p)

Продолжение

Таблица 24.1 *(продолжение)*

Что собой представляет	Результат работы
xs partitionMap f	Преобразует каждый элемент xs в значение Either[X, Y] и разбивает результат на две коллекции: одна с элементами из Left, а другая — с элементами из Right
xs groupBy f	Разбивает xs на отображение коллекций в соответствии с функцией-дискриминатором f
xs.groupMap(f)(g)	Превращает xs в отображение коллекций в соответствии с функцией-дискриминатором f и применяет функцию преобразования g к каждому элементу каждой коллекции
xs.groupMapReduce(f)(g)(h)	Превращает xs в отображение коллекций в соответствии с функцией-дискриминатором f, применяет функцию преобразования g к каждому элементу каждой коллекции и сводит каждую коллекцию к единственному значению, объединяя ее элементы с помощью функции h
Состояние элементов	
xs forall p	Булево значение, показывающее, соблюдается ли условие p для всех элементов xs
xs exists p	Булево значение, показывающее, соблюдается ли условие p для каких-либо элементов xs
xs count p	Количество элементов в xs, удовлетворяющих условию p
Свертка	
xs.foldLeft(z)(op)	Применяет бинарную операцию op к последовательным элементам xs, продвигаясь слева направо, начиная с z
xs.foldRight(z)(op)	Применяет бинарную операцию op к последовательным элементам xs, продвигаясь справа налево, начиная с z
xs reduceLeft op	Применяет бинарную операцию op к последовательным элементам xs непустой коллекции xs, продвигаясь слева направо
xs reduceRight op	Применяет бинарную операцию op к последовательным элементам xs непустой коллекции xs, продвигаясь справа налево
Специализированная свертка	
xs.sum	Сумма числовых элементов коллекции xs
xs.product	Произведение числовых элементов коллекции xs
xs.min	Минимальное значение упорядочиваемых элементов коллекции xs
xs.max	Максимальное значение упорядочиваемых элементов коллекции xs

Что собой представляет	Результат работы
Строки	
xs addString (b, start, sep, end)	Добавляет строку к строковому буферу b типа StringBuilder со всеми элементами xs с разделителями sep, заключенными между строками start и end. Аргументы start, sep и end являются необязательными
xs mkString (start, sep, end)	Превращает коллекцию в строку со всеми элементами xs с разделителями sep, заключенными между строками start и end. Аргументы start, sep и end являются необязательными
Представления	
xs.view	Создает представление xs

Подкатегории Iterable

В иерархии наследования ниже `Iterable` находятся три трейта: `Seq`, `Set` и `Map`. Трейты `Seq` и `Map` объединяет то, что они реализуют трейт `PartialFunction`[1] с методами `apply` и `isDefinedAt`. Но способы реализации `PartialFunction` у каждого трейта свои.

Для последовательностей `apply` — позиционное индексирование, в котором элементы всегда нумеруются с нуля. То есть `Seq(1, 2, 3)(1) == 2`. Для множеств `apply` — проверка на принадлежность. Например, `Set('a', 'b', 'c')('b') == true`, а `Set()('a') == false`. И наконец, для отображений `apply` — средство выбора. Например, `Map('a' -> 1, 'b' -> 10, 'c' -> 100)('b') == 10`.

Более подробно каждый из этих видов коллекций мы рассмотрим в следующих трех разделах.

24.4. Трейты последовательностей Seq, IndexedSeq и LinearSeq

Трейт `Seq` представляет последовательности. Последовательностью называется разновидность итерируемой коллекции, у которой есть длина и все элементы которой имеют фиксированные индексированные позиции, отсчет которых начинается с нуля. Операции с последовательностями (сведены в табл. 24.2 ниже) разбиваются на следующие категории.

- **Операции индексирования и длины** — `apply`, `isDefinedAt`, `length`, `indices` `lengthCompare` и `lengthIs`. Для `Seq` операция `apply` означает индексирование,

[1] Частично определенные функции описаны в разделе 15.7.

поэтому последовательность типа `Seq[T]` является частично примененной функцией, которая получает аргумент типа `Int` (индекс) и выдает элемент последовательности типа `T`. Иными словами, `Seq[T]` расширяет `PartialFunction[Int, T]`. Элементы последовательности индексируются от нуля до длины последовательности `length` за вычетом единицы. Метод `length`, применяемый в отношении последовательностей, — псевдоним общего для коллекций метода `size`. Метод `lengthCompare` позволяет сравнивать длины двух последовательностей, даже если одна из них имеет бесконечную длину. Метод `lengthIs` — псевдоним метода `sizeIs`.

- **Операции поиска индекса** — `indexOf`, `lastIndexOf`, `indexOfSlice`, `lastIndexOfSlice`, `indexWhere`, `lastIndexWhere`, `segmentLength` и `prefixLength` — возвращают индекс элемента, равного заданному значению или соответствующего некоему условию.
- **Операции добавления** — `+:` (псевдоним: `prepended`), `++:` (псевдоним: `prependedAll`), `:+` (псевдоним: `appended`), `:++` (псевдоним: `appendedAll`) и `padTo` — возвращают новые последовательности, получаемые путем добавления к началу или концу последовательности.
- **Операции обновления** — `updated` и `patch` — возвращают новую последовательность, получаемую путем замены некоторых элементов исходной последовательности.
- **Операции сортировки** — `sorted`, `sortWith` и `sortBy` — сортируют элементы последовательности в соответствии с различными критериями.
- **Операции реверсирования** — `reverse` и `reverseIterator` — обрабатывают или возвращают элементы последовательности в обратном порядке, с последнего до первого.
- **Операции сравнения** — `startsWith`, `endsWith`, `contains`, `corresponds`, `containsSlice` и `search` — определяют соотношение двух последовательностей или ищут элемент в последовательности.
- **Операции над множествами** — `intersect`, `diff`, `distinct` и `distinctBy` — выполняют над элементами двух последовательностей операции, подобные операциям с множествами, или удаляют дубликаты.

Для изменяемой последовательности предлагается дополнительный метод добавления `update` с побочным эффектом, который позволяет элементам последовательности обновляться. Вспомним: в главе 3 мы уже говорили, что синтаксис вида `seq(idx) = elem` — не более чем сокращение для выражения `seq.update(idx, elem)`. Обратите внимание на разницу между методами `update` и `updated`. Первый изменяет элемент последовательности на месте и доступен только для изменяемых последовательностей. Второй доступен для всех последовательностей и всегда вместо изменения исходной возвращает новую последовательность.

Таблица 24.2. Операции в трейте Seq

Что собой представляет	Результат работы
Индексирование и длина	
xs(i)	(Или после раскрытия xs apply i) элемент xs с индексом i
xs isDefinedAt i	Проверяет, содержится ли i в xs.indices
xs.length	Длина последовательности (то же самое, что и size)
xs.lengthCompare len	Возвращает отрицательное значение Int, если длина xs меньше len, положительное, если больше, и 0, если они равны. Работает даже для бесконечных коллекций xs
xs.indices	Диапазон индексов xs от 0 до xs.length – 1
Поиск индекса	
xs indexOf x	Индекс первого элемента в xs, равного значению x (существует несколько вариантов)
xs lastIndexOf x	Индекс последнего элемента в xs, равного значению x (существует несколько вариантов)
xs indexOfSlice ys	Первый индекс xs при условии, что следующие друг за другом элементы, начинающиеся с элемента с этим индексом, составляют последовательность ys
xs lastIndexOfSlice ys	Последний индекс xs при условии, что следующие друг за другом элементы, начинающиеся с элемента с этим индексом, составляют последовательность ys
xs indexWhere p	Индекс первого элемента в xs, удовлетворяющего условию p (существует несколько вариантов)
xs segmentLength (p, i)	Длина самого длинного непрерывного сегмента элементов в xs, начинающегося с xs(i), который удовлетворяет условию p
Добавление	
x +: xs (или xs prepended x)	Новая последовательность со значением x, добавленным в начало xs
ys ++: xs (или xs prependedAll ys)	Новая последовательность, состоящая из всех элементов ys, добавленных в начало xs
xs :+ x (или xs appended x)	Новая последовательность со значением x, добавленным в конец xs
xs :++ ys (или xs appendedAll ys)	Новая последовательность, состоящая из всех элементов ys, добавленных в конец xs. То же самое, что xs ++ ys
xs padTo (len, x) (или xs prepended x)	Последовательность, получающаяся при добавлении значения x к xs, до тех пор пока длина не достигнет значения len
Обновление	
xs.patch(i, ys, r)	Последовательность, получающаяся путем замены r элементов последовательности xs, начиная с i, элементами последовательности ys

Продолжение

Таблица 24.2 *(продолжение)*

Что собой представляет	Результат работы
xs.updated(i, x)	Копия xs, в которой элемент с индексом i заменяется значением x
xs(i) = x	(Или после раскрытия xs.update(i, x), которая доступна только для изменяемых последовательностей mutable.Seq) изменяет значение элемента xs с индексом i на значение x
Сортировка	
xs.sorted	Новая последовательность, полученная путем сортировки элементов xs в порядке следования элементов типа xs
xs sortWith lessThan	Новая последовательность, полученная путем сортировки элементов xs с использованием в качестве операции сравнения lessThan (меньше чем)
xs sortBy f	Новая последовательность, полученная путем сортировки элементов xs. Сравнение двух элементов выполняется путем применения к ним функции f и сравнения результатов
Реверсирование	
xs.reverse	Последовательность из элементов xs, следующих в обратном порядке
xs.reverseIterator	Итератор, выдающий все элементы xs в обратном порядке
Сравнение	
xs sameElememts ys	Проверяет, содержат ли xs и ys одинаковые элементы в одном и том же порядке
xs startsWith ys	Проверяет, не начинается ли xs с последовательности ys (имеется несколько вариантов)
xs endsWith ys	Проверяет, не заканчивается ли xs последовательностью ys (имеется несколько вариантов)
xs contains x	Проверяет, имеется ли в xs элемент, равный x
xs search x	Проверяет, содержит ли отсортированная коллекция xs элемент, равный x, и делает это потенциально эффективнее, чем xs contains x
xs containsSlice ys	Проверяет, имеется ли в xs непрерывная последовательность, равная ys
xs.corresponds(ys)(p)	Проверяет, удовлетворяют ли соответствующие элементы xs и ys бинарному предикату p
Операции над множествами	
xs intersect ys	Пересечение множеств, состоящих из элементов последовательностей xs и ys, которое сохраняет порядок следования элементов в xs
xs diff ys	Разность множеств, состоящих из элементов последовательностей xs и ys, которое сохраняет порядок следования элементов в xs

Что собой представляет	Результат работы
xs.distinct	Часть последовательности xs, не содержащая дубликатов
xs.distinctBy f	Подпоследовательность xs, в которой после применения преобразующей функции f нет повторяющихся элементов

У каждого трейта `Seq` есть два подтрейта: `LinearSeq` и `IndexedSeq`. Они не добавляют никаких новых операций, но каждый их них предлагает разные характеристики производительности. У линейной последовательности (linear sequence) есть эффективные операции `head` и `tail`, а у индексированной — эффективные операции `apply`, `length` и (если последовательность изменяемая) `update`. В `List` зачастую применяется линейная последовательность, как и в `LazyList`. Представители двух часто используемых индексированных последовательностей — `Array` и `ArrayBuffer`. Класс `Vector` обеспечивает весьма интересный компромисс между индексированным и последовательным доступом. У него практически постоянные линейные издержки как на индексированный, так и на последовательный доступ. Поэтому векторы служат хорошей основой для смешанных схем доступа, в которых используется как индексированный, так и последовательный доступ. Более подробно мы рассмотрим векторы в разделе 24.7.

Изменяемый подтрейт `IndexedSeq` добавляет операции для преобразования имеющихся элементов. В отличие от `map` и `sort`, которые доступны в `Seq`, эти операции, представленные в табл. 24.3, не возвращают новый экземпляр коллекции.

Таблица 24.3. Операции в трейте mutable.IndexedSeq

Что собой представляет	Результат работы
Добавление	
xs mapInPlace f	Преобразует все элементы xs, применяя к каждому из них функцию f
xs.sortInPlace()	Сортирует элементы xs на месте
xs sortInPlaceBy f	Сортирует элементы xs на месте и в соответствии с порядком, который определяется путем применения функции f к каждому элементу
xs sortInPlaceWith c	Сортирует элементы xs на месте в соответствии с функцией сравнения c

Буферы

Важная подкатегория изменяемых последовательностей — буферы. Они позволяют не только обновлять существующие элементы, но и вставлять и удалять элементы, а также с высокой эффективностью добавлять новые элементы в конец буфера. Принципиально новые методы, поддерживаемые буфером, — `+=` (псевдоним: `append`) и `++=` (псевдоним: `appendAll`) для добавления элементов в конец буфера,

`+=:` (псевдоним: `prepend`) и `++=:` (псевдоним: `prependAll`) для добавления элементов в начало буфера, `insert` и `insertAll` для вставки элементов, а также `remove`, `-=` (псевдоним: `subtractOne`) и `--=` (псевдоним: `subtractAll`) для удаления элементов. Все эти операции сведены в табл. 24.4.

Таблица 24.4. Операции в трейте Buffer

Что собой представляет	Результат работы
Добавление	
buf += x (или buf append x)	Добавляет элемент x в конец buf и возвращает в качестве результата сам buf
buf ++= xs (или buf appendAll xs)	Добавляет в конец буффера все элементы xs
x +=: buf (или buf prepend x)	Добавляет элемент x в начало буфера
xs ++=: buf (или buf prependAll xs)	Добавляет в начало буфера все элементы xs
buf insert (i, x)	Вставляет элемент x в то место в буфере, на которое указывает индекс i
buf insertAll (i, xs)	Вставляет все элементы xs в то место в буфере, на которое указывает индекс i
buf.padToInPlace(n, x)	Добавляет в буфер элементы x, пока общее количество его элементов не достигнет n
Удаление	
buf –= x (или buf subtractOne x)	Удаляет из буфера элемент x
buf ––= x (или buf subtractAll xs)	Удаляет из буфера все элементы xs
buf remove i	Удаляет из буфера элемент с индексом i
buf remove (i, n)	Удаляет из буфера n элементов, начиная с элемента с индексом i
buf trimStart n	Удаляет из буфера первые n элементов
buf trimEnd n	Удаляет из буфера последние n элементов
buf.clear()	Удаляет из буфера все элементы
Замена	
buf.patchInPlace(i, xs, n)	Заменяет (максимум) n элементов буфера элементами из xs, начиная с индекса i
Копирование	
buf.clone	Новый буфер с теми же элементами, что и в buf

Наиболее часто используются две реализации буферов: `ListBuffer` и `ArrayBuffer`. Исходя из названий, `ListBuffer` базируется на классе `List` и поддерживает высокоэффективное преобразование своих элементов в список, а `ArrayBuffer` базируется на массиве и может быть быстро превращен в массив. Наметки реализации `ListBuffer` мы показали в разделе 22.2.

24.5. Множества

Коллекции `Set` — это итерируемые `Iterable`-коллекции, которые не содержат повторяющихся элементов. Общие операции над множествами сведены в табл. 24.5, в табл. 24.6 показаны операции для неизменяемых множеств, а в табл. 24.7 — операции для изменяемых множеств. Операции разбиты на следующие категории.

- **Операции проверки** — `contains`, `apply` и `subsetOf`. Метод `contains` показывает, содержит ли множество заданный элемент. Метод `apply` для множества является аналогом `contains`, поэтому `set(elem)` — то же самое, что и `set contains elem`. Следовательно, множества могут также использоваться в качестве тестовых функций, возвращающих `true` для содержащихся в них элементов, например:

```
scala> val fruit = Set("apple", "orange", "peach", "banana")
fruit: scala.collection.immutable.Set[String] =
  Set(apple, orange, peach, banana)

scala> fruit("peach")
res7: Boolean = true

scala> fruit("potato")
res8: Boolean = false
```

- **Операции добавления** — `+` (псевдоним: `incl`) и `++` (псевдоним: `concat`) — добавляют во множество один и более элементов, возвращая в качестве результата новое множество.
- **Операции удаления** — `-` (псевдоним: `excl`) и `--` (псевдоним: `removedAll`) — удаляют из множества один и более элементов, возвращая новое множество.
- **Операции над множествами** для объединения, пересечения и разности множеств. Существуют в двух формах: текстовом и символьном. К текстовым относятся версии `intersect`, `union` и `diff`, а к символьным — `&`, `|` и `&~`. Оператор `++`, наследуемый `Set` из `Iterable`, может рассматриваться в качестве еще одного псевдонима `union` или `|`, за исключением того, что `++` получает `IterableOnce`-аргумент, а `union` и `|` получают множества.

Неизменяемые множества предлагают методы добавления и удаления элементов путем возвращения новых множеств, которые сведены в табл. 24.6.

У изменяемых множеств есть методы, которые добавляют, удаляют и обновляют элементы, которые сведены в табл. 24.7.

Таблица 24.5. Операции в трейте Set

Что собой представляет	Результат работы
Проверка	
xs contains x	Проверяет, является ли x элементом xs
xs(x)	Делает то же самое, что и xs contains x
xs subsetOf ys	Проверяет, является ли xs подмножеством ys
Удаление	
xs.empty	Пустое множество того же класса, что и xs
Бинарные операции	
xs & ys (или xs intersect ys)	Пересечение множеств xs и ys
xs | ys (или xs union ys)	Объединение множеств xs и ys
xs & ~ ys (или xs diff ys)	Разность множеств xs и ys

Таблица 24.6. Операции в трейте immutable.Set

Что собой представляет	Результат работы
Добавление	
xs + x (или xs incl x)	Множество, содержащее все элементы xs и элемент x
xs ++ ys (или xs concat ys)	Множество, содержащее все элементы xs и все элементы ys
Удаление	
xs – x (или xs excl x)	Множество, содержащее все элементы xs, кроме x
xs –– ys (или xs removedAll ys)	Множество, содержащее все элементы xs, кроме элементов множества ys

Таблица 24.7. Операции в трейте mutable.Set

Что собой представляет	Результат работы
Добавление	
xs += x (или xs addOne x)	Добавляет элемент x во множество xs как побочный эффект и возвращает само множество xs
xs ++= ys (или xs addAll ys)	Добавляет все элементы ys во множество xs как побочный эффект и возвращает само множество xs
xs add x	Добавляет элемент x в xs и возвращает true, если x прежде не был во множестве, или false, если уже был

Что собой представляет	Результат работы
Удаление	
xs –= x (или xs subtractOne x)	Удаляет элемент x из множества xs как побочный эффект и возвращает само множество xs
xs ––= ys (или xs subtractAll ys)	Удаляет все элементы ys из множества xs как побочный эффект и возвращает само множество xs
xs remove x	Удаляет элемент x из xs и возвращает true, если x прежде уже был во множестве, или false, если его прежде там не было
xs filterInPlace p	Сохраняет только те элементы в xs, которые удовлетворяют условию p
xs.clear()	Удаляет из xs все элементы
Обновление	
xs(x) = b	(Или после раскрытия xs.update(x, b)) если аргумент b типа Boolean имеет значение true, то добавляет x в xs, в противном случае удаляет x из xs
Клонирование	
xs.clone()	Возвращает новое изменяемое множество с такими же элементами, как и в xs

Операция `s += elem` в качестве побочного эффекта добавляет `elem` во множество `s` и в качестве результата возвращает измененное множество. По аналогии с этим `s -= elem` удаляет элемент `elem` из множества и возвращает в качестве результата измененное множество. Помимо `+=` и `-=`, есть также операции над несколькими элементами `++=` и `--=`, которые добавляют или удаляют все элементы `Iterable` или итератора.

Выбор в качестве имен методов `+=` и `-=` означает, что очень похожий код может работать как с изменяемыми, так и с неизменяемыми множествами. Рассмотрим сначала диалог с интерпретатором, в котором используется неизменяемое множество `s`:

```
scala> var s = Set(1, 2, 3)
s: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> s += 4; s -=2

scala> s
res10: scala.collection.immutable.Set[Int] = Set(1, 3, 4)
```

В этом примере в отношении `var`-переменной типа `immutable.Set` используются методы `+=` и `-=`. Согласно объяснениям, которые были даны на шаге 10 в главе 3, инструкции вида `s += 4` — это сокращенная форма записи для `s = s + 4`. Следовательно, в их выполнении участвует еще один метод `+`, применяемый в отношении множества `s`, а затем результат присваивается переменной `s`. А теперь рассмотрим аналогичную работу в интерпретаторе с изменяемым множеством:

```
scala> val s = collection.mutable.Set(1, 2, 3)
s: scala.collection.mutable.Set[Int] = Set(1, 2, 3)
```

```
scala> s += 4
res11: s.type = Set(1, 2, 3, 4)

scala> s -= 2
res12: s.type = Set(1, 3, 4)
```

Конечный эффект очень похож на предыдущий диалог с интерпретатором: начинаем мы с множеством `Set(1, 2, 3)`, а заканчиваем с множеством `Set(1, 3, 4)`. Но даже притом, что инструкции выглядят такими же, как и раньше, они выполняют несколько иные действия. Теперь инструкция `s += 4` вызывает метод `+=` в отношении значения `s`, которое представляет собой изменяемое множество, выполняя изменения на месте. Аналогично этому инструкция `s -= 2` теперь вызывает в отношении этого же множества метод `-=`.

Сравнение этих двух диалогов позволяет выявить весьма важный принцип. Зачастую можно заменить изменяемую коллекцию, хранящуюся в `val`-переменной, неизменяемой коллекцией, хранящейся в `var`-переменной, и *наоборот*. Это работает по крайней мере до тех пор, пока нет псевдонимов ссылок на коллекцию, позволяющих заметить, обновилась она на месте или была создана новая коллекция.

Изменяемые множества также предоставляют в качестве вариантов `+=` и `-=` методы `add` и `remove`. Разница в том, что методы `add` и `remove` возвращают булев результат, показывающий, возымела ли операция эффект над множеством.

В текущей реализации по умолчанию изменяемого множества его элементы хранятся с помощью хеш-таблицы. В реализации по умолчанию неизменяемых множеств используется представление, которое адаптируется к количеству элементов множества. Пустое множество представляется в виде простого объекта-одиночки. Множества размером до четырех элементов представляются в виде одиночного объекта, сохраняющего все элементы как поля. Все неизменяемые множества, имеющие большие размеры, реализуются в виде префиксных деревьев на основе сжатых хеш-массивов (compressed hash-array mapped prefix trie, CHAMP)[1].

Последствия применения таких вариантов представления заключаются в том, что для множеств небольших размеров с количеством элементов, не превышающим четырех, неизменяемые множества получаются более компактными и более эффективными в работе, чем изменяемые. Поэтому, если предполагается, что множество будет небольшим, попробуйте сделать его неизменяемым.

24.6. Отображения

Коллекции типа `Map` представляют собой `Iterable`-коллекции, состоящие из пар «ключ — значение», которые также называются отображениями или ассоциациями. Как говорилось в разделе 21.4, имеющийся в Scala объект `Predef` предлагает неявное преобразование, позволяющее использовать запись вида *ключ* –> *значение* в каче-

[1] Префиксные деревья на основе сжатых хеш-массивов описываются в разделе 24.7.

стве альтернативы синтаксиса для пары вида (*ключ, значение*). Таким образом, выражение `Map("x" -> 24, "y" -> 25, "z" -> 26)` означает абсолютно то же самое, что и выражение `Map(("x", 24), ("y", 25), ("z", 26))`, но читается легче.

Основные операции над отображениями, сведенные в табл. 24.8, похожи на аналогичные операции над множествами. Неизменяемые отображения поддерживают дополнительные операции добавления и удаления, которые возвращают новые отображения, как показано в табл. 24.9. Изменяемые отображения дополнительно поддерживают операции, перечисленные в табл. 24.10. Операции над отображениями разбиваются на следующие категории.

- **Операции поиска** — `apply`, `get`, `getOrElse`, `contains` и `isDefinedAt` — превращают отображения в частично примененные функции от ключей к значениям. Основной метод поиска для отображений выглядит так:

  ```
  def get(key): Option[Value]
  ```

  Операция `m get key` проверяет, содержит ли отображение ассоциацию для заданного ключа. Будучи в наличии, такая ассоциация возвращает значение ассоциации в виде объекта типа `Some`. Если такой ключ в отображении не определен, то `get` возвращает `None`. В отображениях также определяется метод `apply`, возвращающий значение, непосредственно ассоциированное с заданным ключом, без его инкапсуляции в `Option`. Если ключ в отображении не определен, то выдается исключение.
- **Операции добавления и обновления** — `+` (псевдоним: `updated`), `++` (псевдоним: `concat`) `updateWith`, и `updatedWith` — позволяют добавлять к отображению новые привязки или изменять уже существующие.
- **Операции удаления** — `-` (псевдоним: `removed`) и `--` (псевдоним: `removedAll`) — позволяют удалять привязки из отображения.
- **Операции создания подколлекций** — `keys`, `keySet`, `keysIterator`, `valuesIterator` и `values` — возвращают по отдельности ключи и значения отображений в различных формах.
- **Операции преобразования** — `filterKeys` и `mapValues` — создают новое отображение путем фильтрации и преобразования привязок существующего отображения.

Таблица 24.8. Операции в трейте Map

Что собой представляет	Результат работы
Поиск	
ms get k	Значение Option, связанное с ключом k в отображении ms, или None, если ключ не найден
ms(k)	(Или после раскрытия ms apply k) значение, связанное с ключом k в отображении ms, или выдает исключение, если ключ не найден

Продолжение

Таблица 24.8 *(продолжение)*

Что собой представляет	Результат работы
ms getOrElse (k, d)	Значение, связанное с ключом k в отображении ms, или значение по умолчанию d, если ключ не найден
ms contains k	Проверяет, содержится ли в ms отображение для ключа k
ms isDefinedAt k	То же, что и contains
Создание подколлекций	
ms.keys	Iterable-коллекция, содержащая каждый ключ, имеющийся в ms
ms.keySet	Множество, содержащее каждый ключ, имеющийся в ms
ms.keysIterator	Итератор, выдающий каждый ключ, имеющийся в ms
ms.values	Iterable-коллекция, содержащая каждое значение, связанное с ключом в ms
ms.valuesIterator	Итератор, выдающий каждое значение, связанное с ключом в ms
Преобразование	
ms.view filterKeys p	Представление отображения, содержащее только те отображения в ms, в которых ключ удовлетворяет условию p
ms.view mapValues f	Представление отображения, получающееся в результате применения функции f к каждому значению, связанному с ключом в ms

Таблица 24.9. Операции в трейте immutable.Map

Что собой представляет	Результат работы
Добавление и обновление	
ms + (k –> v) (или ms.updated(k, v))	Отображение, содержащее все ассоциации ms, а также ассоциацию k –> v ключа k со значением v
ms ++= kvs (или ms.concat(kvs))	Отображение, содержащее все ассоциации ms, а также все пары «ключ — значение» из kvs
ms.updatedWith(k)(f)	Отображение с добавлением, обновлением или удалением привязки для ключа k. Функция f принимает в качестве параметра значение, связанное в настоящий момент с ключом k (или None, если такой привязки нет), и возвращает новое значение (или None для удаления привязки)
Удаление	
ms – k (или ms removed k)	Отображение, содержащее все ассоциации ms, за исключением тех, которые относятся к ключу k
ms –– ks (или ms removedAll ks)	Отображение, содержащее все ассоциации ms, за исключением тех, ключи которых входят в ks

Таблица 24.10. Операции в трейте mutable.Map

Что собой представляет	Результат работы
Добавление и обновление	
ms(k) = v	(Или после раскрытия ms.update(k, v)) добавляет в качестве побочного эффекта ассоциацию ключа k со значением v к отображению ms, перезаписывая все ранее имевшиеся ассоциации k
ms += (k -> v)	Добавляет в качестве побочного эффекта ассоциацию ключа k со значением v к отображению ms и возвращает само отображение ms
ms ++= kvs	Добавляет в качестве побочного эффекта все ассоциации, имеющиеся в kvs, к ms и возвращает само отображение ms
ms.put(k, v)	Добавляет к ms ассоциацию ключа k со значением v и возвращает как Option любое значение, ранее связанное с k
ms getOrElseUpdate (k, d)	Если ключ k определен в отображении ms, то возвращает связанное с ним значение. В противном случае обновляет ms ассоциацией k –> d и возвращает d
ms.updateWith(k)(f)	Добавляет, обновляет или удаляет ассоциацию с ключом k. Функция f принимает в качестве параметра значение, которое в настоящий момент связано с k (или None, если такой ассоциации нет) и возвращает новое значение (или None t при удалении ассоциации)
Удаление	
ms –= k	Удаляет в качестве побочного эффекта ассоциацию с ключом k из ms и возвращает само отображение ms
ms ––= ks	Удаляет в качестве побочного эффекта все ассоциации из ms с ключами, имеющимися в ks, и возвращает само отображение ms
ms remove k	Удаляет все ассоциации с ключом k из ms и возвращает как Option любое значение, ранее связанное с k
ms filterInPlace p	Сохраняет в ms только те ассоциации, у которых ключ удовлетворяет условию p
ms.clear()	Удаляет из ms все ассоциации
Преобразование и клонирование	
ms mapValuesInPlace f	Выполняет преобразование всех связанных значений в отображении ms с помощью функции f
ms.clone()	Возвращает новое изменяемое отображение с такими же ассоциациями, как и в ms

Операции добавления и удаления для отображений — зеркальные отражения таких же операций для множеств. Неизменяемое отображение может быть преобразовано с помощью операций `+`, `-` и `updated`. Для сравнения, изменяемое

отображение `m` можно обновить «на месте» двумя способами: `m(key) = value` и `m += (key -> value)`. Изменяемые отображения также поддерживают вариант `m.put(key, value)`, который возвращает значение `Option`, содержащее то, что прежде ассоциировалось с ключом, или `None`, если ранее такой ключ в отображении отсутствовал.

Метод `getOrElseUpdate` пригодится для обращения к отображениям там, где они действуют в качестве кэша. Скажем, у вас есть весьма затратное вычисление, запускаемое путем вызова функции `f`:

```
scala> def f(x: String) = {
       println("taking my time."); Thread.sleep(100)
       x.reverse }
f: (x: String)String
```

Далее предположим, что у `f` нет побочных эффектов, поэтому ее повторный вызов с тем же самым аргументом всегда будет выдавать тот же самый результат. В таком случае можно сберечь время, сохранив ранее вычисленные привязки аргумента и результата выполнения `f` в отображении, и вычислять результат выполнения `f`, только если результат для аргумента не был найден в отображении. Можно сказать, что отображение — это *кэш* для вычислений функции `f`:

```
scala> val cache = collection.mutable.Map[String, String]()
cache: scala.collection.mutable.Map[String,String] = Map()
```

Теперь можно создать более эффективную кэшированную версию функции `f`:

```
scala> def cachedF(s: String) = cache.getOrElseUpdate(s, f(s))
cachedF: (s: String)String

scala> cachedF("abc")
taking my time.
res16: String = cba

scala> cachedF("abc")
res17: String = cba
```

Обратите внимание: второй аргумент `getOrElseUpdate` — это аргумент, передаваемый по имени. Следовательно, показанное ранее вычисление `f("abc")` выполняется лишь в том случае, если методу `getOrElseUpdate` потребуется значение его второго аргумента, что происходит именно тогда, когда его первый аргумент не найден в кэширующем отображении. Вы могли бы также непосредственно реализовать `cachedF`, используя только основные операции с отображениями, но для этого понадобится дополнительный код:

```
def cachedF(arg: String) = cache get arg match {
  case Some(result) => result
  case None =>
    val result = f(arg)
    cache(arg) = result
    result
}
```

24.7. Конкретные классы неизменяемых коллекций

В Scala на выбор предлагается множество конкретных классов неизменяемых коллекций. Друг от друга они отличаются трейтами, которые реализуют (отображения, множества, последовательности), тем, могут ли они быть бесконечными, и скоростью выполнения различных операций. Начнем с обзора наиболее востребованных типов неизменяемых коллекций.

Списки

Списки относятся к конечным неизменяемым последовательностям. Они обеспечивают постоянное время доступа к своим первым элементам, а также ко всей остальной части списка, и имеют постоянное время выполнения операций `cons` для добавления нового элемента в начало списка. Многие другие операции имеют линейную зависимость времени выполнения от длины списка. Более подробно списки рассматриваются в главах 16 и 22.

Ленивые списки

Ленивые списки похожи на списки, за исключением того, что их элементы вычисляются лениво (или отложенно). Вычисляться будут только запрошенные элементы. Поэтому ленивые списки могут быть бесконечно длинными. Во всем остальном они имеют такие же характеристики производительности, что и списки.

В то время как списки конструируются с помощью оператора `::`, ленивые конструируются с помощью похожего оператора `#::`. Пример ленивого списка, содержащего целые числа `1`, `2` и `3`, выглядит следующим образом:

```
scala> val str = 1 #:: 2 #:: 3 #:: LazyList.empty
str: scala.collection.immutable.LazyList[Int] =
  LazyList(<not computed>)
```

«Голова» этого ленивого списка — `1`, а «хвост» — `2` и `3`. Но никакие элементы здесь не выводятся, поскольку список еще не вычислен! Ленивые списки объявлены как вычисляющиеся лениво, и метод `toString`, вызванный в отношении такого списка, заботится о том, чтобы не навязывать лишние вычисления.

Далее показан более сложный пример. В нем вычисляется ленивый список, содержащий последовательность чисел Фибоначчи с заданными двумя числами. Каждый элемент последовательности Фибоначчи есть сумма двух предыдущих элементов в серии:

```
scala> def fibFrom(a: Int, b: Int): LazyList[Int] =
         a #:: fibFrom(b, a + b)
fibFrom: (a: Int, b: Int)LazyList[Int]
```

Эта функция кажется обманчиво простой. Понятно, что первый элемент последовательности — `a`, за ним стоит последовательность Фибоначчи, начинающаяся с `b`, затем — элемент со значением `a + b`. Самое сложное здесь — вычислить данную последовательность, не вызвав бесконечную рекурсию. Если функция вместо `#::` использует оператор `::`, то каждый вызов функции будет приводить к еще одному вызову, что выльется в бесконечную рекурсию. Но поскольку применяется оператор `#::`, правая часть выражения не вычисляется до тех пор, пока не будет востребована.

Вот как выглядят первые несколько элементов последовательности Фибоначчи, начинающейся с двух заданных чисел:

```
scala> val fibs = fibFrom(1, 1).take(7)
fibs: scala.collection.immutable.LazyList[Int] =
  LazyList(<not computed>)

scala> fibs.toList
res23: List[Int] = List(1, 1, 2, 3, 5, 8, 13)
```

Неизменяемые ArraySeq

Списки очень эффективны в алгоритмах, которые работают исключительно с начальными элементами. Получение, добавление и удаление начала списка занимает постоянное время. А вот получение или изменения последующих элементов — это операции линейного времени, зависящие от длины списка. В результате список может быть не самым оптимальным вариантом для алгоритмов, которые не ограничиваются обработкой начальных элементов.

Последовательный массив (`ArraySeq`) — это неизменяемая последовательность на основе приватного массива, которая решает проблему неэффективного произвольного доступа в списках. Последовательные массивы позволяют получить любой элемент коллекции за постоянное время. Благодаря этому вы можете не ограничиваться работой только с начальными элементами. Время получения элемента не зависит от его местоположения, и потому `ArraySeq` может оказаться эффективней списков в некоторых алгоритмах.

С другой стороны, тип `ArraySeq` основан на `Array`, поэтому добавление элементов в начало занимает линейное время, а не постоянное, как в случае со списками. Более того, на всякое добавление или обновление одного элемента в `ArraySeq` уходит линейное время, поскольку при этом копируется весь внутренний массив.

Векторы

`List` и `ArraySeq` — структуры данных, эффективные для одних вариантов использования и неэффективные для других. Например, добавление элемента в начало `List` занимает постоянное время, а в `ArraySeq` — линейное. С другой стороны, индексированный доступ занимает постоянное время в `ArraySeq` и линейное в `List`.

Вектор обеспечивает хорошую производительность для всех своих операций. Доступ и обновление любых элементов вектора занимает «эффективно постоянное время», как описано ниже. Эти операции выполняются медленнее, чем получение начала списка или чтение элементов в последовательном массиве, но время их выполнения остается постоянным. В результате алгоритмы, использующие векторы, не должны ограничиваться доступом или обновлением только к началу последовательности. Они могут обращаться к элементам и обновлять их в произвольном месте и поэтому могут быть гораздо более удобными в написании.

Создаются и изменяются векторы точно так же, как и любые другие последовательности:

```
scala> val vec = scala.collection.immutable.Vector.empty
vec: scala.collection.immutable.Vector[Nothing] = Vector()

scala> val vec2 = vec :+ 1 :+ 2
vec2: scala.collection.immutable.Vector[Int] = Vector(1, 2)

scala> val vec3 = 100 +: vec2
vec3: scala.collection.immutable.Vector[Int] = Vector(100, 1, 2)

scala> vec3(0)
res24: Int = 100
```

Для представления векторов используются широкие неглубокие деревья. Каждый узел дерева содержит до 32 элементов вектора или до 32 других узлов дерева. Векторы, содержащие до 32 элементов, могут быть представлены одним узлом. Векторы с количеством элементов до $32 \cdot 32 = 1024$ могут быть представлены в виде одного направления. Двух переходов от корня дерева к конечному элементу достаточно для векторов, имеющих до 2^{15} элементов, трех — для векторов с 2^{20}, четырех — для векторов с 2^{25} элементов, а пяти — для векторов с количеством элементов до 2^{30}. Следовательно, для всех векторов разумного размера выбор элемента требует до пяти обычных доступов к массиву. Именно это мы имели в виду, когда написали «эффективно постоянное время».

Векторы неизменяемы, следовательно, внести в элемент вектора изменение на месте невозможно. Но метод `updated` позволяет создать новый вектор, отличающийся от заданного только одним элементом:

```
scala> val vec = Vector(1, 2, 3)
vec: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)

scala> vec updated (2, 4)
res25: scala.collection.immutable.Vector[Int] = Vector(1, 2, 4)

scala> vec
res26: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)
```

Последняя строка данного кода показывает, что вызов метода `updated` никак не повлиял на исходный вектор `vec`. Функциональное обновление векторов также занимает «эффективно постоянное время». Обновить элемент в середине вектора

можно с помощью копирования узла, содержащего элемент, и каждого указывающего на него узла, начиная с корня дерева. Это значит, функциональное обновление создает от одного до пяти узлов, каждый из которых содержит до 32 элементов или поддеревьев. Конечно, это гораздо затратнее обновления на месте в изменяемом массиве, но все же намного дешевле копирования всего вектора.

Векторы сохраняют разумный баланс между быстрым произвольным доступом и быстрыми произвольными функциональными обновлениями, поэтому в настоящий момент они представляют исходную реализацию неизменяемых индексированных последовательностей:

```
scala> collection.immutable.IndexedSeq(1, 2, 3)
res27: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 2, 3)
```

Неизменяемые очереди

В очередях используется принцип «первым пришел, первым вышел». Упрощенная реализация неизменяемых очередей рассматривалась в главе 19. Создать пустую неизменяемую очередь можно следующим образом:

```
scala> val empty = scala.collection.immutable.Queue[Int]()
empty: scala.collection.immutable.Queue[Int] = Queue()
```

Добавить элемент в неизменяемую очередь можно с помощью метода `enqueue`:

```
scala> val has1 = empty.enqueue(1)
has1: scala.collection.immutable.Queue[Int] = Queue(1)
```

Чтобы добавить в очередь сразу несколько элементов, следует вызвать метод `enqueueAll` с коллекцией в качестве его аргументов:

```
scala> val has123 = has1.enqueueAll(List(2, 3))
has123: scala.collection.immutable.Queue[Int] = Queue(1, 2, 3)
```

Чтобы удалить элемент из начала очереди, следует воспользоваться методом `dequeue`:

```
scala> val (element, has23) = has123.dequeue
element: Int = 1
has23: scala.collection.immutable.Queue[Int] = Queue(2, 3)
```

Обратите внимание: `dequeue` возвращает пару, состоящую из удаленного элемента и оставшейся части очереди.

Диапазоны

Диапазон представляет собой упорядоченную последовательность целых чисел с одинаковым интервалом между ними. Например, 1, 2, 3 является диапазоном точно так же, как и 5, 8, 11, 14. Чтобы создать в Scala диапазон, следует воспользоваться предопределенными методами `to` и `by`. Рассмотрим несколько примеров:

```
scala> 1 to 3
res31: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3)

scala> 5 to 14 by 3
res32: scala.collection.immutable.Range = Range(5, 8, 11, 14)
```

Если нужно создать диапазон, исключая его верхний предел, то следует вместо метода `to` воспользоваться методом-помощником `until`:

```
scala> 1 until 3
res33: scala.collection.immutable.Range = Range(1, 2)
```

Диапазоны представлены постоянным объемом памяти, поскольку могут быть определены всего тремя числами: их началом, их концом и значением шага. Благодаря такому представлению большинство операций над диапазонами выполняется очень быстро.

Сжатые коллекции HAMT

Хеш-извлечения (hash-tries[1]) — стандартный способ эффективной реализации неизменяемых множеств и отображений. Сжатые коллекции HAMT[2] — разновидность хеш-извлечений в `JVM`, которая оптимизирует размещение элементов и следит за тем, чтобы деревья были представлены каноничным и компактным образом.

Их представление похоже на векторы тем, что в них также используются деревья, где у каждого узла имеется 32 элемента или 32 поддерева, но выбор осуществляется на основе хеш-кода. Например, самые младшие пять разрядов хеш-кода ключа используются в целях поиска заданного ключа в отображении для выбора первого поддерева, следующие пять — для выбора следующего поддерева и т. д. Процесс выбора прекращается, как только у всех элементов, хранящихся в узле, будет хеш-код, отличающийся от всех остальных в разрядах, выбранных до сих пор. Таким образом, не обязательно используются все разряды хеш-кода.

Хеш-извлечения позволяют достичь хорошего баланса между достаточно быстрым поиском и достаточно эффективными функциональными вставками (`+`) и удалениями (`-`). Именно поэтому в Scala они положены в основу реализаций по умолчанию неизменяемых отображений и множеств. Фактически для неизменяемых множеств и отображений, содержащих менее пяти элементов, в Scala предусматривается дальнейшая оптимизация. Множества и отображения, содержащие от одного до четырех элементов, хранятся как отдельные объекты, содержащие эти элементы (или в случае с отображениями — пары «ключ — значение») в виде полей. Пустое изменяемое множество и пустое изменяемое отображение во всех случаях являются объектами-одиночками — не нужно дублировать для них место хранения, поскольку пустые неизменяемые множество или отображение всегда будут оставаться пустыми.

---

1 Trie происходит от слова retrieval («извлечение») и произносится как «три» или «трай».

2 *Steindorfer M. J., Vinju J. J.* Optimizing hash-array mapped tries for fast and lean immutable JVM collections // ACM SIGPLAN Notices. Vol. 50. ACM, 2015. P. 783–800.

Красно-черные деревья

Форма сбалансированных двоичных деревьев — красно-черные деревья, в которых одни узлы обозначены как красные, а другие — как черные. Операции над ними, как и над любыми другими сбалансированными двоичными деревьями, гарантированно завершаются за время, экспоненциально зависящее от размера дерева.

Scala предоставляет реализации множеств и отображений, во внутреннем представлении которых используется красно-черное дерево. Доступ к ним осуществляется по именам `TreeSet` и `TreeMap`:

```
scala> val set = collection.immutable.TreeSet.empty[Int]
set: scala.collection.immutable.TreeSet[Int] = TreeSet()

scala> set + 1 + 3 + 3
res34: scala.collection.immutable.TreeSet[Int] = TreeSet(1, 3)
```

Кроме того, красно-черные деревья в Scala — стандартный механизм реализации `SortedSet`, поскольку предоставляют весьма эффективный итератор, возвращающий все элементы множества в отсортированном виде.

Неизменяемые битовые множества

Битовое множество представляет коллекцию небольших целых чисел в виде битов большего целого числа. Например, битовое множество, содержащее 3, 2 и 0, будет представлено в двоичном виде как целое число 1101, которое в десятичном виде является числом 13.

Во внутреннем представлении битовые множества используют массив из 64-разрядных `Long`-значений. Первое `Long`-значение в массиве предназначено для целых чисел от 0 до 63, второе — от 64 до 127 и т. д. Таким образом, битовые множества очень компактны, поскольку самое большое целое число во множестве чуть меньше нескольких сотен или около того.

Операции над битовыми множествами выполняются очень быстро. Проверка на присутствие занимает постоянное время. На добавление элемента во множество уходит время, пропорциональное количеству `Long`-значений в массиве битового множества, которых обычно немного. Некоторые примеры использования битового множества выглядят следующим образом:

```
scala> val bits = scala.collection.immutable.BitSet.empty
bits: scala.collection.immutable.BitSet = BitSet()

scala> val moreBits = bits + 3 + 4 + 4
moreBits: scala.collection.immutable.BitSet = BitSet(3, 4)

scala> moreBits(3)
res35: Boolean = true
```

```
scala> moreBits(0)
res36: Boolean = false
```

Векторные отображения

`VectorMap` (векторное отображение) представляет отображение с использованием как вектора (для ключей), так и `HashMap`. Оно предоставляет итератор, возвращающий все элементы в том порядке, в котором они были вставлены.

```
scala> val vm = scala.collection.immutable.VectorMap.empty[Int, String]
vm: scala.collection.immutable.VectorMap[Int,String] = VectorMap()

scala> val vm1 = vm + (1 -> "one")
vm1: scala.collection.immutable.VectorMap[Int,String] =
  VectorMap(1 -> one)

scala> val vm2 = vm1 + (2 -> "two")
vm2: scala.collection.immutable.VectorMap[Int,String] =
  VectorMap(1 -> one, 2 -> two)

scala> vm2 == Map(2 -> "two", 1 -> "one")
res29: Boolean = true
```

В первых строчках видно, что содержимое `VectorMap` сохраняет порядок вставки, а последняя строчка показывает, что векторные отображения можно сравнивать с другими видами отображений и в ходе этого сравнения не учитывается порядок следования элементов.

Списочные отображения

Списочное отображение представляет собой отображение в виде связанного списка пар «ключ — значение». Как правило, операции над списочными отображениями могут потребовать перебора всего списка. В связи с этим такие операции занимают время, пропорциональное размеру отображения. Списочные отображения не нашли в Scala широкого применения, поскольку работа со стандартными неизменяемыми отображениями почти всегда выполняется быстрее. Единственно возможное положительное отличие наблюдается в том случае, если по какой-то причине отображение сконструировано так, что первые элементы списка выбираются гораздо чаще других элементов.

```
scala> val map = collection.immutable.ListMap(
         1 -> "one", 2 -> "two")
map: scala.collection.immutable.ListMap[Int,String] = Map(1
-> one, 2 -> two)

scala> map(2)
res37: String = "two"
```

24.8. Конкретные классы изменяемых коллекций

Изучив наиболее востребованные из имеющихся в стандартной библиотеке Scala классы неизменяемых коллекций, рассмотрим классы изменяемых коллекций.

Буферы массивов

Буферы массивов уже встречались нам в разделе 17.1. В таком буфере содержатся массив и его размер. Большинство операций над буферами массивов выполняются с такой же скоростью, что и над массивами, поскольку эти операции просто обращаются к обрабатываемому массиву и вносят в него изменения. Кроме того, у буфера массива могут быть данные, весьма эффективно добавляемые в его конец. Добавление записи в буфер массива занимает амортизированно постоянное время. Из этого следует, что буферы массивов хорошо подходят для эффективного создания больших коллекций, когда новые элементы всегда добавляются в конец. Вот как выглядят некоторые примеры их применения:

```
scala> val buf = collection.mutable.ArrayBuffer.empty[Int]
buf: scala.collection.mutable.ArrayBuffer[Int]
  = ArrayBuffer()

scala> buf += 1
res38: buf.type = ArrayBuffer(1)

scala> buf += 10
res39: buf.type = ArrayBuffer(1, 10)

scala> buf.toArray
res40: Array[Int] = Array(1, 10)
```

Буферы списков

В разделе 17.1 нам встречались и буферы списков. Буфер списка похож на буфер массива, за исключением того, что внутри него используется не массив, а связанный список. Если сразу после создания буфер предполагается превращать в список, то лучше воспользоваться буфером списка, а не буфером массива. Вот как выглядит соответствующий пример[1]:

```
scala> val buf = collection.mutable.ListBuffer.empty[Int]
buf: scala.collection.mutable.ListBuffer[Int] = ListBuffer()
```

[1] Появляющийся в ответах интерпретатора в этом и в нескольких других примерах раздела buf.type является синглтон-типом. Как будет сказано в разделе 29.6 , buf.type означает, что переменная содержит именно тот объект, на который указывает buf.

```
scala> buf += 1
res41: buf.type = ListBuffer(1)

scala> buf += 10
res42: buf.type = ListBuffer(1, 10)

scala> buf.toList
res43: List[Int] = List(1, 10)
```

Построители строк

По аналогии с тем, что буфер массива используется для создания массивов, а буфер списка — для создания списков, построитель строк применяется для создания строк. Построители строк используются настолько часто, что заранее импортируются в пространство имен по умолчанию. Создать их можно с помощью простого выражения `new StringBuilder`:

```
scala> val buf = new StringBuilder
buf: StringBuilder =

scala> buf += 'a'
res44: buf.type = a

scala> buf ++= "bcdef"
res45: buf.type = abcdef

scala> buf.toString
res46: String = abcdef
```

ArrayDeque

`ArrayDeque` — изменяемая последовательность, которая поддерживает эффективное добавление элементов в начало и в конец. Внутри она использует массив изменяемого размера. Если вам нужно вставлять элементы в начало или в конец буфера, то используйте `ArrayDeque` вместо `ArrayBuffer`.

Очереди

Наряду с неизменяемыми очередями в Scala есть и изменяемые. Используются они аналогично, но для добавления элементов вместо `enqueue` применяются операторы `+=` и `++=`. Кроме того, метод `dequeue` будет просто удалять из изменяемой очереди головной элемент и возвращать его. Примеры использования даны ниже:

```
scala> val queue = new scala.collection.mutable.Queue[String]
queue: scala.collection.mutable.Queue[String] = Queue()
```

```
scala> queue += "a"
res47: queue.type = Queue(a)

scala> queue ++= List("b", "c")
res48: queue.type = Queue(a, b, c)

scala> queue
res49: scala.collection.mutable.Queue[String] = Queue(a, b, c)

scala> queue.dequeue
res50: String = a

scala> queue
res51: scala.collection.mutable.Queue[String] = Queue(b, c)
```

Стеки

Scala предоставляет изменяемый стек. Ниже представлен пример:

```
scala> val stack = new scala.collection.mutable.Stack[Int]
stack: scala.collection.mutable.Stack[Int] = Stack()

scala> stack.push(1)
res52: stack.type = Stack(1)

scala> stack
res53: scala.collection.mutable.Stack[Int] = Stack(1)

scala> stack.push(2)
res54: stack.type = Stack(2, 1)

scala> stack
res55: scala.collection.mutable.Stack[Int] = Stack(2, 1)

scala> stack.top
res56: Int = 2

scala> stack
res57: scala.collection.mutable.Stack[Int] = Stack(2, 1)

scala> stack.pop
res58: Int = 2

scala> stack
res59: scala.collection.mutable.Stack[Int] = Stack(1)
```

Обратите внимание: в Scala нет поддержки неизменяемых стеков, поскольку данная функциональность уже реализована в списках. Операция `push` над неизменяемым стеком аналогична выражению `a ::` для списка. Операция `pop` — то же самое, что вызвать `head` и `tail` для списка.

Изменяемые ArraySeq

Последовательный массив — это изменяемая последовательность фиксированного размера, которая хранит свои элементы внутри `Array[AnyRef]`. В Scala он реализован в виде класса `ArraySeq`.

Данный класс обычно используется в случаях, когда вам нужен производительный массив и притом необходимо создавать обобщенные экземпляры последовательности с элементами, тип которых не известен заранее и не может быть получен во время выполнения с помощью `ClassTag`. С этими проблемами, которые присущи массивам, вы познакомитесь чуть позже, в разделе 24.9.

Хеш-таблицы

Хеш-таблица сохраняет свои элементы в образующем ее массиве, помещая каждый в позицию в массиве, определяемую хеш-кодом этого элемента. На добавление элемента в хеш-таблицу всегда уходит одно и то же время, если только в массиве нет еще одного элемента с точно таким же хеш-кодом. Поэтому, пока помещенные в хеш-таблицу элементы имеют хорошее распределение хеш-кодов, работа с ней выполняется довольно быстро. По этой причине типы изменяемых отображений и множеств по умолчанию в Scala основаны на хеш-таблицах.

Хеш-множества и хеш-отображения используются точно так же, как и любые другие множества или отображения. Ниже представлены некоторые простые примеры:

```
scala> val map = collection.mutable.HashMap.empty[Int,String]
map: scala.collection.mutable.HashMap[Int,String] = Map()

scala> map += (1 -> "make a web site")
res60: map.type = Map(1 -> make a web site)

scala> map += (3 -> "profit!")
res61: map.type = Map(1 -> make a web site, 3 -> profit!)

scala> map(1)
res62: String = make a web site

scala> map contains 2
res63: Boolean = false
```

Конкретный порядок обхода элементов хеш-таблицы не гарантируется. Выполняется простой обход элементов массива, на котором основана хеш-таблица, в порядке расположения его элементов. Чтобы получить гарантированный порядок обхода, следует воспользоваться *связанными* хеш-отображением или множеством вместо обычных. Связанные хеш-отображение или множество почти аналогичны обычным хеш-отображению или множеству, но включают связанный список элементов в порядке их добавления. Обход элементов такой коллекции всегда выполняется в том же порядке, в котором они добавлялись в нее изначально.

Слабые хеш-отображения

Слабое хеш-отображение представляет собой особую разновидность хеш-отображения, в которой сборщик мусора не следует по ссылкам от отображения к хранящимся в нем ключам. Это значит, ключ и связанное с ним значение исчезнут из отображения, если на данный ключ нет другой ссылки. Слабые хеш-отображения используются для решения таких задач, как кэширование, когда нужно повторно задействовать результат затратной функции в случае повторного вызова функции в отношении того же самого ключа. Если ключи и результаты функции хранятся в обычном хеш-отображении, то оно может бесконечно разрастись и никакие ключи никогда не станут мусором. Этой проблемы удается избежать с помощью слабого хеш-отображения. Как только объект ключа становится недоступным, связанная с ним запись удаляется из такого отображения. Слабые хеш-отображения реализованы в Scala в виде обертки положенной в их основу Java-реализации `java.util.WeakHashMap`.

Совместно используемые отображения

К совместно используемому отображению могут обращаться сразу несколько потоков. В дополнение к обычным операциям с `Map` это отображение предоставляет атомарные операции (табл. 24.11).

Таблица 24.11. Операции в трейте concurrent.Map

Что собой представляет	Результат работы
m.putIfAbsent(k, v)	Добавляет привязку «ключ — значение» k –> v, кроме тех случаев, когда k уже определен в m
m.remove (k, v)	Удаляет запись для ключа k, если он в данный момент отображен на значение v
m.replace (k, old, new)	Заменяет значение, связанное с k, новым значением new, если ранее с ключом было связано значение old
m.replace (k, v)	Заменяет значение, связанное с k, значением v, если ранее этот ключ был связан с каким-либо значением

Трейт `scala.concurrent.Map` определяет интерфейс для изменяемых отображений с поддержкой конкурентного доступа. Стандартная библиотека предлагает две реализации этого трейта. Первая — это `java.util.concurrent.ConcurrentMap` из Java, которую можно автоматически превратить в отображение языка Scala, используя стандартные для Java/Scala операции приведения коллекций (эти преобразования будут описаны в разделе 24.16). Вторая реализация, `TrieMap`, основана на HAMT без блокировок.

Изменяемые битовые множества

Изменяемое битовое множество похоже на неизменяемое, но отличается тем, что может быть изменено на месте. По сравнению с неизменяемым работает при обновлениях немного эффективнее, поскольку неизмененные `Long`-значения копировать не нужно. Пример использования выглядит так:

```
scala> val bits = scala.collection.mutable.BitSet.empty
bits: scala.collection.mutable.BitSet = BitSet()

scala> bits += 1
res64: bits.type = BitSet(1)

scala> bits += 3
res65: bits.type = BitSet(1, 3)

scala> bits
res66: scala.collection.mutable.BitSet = BitSet(1, 3)
```

24.9. Массивы

Массивы в Scala — особая разновидность коллекции. С одной стороны, массивы Scala в точности соответствуют массивам Java. То есть Scala-массив `Array[Int]` представлен как Java-массив `int[]`, `Array[Double]` — как `double[]`, а `Array[String]` — как `String[]`. Но вместе с тем массивы Scala предоставляют гораздо больше, чем их Java-аналоги. Во-первых, массивы Scala могут быть *обобщенными*. То есть можно воспользоваться массивом `Array[T]`, где `T` является параметром типа или абстрактным типом. Во-вторых, массивы Scala совместимы со Scala-последовательностями, то есть туда, где требуется `Seq[T]`, можно передавать `Array[T]`. И наконец, массивы Scala также поддерживают все операции с последовательностями. Приведем несколько практических примеров:

```
scala> val a1 = Array(1, 2, 3)
a1: Array[Int] = Array(1, 2, 3)

scala> val a2 = a1 map (_ * 3)
a2: Array[Int] = Array(3, 6, 9)

scala> val a3 = a2 filter (_ % 2 != 0)
a3: Array[Int] = Array(3, 9)

scala> a3.reverse
res1: Array[Int] = Array(9, 3)
```

Если учесть, что массивы Scala представлены точно так же, как массивы Java, то как эти дополнительные свойства могут поддерживаться в Scala?

Ответ заключается в систематическом использовании неявных преобразований. Массив не может претендовать на то, чтобы *быть* последовательностью, поскольку тип данных, представляющих настоящий массив, не является подтипом типа `Seq`. Вместо этого там, где массив будет использоваться в качестве последовательности `Seq`, он будет неявно обернут в подкласс класса `Seq`. Имя этого подкласса — `scala.collection.mutable.ArraySeq`. Вот как это работает:

```
scala> val seq: collection.Seq[Int] = a1
seq: scala.collection.Seq[Int] = ArraySeq(1, 2, 3)

scala> val a4: Array[Int] = seq.toArray
a4: Array[Int] = Array(1, 2, 3)

scala> a1 eq a4
res2: Boolean = false
```

Сеанс работы с интерпретатором показывает, что массивы совместимы с последовательностями благодаря неявному преобразованию из `Array` в `ArraySeq`. Преобразование в обратном направлении, из `ArraySeq` в `Array`, можно выполнить с использованием метода `toArray`, который определен в классе `Iterable`. В последней строке приведенного ранее диалога с интерпретатором показано, что заключенный в оболочку массив при его изъятии оттуда с помощью метода `toArray` возвращает копию исходного массива.

Есть еще одно неявное преобразование, применимое к массивам. Оно просто добавляет к массивам все методы, применимые к последовательностям, но сами массивы в последовательности не превращает. «Добавляет» означает, что массив заворачивается в другой объект типа `ArrayOps`, который поддерживает все методы работы с последовательностями. Обычно этот `ArrayOps`-объект существует весьма непродолжительное время — он становится недоступен после вызова метода для работы с последовательностью, и его место хранения может быть использовано повторно. Современные виртуальные машины зачастую вообще избегают создания таких объектов.

Разница между этими двумя неявными преобразованиями массивов показана в следующем примере:

```
scala> val seq: collection.Seq[Int] = a1
seq: scala.collection.Seq[Int] = ArraySeq(1, 2, 3)

scala> seq.reverse
res2: scala.collection.Seq[Int] = ArraySeq(3, 2, 1)

scala> val ops: collection.ArrayOps[Int] = a1
ops: scala.collection.ArrayOps[Int] = ...

scala> ops.reverse
res3: Array[Int] = Array(3, 2, 1)
```

Как видите, при вызове метода reverse в отношении объекта seq типа ArraySeq опять будет получен объект типа ArraySeq. Это не противоречит здравому смыслу, поскольку массивы ArraySeq, заключенные в оболочку, относятся к типам Seq, а вызов метода reverse в отношении любого Seq-объекта вновь даст Seq-объект. В то же время вызов метода reverse в отношении значения ops класса ArrayOps приведет к возвращению значения типа Array, а не Seq.

Приведенный ранее пример с ArrayOps был надуманным, предназначенным лишь для демонстрации разницы с ArraySeq. В обычной ситуации вы бы не стали определять значение класса ArrayOps, а просто вызвали бы в отношении массива метод из класса Seq:

```
scala> a1.reverse
res4: Array[Int] = Array(3, 2, 1)
```

Объект ArrayOps вставляется автоматически в ходе неявного преобразования. Следовательно, показанная выше строка кода эквивалентна следующей строке, где метод intArrayOps является преобразованием, которое неявно вставлялось в предыдущем примере:

```
scala> intArrayOps(a1).reverse
res5: Array[Int] = Array(3, 2, 1)
```

Возникает вопрос, касающийся способа выбора компилятором intArrayOps в показанной выше строке среди других неявных преобразований в ArraySeq. Ведь оба преобразования отображают массив на тип, поддерживающий метод reverse, указанный во введенном коде. Ответ на данный вопрос — уровни приоритета этих двух преобразований. У преобразования ArrayOps более высокий приоритет, чем у ArraySeq. Первое преобразование определено в объекте Predef, а второе — в классе scala.LowPriorityImplicits, являющемся суперклассом для Predef. Неявные преобразования в подклассах и подобъектах имеют приоритет над неявными преобразованиями в базовых классах. Следовательно, если применимы оба преобразования, то будет выбрано то, которое определено в Predef. Очень похожая схема, рассмотренная в разделе 21.7, работает для строк.

Теперь вы знаете, что массивы совместимы с последовательностями и могут поддерживать все операции, применяемые к последовательностям. А как насчет обобщенности? В Java вы не можете воспользоваться записью T[], где T представляет собой параметр типа. А как же тогда представлен имеющийся в Scala тип Array[T]? Фактически такой обобщенный массив, как Array[T], может во время выполнения программы стать любым из восьми примитивных типов массивов Java: byte[], short[], char[], int[], long[], float[], double[], boolean[] — или же стать массивом объектов. Единственный общий тип, охватывающий во время выполнения программы все эти типы, — AnyRef (или его аналог java.lang.Object), следовательно, это именно тот тип, на который компилятор Scala отображает Array[T]. Во время выполнения программы, когда происходит обращение к элементу массива

типа `Array[T]` или обновление этого элемента, производится ряд проверок на соответствие типам. Благодаря этому определяется действительный тип массива, а затем выполняется корректная операция уже над Java-массивом. Проверки на соответствие типам несколько замедляют операции над массивами. Можно ожидать, что обращения к обобщенным массивам будут в три-четыре раза медленнее обращений к простым массивам или массивам объектов. Следовательно, если требуется максимально высокая производительность, то предпочтение следует отдавать конкретным, а не обобщенным массивам.

Но одного представления типа обобщенного массива недостаточно, должен существовать также способ *создания* обобщенных массивов. А это еще более сложная задача, которая требует от вас оказать некоторую помощь. В качестве примера рассмотрим попытку создания метода, работающего с обобщениями и создающего массив:

```
// Неправильно!
def evenElems[T](xs: Vector[T]): Array[T] = {
  val arr = new Array[T]((xs.length + 1) / 2)
  for (i <- 0 until xs.length by 2)
    arr(i / 2) = xs(i)
  arr
}
```

Метод `evenElems` возвращает новый массив, состоящий из тех элементов используемого в качестве аргумента вектора `xs`, которые находятся в нем на четных позициях. В первой строке тела метода `evenElems` создается получаемый в результате массив, имеющий тот же тип элементов, что и аргумент. Следовательно, в зависимости от фактического параметра типа для `T` это может быть `Array[Int]`, или `Array[Boolean]`, или массив из других примитивных типов Java, или же массив какого-нибудь ссылочного типа. Но все эти типы имеют во время выполнения программы различные представления, поэтому возникает вопрос: как среда выполнения Scala собирается выбирать из них нужное? По сути, сделать это, основываясь на имеющейся информации, она не может, поскольку фактический тип, который соответствует параметру типа `T`, во время выполнения кода затирается. Поэтому при попытке скомпилировать показанный ранее код будет получено следующее сообщение об ошибке:

```
error: cannot find class tag for element type T
  val arr = new Array[T]((arr.length + 1) / 2)
```

Здесь вам следует помочь компилятору, предоставив подсказку времени выполнения о том, какой параметр типа у `evenElems`. Эта подсказка принимает форму *тега класса* типа `scala.reflect.ClassTag`. Тег класса описывает заданный *стираемый тип*, предоставляя исчерпывающую информацию о нем конструктору массива.

Во многих случаях компилятор может создавать тег класса самостоятельно. Именно так обстоит дело с конкретными типами наподобие `Int` или `String`. То же распространяется и на некоторые обобщенные типы наподобие `List[T]`, где для

выстраивания предположения о стираемом типе информации вполне достаточно; в данном примере стираемым типом будет `List`.

Для полностью обобщенных случаев обычно практикуется передача тега класса с помощью контекстного ограничителя, рассмотренного в разделе 21.6. А вот как, используя этот ограничитель, можно исправить показанное ранее определение:

```
// Этот код работает
import scala.reflect.ClassTag
def evenElems[T: ClassTag](xs: Vector[T]): Array[T] = {
  val arr = new Array[T]((xs.length + 1) / 2)
  for (i <- 0 until xs.length by 2)
    arr(i / 2) = xs(i)
  arr
}
```

В этом новом определении компилятор при создании `Array[T]` ищет тег класса для параметра типа `T`, то есть будет искать неявное значение типа `ClassTag[T]`. Если такое значение будет найдено, то тег класса будет использован для создания массива нужного вида. В противном случае вы увидите сообщение об ошибке, похожее на показанное ранее.

Вот как выглядит диалог с интерпретатором, в котором используется метод `evenElems`:

```
scala> evenElems(Vector(1, 2, 3, 4, 5))
res6: Array[Int] = Array(1, 3, 5)

scala> evenElems(Vector("this", "is", "a", "test", "run"))
res7: Array[java.lang.String] = Array(this, a, run)
```

В обоих случаях компилятор Scala автоматически создает тег класса для типа элемента (сначала `Int`, потом `String`) и передает его неявному параметру метода `evenElems`. Компилятор может сделать то же самое для всех конкретных типов, но не способен на это, если сам аргумент является еще одним параметром типа без признака класса. Например, следующий код не пройдет компиляцию:

```
scala> def wrap[U](xs: Vector[U]) = evenElems(xs)
<console>:9: error: No ClassTag available for U
     def wrap[U](xs: Vector[U]) = evenElems(xs)
                                          ^
```

Здесь метод `evenElems` получает тег класса для параметра типа `U`, но ничего не находит. Разумеется, решение в данном случае — потребовать еще один неявный тег класса для `U`. Код, представленный ниже, уже пройдет компиляцию:

```
scala> def wrap[U: ClassTag](xs: Vector[U]) = evenElems(xs)
wrap: [U](xs: Vector[U])(implicit evidence$1:
     scala.reflect.ClassTag[U])Array[U]
```

Этот пример показывает также, что контекстное ограничение в определении `U` — краткая форма неявного параметра, названного здесь `evidence$1` и имеющего тип `ClassTag[U]`.

24.10. Строки

Как и массивы, строки не являются последовательностями в прямом смысле слова, но могут быть в них преобразованы и вдобавок поддерживают все операции с последовательностями. Ниже приводятся примеры операций, которые могут вызываться в отношении строк:

```
scala> val str = "hello"
str: java.lang.String = hello

scala> str.reverse
res6: String = olleh

scala> str.map(_.toUpper)
res7: String = HELLO

scala> str drop 3
res8: String = lo

scala> str.slice(1, 4)
res9: String = ell

scala> val s: Seq[Char] = str
s: Seq[Char] = hello
```

Эти операции поддерживаются двумя неявными преобразованиями, рассмотренными в разделе 21.7. Первое, имеющее более низкий уровень приоритета, отображает класс `String` на класс `WrappedString`, являющий подклассом `immutable.IndexedSeq`. Это преобразование было применено в последней строке предыдущего примера, в котором строка была преобразована в значение типа `Seq`. Другое преобразование, с более высоким уровнем приоритета, отображает строку на объект `StringOps`, который добавляет к строкам все методы, применяемые к неизменяемым последовательностям. В предыдущем примере это преобразование было неявно вставлено в вызовы методов `reverse`, `map`, `drop` и `slice`.

24.11. Характеристики производительности

Как показали все предыдущие разъяснения, разным типам коллекций свойственны различные характеристики производительности. Именно это обстоятельство становится главной причиной выбора конкретного типа коллекции из множества других типов. Характеристики производительности некоторых наиболее востребованных операций над коллекциями сведены в табл. 24.12 и 24.13.

Таблица 24.12. Характеристики производительности типов последовательностей

Операции	head	tail	apply	update	prepend	append	insert
Неизменяемые							
List	C	C	L	L	C	L	—
LazyList	C	C	L	L	C	L	—
ArraySeq	C	L	C	L	L	L	—
Vector	eC	eC	eC	eC	eC	eC	—
Queue	aC	aC	L	L	L	C	—
Range	C	C	C	—	—	—	—
String	C	L	C	L	L	L	—
Изменяемые							
ArrayBuffer	C	L	C	C	L	aC	L
ListBuffer	C	L	L	L	C	C	L
StringBuilder	C	L	C	C	L	aC	L
Queue	C	L	L	L	C	C	L
ArraySeq	C	L	C	C	—	—	—
Stack	C	L	L	L	C	L	L
Array	C	L	C	C	—	—	—
ArrayDeque	C	L	C	C	aC	aC	L

Таблица 24.13. Характеристики производительности типов множеств и отображений

Операции	lookup	add	remove	min
Неизменяемые				
HashSet/HashMap	eC	eC	eC	L
TreeSet/TreeMap	Log	Log	Log	Log
BitSet	C	L	L	eC[a]
VectorMap	eC	eC	aC	L
ListMap	L	L	L	L
Изменяемые				
HashSet/HashMap	eC	eC	eC	L
WeakHashMap	eC	eC	eC	L
BitSet	C	aC	C	eC[a]

[a] Если биты плотно упакованы.

Записи в этих двух таблицах расшифровываются следующим образом:

- `C` — операция занимает постоянное (короткое) время;
- `eC` — операция занимает эффективно постоянное время, но это может зависеть от некоторых допущений, таких как максимальная длина вектора или распределение хеш-ключей;
- `aC` — операция занимает амортизированное постоянное время. Некоторые вызовы операции могут занимать больше времени, но если выполняется множество операций, то берется только постоянное среднее время, затрачиваемое на одну операцию;
- `Log` — на операцию уходит время, пропорциональное логарифму размера коллекции;
- `L` — операция имеет линейный характер, то есть на нее уходит время, пропорциональное размеру коллекции;
- `-` — операция не поддерживается.

В табл. 24.12 рассматриваются как неизменяемые, так и изменяемые типы последовательностей со следующими операциями:

- `head` — выбор первого элемента последовательности;
- `tail` — создание новой последовательности, содержащей все элементы, за исключением первого;
- `apply` — индексирование;
- `update` — функциональное обновление (с помощью метода `updated`) для неизменяемых последовательностей, обновление с побочными эффектами (с помощью метода `update`) для изменяемых;
- `prepend` — добавление элемента в начало последовательности. Если последовательность неизменяемая, то данная операция приводит к созданию новой последовательности. Если изменяемая, то существующая последовательность изменяется;
- `append` — добавление элемента в конец последовательности. Если последовательность неизменяемая, то данная операция приводит к созданию новой последовательности. Если изменяемая, то существующая последовательность изменяется;
- `insert` — вставка элемента в произвольную позицию последовательности. Поддерживается только для изменяемых последовательностей.

В табл. 24.13 рассматриваются как неизменяемые, так и изменяемые типы множеств и отображений со следующими операциями:

- `lookup` — проверка на наличие элемента во множестве или выбор значения, связанного с ключом;
- `add` — добавление нового элемента во множество или новой пары «ключ — значение» в отображение;

- `remove` — удаление элемента из множества или ключа из отображения;
- `min` — наименьший элемент множества или наименьший ключ отображения.

24.12. Равенство

В библиотеках коллекций соблюдается единообразный подход к равенству и хешированию. В первую очередь идея заключается в разбиении коллекций на категории: множества, отображения и последовательности. Коллекции из разных категорий всегда не равны. Например, коллекция `Set(1, 2, 3)` не равна коллекции `List(1, 2, 3)` даже притом, что они содержат одни и те же элементы. В то же время внутри одной и той же категории коллекции равны лишь при условии, что содержат одинаковые элементы (для последовательностей — одинаковые элементы, в одинаковом порядке), например `List(1, 2, 3) == Vector(1, 2, 3)` и `HashSet(1, 2) == TreeSet(2, 1)`.

При проверке равенства неважно, является коллекция изменяемой или неизменяемой. Для изменяемых коллекций равенство просто зависит от содержащихся в ней элементов на момент выполнения проверки на равенство. Это значит, в разное время изменяемая коллекция может быть равна разным коллекциям в зависимости от того, какие элементы были добавлены или удалены. Данное обстоятельство может стать ловушкой в случае использования изменяемых коллекций в качестве ключа в хеш-отображении. Например:

```
scala> import collection.mutable.{HashMap, ArrayBuffer}
import collection.mutable.{HashMap, ArrayBuffer}

scala> val buf = ArrayBuffer(1, 2, 3)
buf: scala.collection.mutable.ArrayBuffer[Int] =
ArrayBuffer(1, 2, 3)

scala> val map = HashMap(buf -> 3)
map: scala.collection.mutable.HashMap[scala.collection.
mutable.ArrayBuffer[Int],Int] = Map((ArrayBuffer(1, 2, 3),3))

scala> map(buf)
res13: Int = 3

scala> buf(0) += 1

scala> map(buf)
java.util.NoSuchElementException: key not found:
  ArrayBuffer(2, 2, 3)
```

В данном примере выбор, сделанный в последней строке, скорее всего, завершится сбоем, поскольку хеш-код массива `xs` в предпоследней строке изменился. Поэтому при поиске на основе хеш-кода будет отыскиваться другое место, отличное от того, в котором был сохранен `xs`.

24.13. Представления

Коллекции имеют всего несколько методов, которые конструируют новые коллекции. В качестве примеров можно привести `map`, `filter` и `++`. Такие методы мы называем *преобразователями*, поскольку они получают как минимум одну коллекцию в качестве объекта-получателя и создают в результате своей работы другую.

Преобразователи можно реализовать двумя основными способами — строгим и нестрогим (или ленивым). Строгий преобразователь создает новую коллекцию со всеми ее элементами. А нестрогий создает только заместитель получаемой в результате коллекции, а ее элементы конструируются по требованию.

В качестве примера нестрогого преобразователя рассмотрим следующую реализацию ленивой операции `map`:

```
def lazyMap[T, U](coll: Iterable[T], f: T => U) =
  new Iterable[U] {
    def iterator = coll.iterator map f
  }
```

Обратите внимание на то, что `lazyMap` создает новый объект типа `Iterable`, не прибегая к обходу всех элементов заданной коллекции `coll`. Вместо этого заданная функция `f` применяется к элементам итератора `iterator` новой коллекции по мере их востребованности.

По умолчанию коллекции в Scala — строгие во всех своих проявлениях, за исключением `LazyList`, в котором все методы преобразования реализованы лениво. Но существует систематический подход для превращения каждой коллекции в ленивую и *наоборот*, основанный на представлениях коллекций. *Представление* — это особая разновидность коллекции, которая изображает какую-либо основную коллекцию, но реализует все ее преобразователи лениво.

Для перехода от коллекции к ее представлению можно воспользоваться в отношении коллекции методом `view`. Если `xs` — некая коллекция, то `xs.view` создает точно такую же коллекцию, но с ленивой реализацией всех преобразователей. Перейти обратно от представления к строгой коллекции можно с помощью операции приведения с фабрикой строгих коллекций в качестве параметра.

В качестве примера предположим, что имеется вектор `Int`-значений, в отношении которого нужно последовательно применить две функции `map`:

```
scala> val v = Vector(1 to 10: _*)
v: scala.collection.immutable.Vector[Int] =
  Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> v map (_ + 1) map (_ * 2)
res5: scala.collection.immutable.Vector[Int] =
  Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

В последней инструкции выражение `v map (_ + 1)` создает новый вектор, который второй вызов `map (_ * 2)` преобразует в третий вектор. Во многих ситуациях

создание промежуточного результата из первого вызова `map` представляется неэкономным. В надуманном примере быстрее было бы воспользоваться одним вызовом `map` в сочетании с двумя функциями, `(_ + 1)` и `(_ * 2)`. При наличии двух функций, доступных в одном и том же месте, это можно сделать самостоятельно. Но довольно часто последовательные преобразования структур данных выполняются в различных программных модулях. Объединение этих преобразований может разрушить модульность. Более универсальный способ избавления от промежуточных результатов заключается в том, что сначала вектор превращается в представление, затем к представлению применяются все преобразования, после чего оно опять преобразуется в вектор:

```
scala> (v.view map (_ + 1) map (_ * 2)).to(Vector)
res12: Seq[Int] = Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

Мы вновь поочередно выполним эту последовательность операций:

```
scala> val vv = v.view
vv: scala.collection.IndexedSeqView[Int] =
  IndexedSeqView(<not computed>)
```

Применение `v.view` выдает значение типа `IndexedSeqView`, то есть лениво вычисляемую `IndexedSeq`-последовательность. Как и в случае с `LazyList`, применение `toString` к представлениям не приводит к вычислению их элементов. Вот почему элементы `vv` выводятся на экран как `<not computed>`.

Применение первого метода `map` к представлению дает следующий результат:

```
scala> vv map (_ + 1)
res13: scala.collection.IndexedSeqView[Int] =
  IndexedSeqView(<not computed>)
```

Результат выполнения `map` — другое значение `IndexedSeqView[Int]`. По сути, это оболочка, которая *записывает* тот факт, что `map` с функцией `(_ + 1)` нужно применить к вектору `v`. Но этот метод `map` не применяется, пока не будет принудительно создано представление. Теперь применим к последнему результату второй метод `map`:

```
scala> res13 map (_ * 2)
res14: scala.collection.IndexedSeqView[Int] =
  IndexedSeqView(<not computed>)
```

И наконец, принудительное получение последнего результата приводит к следующему диалогу:

```
scala> res14.to(Vector)
res15: Seq[Int] = Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

Обе сохраненные функции, `(_ + 1)` и `(_ * 2)`, применяются как часть выполнения операции `to`, и создается новый вектор. При таком способе промежуточные структуры данных не нужны.

Операции преобразования, применяемые к представлению, не создают новую структуру данных. Вместо этого они возвращают объект `Iterable`, итератор

которого является результатом применения операции преобразования к итератору исходной коллекции.

Потребность в использовании представлений обусловлена производительностью. Вы видели, что с помощью переключения коллекции на представление удалось избежать создания промежуточных результатов. Такая экономия может сыграть весьма важную роль. В качестве еще одного примера рассмотрим задачу поиска первого палиндрома в списке слов. Палиндромом называется слово, которое читается в обратном порядке точно так же, как и в прямом. Необходимые для этого определения имеют следующий вид:

```
def isPalindrome(x: String) = x == x.reverse
def findPalindrome(s: Iterable[String]) = s find isPalindrome
```

Теперь предположим, что имеется весьма длинная последовательность слов и нужно найти палиндром в первом ее миллионе слов. Можно ли повторно воспользоваться определением `findPalindrome`? Разумеется, можно создать следующий код:

```
findPalindrome(words take 1000000)
```

Он неплохо разделяет два аспекта, заключающихся в получении первого миллиона слов последовательности и поиска в них палиндрома. Но у этого решения есть недостаток: всегда будет создаваться промежуточная последовательность, состоящая из миллиона слов, даже если первое ее слово уже является палиндромом. Следовательно, потенциально получается, что 999 999 слов копируется в промежуточный результат, не подвергаясь последующей проверке. Многие программисты откажутся от этого и напишут собственную специализированную версию поиска палиндрома в некоем заданном префиксе последовательности аргументов. Но с представлениями этого делать не придется. Нужно просто воспользоваться следующим кодом:

```
findPalindrome(words.view take 1000000)
```

Здесь также неплохо разрешен конфликт интересов, но вместо последовательности из миллиона элементов будет создан отдельный легковесный объект представления. Таким образом, вам не придется выбирать между производительностью и модульностью.

После того как вы увидели все эти интересные примеры использования представлений, у вас может возникнуть вопрос: а зачем вообще нужны строгие коллекции? Одна из причин состоит в том, что сравнение производительности не всегда бывает в пользу ленивых коллекций. Для коллекций меньших размеров добавленные издержки на формирование и применение замыканий в представлениях зачастую выше, чем выгоды, получаемые за счет того, что в них не применяются промежуточные структуры данных. Возможно, еще более существенной причиной является то, что вычисления в представлениях могут создавать серьезные помехи, если у отложенных операций есть побочные эффекты.

Вот пример, о который обожглись многие пользователи Scala версий до 2.8. В этих версиях тип `Range` был ленивым, поэтому вел себя, по сути, как представление. Программисты пробовали создавать такие вот акторы[1]:

```
val actors = for (i <- 1 to 10) yield actor { ... }
```

А потом удивлялись, почему впоследствии ни один из акторов не выполнялся, даже если метод `actor` должен создавать и запускать актор из следующего за ним кода, заключенного в фигурные скобки. Чтобы понять, почему ничего не происходит, вспомним, что показанное ранее выражение `for` эквивалентно применению метода `map`, показанного ниже:

```
val actors = (1 to 10) map (i => actor { ... })
```

Поскольку ранее диапазон, созданный выражением `(1 to 10)`, вел себя подобно представлению, результатом выполнения `map` опять было представление. То есть элементы не вычислялись, а следовательно, акторы не создавались! Они создавались бы с помощью принудительного вычисления всего диапазона, но далеко не очевидно, что это нужно для того, чтобы заставить акторов сделать их работу.

Чтобы избегать подобных сюрпризов, коллекции Scala в версии 2.8 получили более простые правила. Все коллекции, за исключением ленивых списков и представлений, являются строгими. Перейти от строгой коллекции к ленивой можно только через метод представления `view`. Единственный способ перейти обратно — применить метод `to`. Следовательно, акторы, определенные ранее, в Scala 2.8 поведут себя ожидаемым образом, то есть будут созданы и запущены десять акторов. Чтобы вернуться к прежнему парадоксальному поведению, придется добавить явный вызов метода `view`:

```
val actors = for (i <- (1 to 10).view) yield actor { ... }
```

В целом представления — весьма эффективный инструмент, позволяющий увязать соображения эффективности с соображениями модульности. Но, чтобы не путаться в тонкостях отложенных вычислений, применение представлений лучше ограничить чисто функциональным кодом, в котором у преобразований коллекций нет побочных эффектов. И лучше избегать смешивания представлений и операций, создающих новые коллекции, если у них к тому же есть побочные эффекты.

24.14. Итераторы

Итератор — это не коллекция, а скорее способ поочередного обращения к элементам коллекции. Двумя базовыми операциями итератора являются `next` и `hasNext`. Вызов `it.next()` вернет следующий элемент итератора и продвинет итератор дальше. Затем при повторном вызове `next` в отношении того же итератора будет выдан

[1] Библиотека акторов Scala устарела, но этот исторический прием все еще актуален.

элемент, следующий за тем, который был возвращен ранее. Если возвращать станет нечего, то вызов `next` приведет к генерации исключения `NoSuchElementException`. Определить, остались ли еще элементы в коллекции, можно с помощью метода `hasNext` класса `Iterator`.

Наиболее простой способ выполнить последовательный перебор всех элементов, возвращаемых итератором, — использовать цикл `while`:

```
while (it.hasNext)
  println(it.next())
```

Итераторы в Scala также предоставляют аналоги большинства методов, имеющихся в трейтах `Iterable` и `Seq`. Например, они предоставляют метод `foreach`, который выполняет заданную процедуру в отношении каждого элемента, возвращенного итератором. При использовании `foreach` показанный ранее цикл можно сократить до следующего кода:

```
it foreach println
```

Как и всегда, в качестве альтернативного синтаксиса для выражений, использующих `foreach`, `map`, `filter` и `flatMap`, можно воспользоваться выражением `for`. То есть еще одним способом вывести на экран все элементы, возвращенные итератором, мог бы быть следующий код:

```
for (elem <- it) println(elem)
```

Между методом `foreach`, применяемым в отношении итераторов, и таким же методом, применяемым к коллекциям, допускающим обход элементов, есть существенная разница: при вызове в отношении итератора `foreach` оставит итератор в состоянии завершения его работы. Поэтому еще один вызов `next` в отношении того же самого итератора закончится неудачно и приведет к генерации исключения `NoSuchElementException`. Напротив, при вызове в отношении коллекции `foreach` оставляет количество элементов в коллекции без изменений, если только переданная функция не добавляет или не удаляет элементы, чего делать не рекомендуется, поскольку это легко может привести к неожиданным результатам.

Другие операции, общие для `Iterator` и `Iterable`, обладают таким же свойством оставлять итератор в состоянии завершения работы. Например, итераторы предоставляют метод `map`, возвращающий новый итератор:

```
scala> val it = Iterator("a", "number", "of", "words")
it: Iterator[java.lang.String] = <iterator>

scala> it.map(_.length)
res1: Iterator[Int] = <iterator>

scala> it.hasNext
res2: Boolean = true

scala> res1 foreach println
1
6
```

```
2
5

scala> it.hasNext
res4: Boolean = false
```

Как видите, после вызова `map` итератор `it` не перешел в конец, а вот итератор, полученный в результате вызова `it.map`, можно перебрать до конца.

Еще один пример — метод `dropWhile`, который может использоваться для поиска первого элемента итератора, имеющего определенное свойство. Например, для поиска в ранее показанном итераторе первого слова, в котором как минимум два символа, можно задействовать следующий код:

```
scala> val it = Iterator("a", "number", "of", "words")
it: Iterator[java.lang.String] = <iterator>

scala> it dropWhile (_.length < 2)
res4: Iterator[java.lang.String] = <iterator>

scala> it.next()
res5: java.lang.String = number
```

Еще раз обратите внимание на то, что `it` был изменен вызовом `dropWhile`: теперь `it` указывает на второе слово в списке — `"number"`. Фактически `it` и результат `res`, полученный при выполнении `dropWhile`, вернут одинаковую последовательность элементов.

Есть только одна стандартная операция, `duplicate`, позволяющая повторно использовать один и тот же итератор:

```
val (it1, it2) = it.duplicate
```

Вызов метода `duplicate` возвращает два итератора, каждый из которых возвращает точно такие же элементы, что и итератор `it`. Оба итератора работают независимо друг от друга, продвижение одного из них совершенно не влияет на состояние другого. В отличие от этого, исходный итератор `it` продвигается при вызове `duplicate` в самый конец и становится непригодным для дальнейшего использования.

В целом итераторы ведут себя как коллекции, *если вы никогда не обращаетесь к итератору еще раз после вызова метода на нем*. Библиотеки коллекций Scala делают это явным с помощью абстракции под названием `IterableOnce`, которая является обобщающим супертрейтом для `Iterable` и `Iterator`. Судя по названию, объекты типа `IterableOnce` допускают обход своих элементов хотя бы однократно, но состояние таких объектов после обхода не определено. Если объект типа `IterableOnce` фактически относится к типу `Iterator`, то после обхода будет указывать на конец коллекции, если же относится к типу `Iterable`, то не изменит своего состояния. Чаще всего `IterableOnce` используется как тип аргумента для методов, который могут получать в качестве аргумента `Iterator` или `Iterable`. Примером может послужить метод добавления `++` в трейте `Iterable`. Он получает параметр типа `IterableOnce`, позволяя добавлять элементы, получаемые как из коллекции типа `Iterator`, так и из `Iterable`.

Все операции над итераторами сведены в табл. 24.14.

Таблица 24.14. Операции в трейте Iterator

Что собой представляет	Результат работы
Абстрактные методы	
it.next()	Возвращает следующий элемент в итераторе и продвигается за него
it.hasNext	Возвращает true, если может вернуть еще элемент
Вариации	
it.buffered	Буферизованный итератор, возвращающий все элементы it
it grouped size	Итератор, выдающий элементы, возвращаемые it, блоками фиксированного размера
it sliding size	Итератор, выдающий элементы, возвращаемые it, в виде скользящего по коллекции окна фиксированного размера
Копирование	
it.copyToArray(arr, s, l)	Копирует не более одного элемента, возвращенного it, в массив arr, начиная с индекса s. Последние два аргумента являются необязательными
Дублирование	
it.duplicate	Пара итераторов, каждый из которых независимо от другого возвращает все элементы it
Добавление	
it ++ jt	Итератор, выдающий все элементы, возвращаемые итератором it, а затем все элементы, возвращаемые итератором jt
it.padTo(len,x)	Итератор, выдающий все элементы it, а затем копии x до тех пор, пока не будет полностью достигнута длина len
Отображение	
it map f	Итератор, получаемый в результате применения функции к каждому элементу, возвращаемому it
it flatMap f	Итератор, получаемый в результате применения возвращающей итератор функции f к каждому элементу it и добавления результатов
it collect f	Итератор, получаемый в результате использования частично примененной функции f к каждому элементу it, для которого она определена, и сбора результатов
Преобразование	
it.toArray	Собирает возвращаемые it элементы в массив
it.toList	Собирает возвращаемые it элементы в список
it.toIterable	Собирает возвращаемые it элементы в коллекцию Iterable

Что собой представляет	Результат работы
it.toSeq	Собирает возвращаемые it элементы в последовательность
it.toIndexedSeq	Собирает возвращаемые it элементы в индексированную последовательность
it.toSet	Собирает возвращаемые it элементы во множество
it.toMap	Собирает возвращаемые it пары «ключ — значение» в отображение
it to SortedSet	Обобщенная операция преобразования, принимающая в качестве параметра фабрику коллекций
Информация о размере	
it.isEmpty	Проверяет, пуст ли итератор (противоположность hasNext)
it.nonEmpty	Проверяет, содержит ли итератор элементы (псевдоним метода hasNext)
it.size	Количество элементов, возвращенных it (после этой операции it окажется в самом конце!)
it.length	То же самое, что и it.size
it.knownSize	Количество элементов, если его можно узнать без изменения состояния итератора; в противном случае –1
Индексированный поиск с извлечением элемента	
it find p	Option-значение, содержащее первый возвращаемый it элемент, удовлетворяющий условию p, или None при отсутствии таких элементов (итератор продвигается за найденный элемент или, если не найден ни один элемент, оказывается в конечной позиции)
it indexOf x	Индекс первого возвращенного it элемента, равного x (итератор продвигается за найденный элемент)
it indexWhere p	Индекс первого возвращенного it элемента, удовлетворяющего условию p (итератор продвигается за найденный элемент)
Подытераторы	
it take n	Итератор, выдающий первые n элементов итератора it (it продвинется за позицию n-го элемента или же оказывается в конечной позиции, если в it содержится меньше n элементов)
it drop n	Итератор, начинающий выборку с элемента it с позиции (n + 1) (it продвинется до этой же позиции)
it.slice(m, n)	Итератор, выдающий блок элементов, возвращенный из it, начинающийся с m-го элемента и заканчивающийся перед n-м элементом
it takeWhile p	Итератор, выдающий элементы из it, пока условие p вычисляется в true

Продолжение

Таблица 24.14 *(продолжение)*

Что собой представляет	Результат работы
it dropWhile p	Итератор, пропускающий элементы из it, пока условие p вычисляется в true, и выдающий остаток
it filter p	Итератор, выдающий все элементы из it, удовлетворяющие условию p
it withFilter p	То же самое, что и it filter p. Эта операция необходима для использования итераторов в выражениях for
it filterNot p	Итератор, выдающий все элементы из it, не удовлетворяющие условию p
it.distinct	Итератор, выдающий все элементы из it, не включая дубликаты
Деление	
it partition p	Разбивает it на два итератора, один из которых возвращает из it все элементы, удовлетворяющие условию p, а другой — все элементы, не удовлетворяющие этому условию
Состояние элементов	
it forall p	Булево значение, показывающее, соблюдается ли условие p для всех элементов, возвращаемых it
it exists p	Булево значение, показывающее, соблюдается ли условие p для какого-либо из элементов, возвращаемых it
it count p	Количество элементов в it, удовлетворяющих условию p
Свертки	
it.foldLeft(z)(op)	Применяет бинарную операцию op к соседним элементам, возвращаемым it, проходя коллекцию слева направо и начиная с z
it.foldRight(z)(op)	Применяет бинарную операцию op к соседним элементам, возвращаемым it, проходя коллекцию справа налево и начиная с z
it reduceLeft op	Применяет бинарную операцию op к соседним элементам, возвращаемым непустым итератором it, проходя коллекцию слева направо
it reduceRight op	Применяет бинарную операцию op к соседним элементам, возвращаемым непустым итератором it, проходя коллекцию справа налево
Специализированные свертки	
it.sum	Сумма значений числовых элементов, выдаваемых итератором it
it.product	Произведение значений числовых элементов, выдаваемых итератором it
it.min	Минимальное значение упорядочиваемых элементов, выдаваемых итератором it

Что собой представляет	Результат работы
it.max	Максимальное значение упорядочиваемых элементов, выдаваемых итератором it
Слияние	
it zip jt	Итератор пар соответствующих элементов, выдаваемых итераторами it и jt
it.zipAll(jt, x, y)	Итератор пар соответствующих элементов, выдаваемых итераторами it и jt, причем тот итератор, что короче, расширяется, чтобы соответствовать более длинному итератору путем добавления элементов x или y
it.zipWithIndex	Итератор пар, состоящих из элементов, возвращаемых it, и их индексов
Обновление	
it.patch(i, jt, r)	Итератор, получаемый из it путем замены r элементов, начиная с позиции i, итератором вставки jt
Сравнение	
it sameElements jt	Проверяет, не возвращают ли итераторы it и jt одни и те же элементы в одном и том же порядке. Обратите внимание: после этой операции следует отбросить как it, так и jt
Строки	
it.addString(b, start, sep, end)	Добавляет строку к StringBuilder b, которая показывает все элементы, возвращаемые it, между разделителями sep и заключена в строковые значения start и end. Аргументы start, sep и end являются необязательными
it.mkString(start, sep, end)	Преобразует итератор в строку, показывающую все элементы, возвращаемые it, между разделителями sep, и заключенную в строковые значения start и end. Аргументы start, sep и end являются необязательными

Буферизованные итераторы

Порой бывает необходим итератор, способный заглянуть вперед, позволяя проинспектировать следующий возвращаемый элемент, не продвигаясь за него. Рассмотрим, к примеру, задачу пропуска начальных пустых строк итератора, который возвращает последовательность строк. Возможно, возникнет соблазн создать нечто похожее на метод, показанный ниже:

```
// Этот код работать не будет
def skipEmptyWordsNOT(it: Iterator[String]) = {
  while (it.next().isEmpty) {}
}
```

Но, присмотревшись, можно понять, что этот код неработоспособен: разумеется, он будет пропускать начальные пустые строки, но также продвинет `it` за первую непустую строку!

Решить эту задачу можно с помощью буферизованного итератора, экземпляра трейта `BufferedIterator`, который является подтрейтом трейта `Iterator` и предоставляет еще один метод по имени `head`. Вызов `head` в отношении буферизованного итератора приведет к возвращению его первого элемента, но без продвижения итератора. При использовании буферизованного итератора код для пропуска пустых слов может выглядеть следующим образом:

```
def skipEmptyWords(it: BufferedIterator[String]) =
  while (it.head.isEmpty) { it.next() }
```

Каждый итератор может быть преобразован в буферизованный итератор путем вызова своего метода `buffered`. Пример его использования выглядит так:

```
scala> val it = Iterator(1, 2, 3, 4)
it: Iterator[Int] = <iterator>

scala> val bit = it.buffered
bit: java.lang.Object with scala.collection.
  BufferedIterator[Int] = <iterator>

scala> bit.head
res10: Int = 1

scala> bit.next()
res11: Int = 1

scala> bit.next()
res11: Int = 2
```

Следует заметить, что вызов `head` в отношении буферизованного итератора `bit` не приводит к изменению позиции итератора. Поэтому последующий вызов, `bit.next()`, возвращает то же самое значение, что и `bit.head`.

24.15. Создание коллекций с нуля

Вам уже попадался синтаксис наподобие `List(1, 2, 3)`, который создает список из трех целых чисел, и `Map('A' -> 1, 'C' -> 2)`, создающий отображение с двумя привязками. Фактически это универсальная возможность коллекций Scala. Можно взять любое имя коллекции и указать после него в круглых скобках список элементов. В результате получится новая коллекция с заданными элементами. Ниже представлены еще примеры:

```
Iterable()                // пустая коллекция
List()                    // пустой список
List(1.0, 2.0)            // список с элементами 1.0, 2.0
Vector(1.0, 2.0)          // вектор с элементами 1.0, 2.0
Iterator(1, 2, 3)         // итератор, возвращающий три целых числа
Set(dog, cat, bird)       // множество из трех животных
HashSet(dog, cat, bird)   // хеш-множество из тех же животных
Map('a' -> 7, 'b' -> 0)   // отображение символов на целые числа
```

«Скрытно» каждая из показанных ранее строк является вызовом метода `apply` определенного объекта. Например, третья из этих строк раскрывается в следующий код:

```
List.apply(1.0, 2.0)
```

Здесь показан вызов метода `apply`, принадлежащего объекту-компаньону класса `List`. Этот метод получает произвольное число аргументов и создает из них список. Каждый класс коллекций в библиотеке Scala располагает объектом-компаньоном с таким же методом `apply`. И неважно, представлена конкретная реализация классом коллекции, как в случае с `List`, `LazyList` или `Vector`, или же трейтом, как в случае с `Seq`, `Set` или `Iterable`. В последнем случае вызов `apply` приведет к созданию некой исходной реализации трейта. Рассмотрим ряд примеров:

```
scala> List(1, 2, 3)
res17: List[Int] = List(1, 2, 3)

scala> Iterable(1, 2, 3)
res18: Iterable[Int] = List(1, 2, 3)

scala> mutable.Iterable(1, 2, 3)
res19: scala.collection.mutable.Iterable[Int] = ArrayBuffer(1, 2, 3)
```

Помимо `apply`, в каждом объекте-компаньоне определен и элемент `empty`, возвращающий пустую коллекцию. Поэтому вместо `List()` можно воспользоваться кодом `List.empty`, вместо `Map()` задействовать код `Map.empty` и т. д.

Кроме того, потомки трейтов `Seq` и `Set` в своих объектах-компаньонах предоставляют другие факторные операции, которые сведены в табл. 24.15. Вкратце это:

- `concat`, которая конкатенирует произвольное количество коллекций;
- `fill` и `tabulate`, которые создают одно- или многоразмерные коллекции заданной размерности, инициализированные каким-либо выражением или функцией составления таблицы;
- `range`, которая создает коллекции целых чисел с какой-либо постоянной длиной шага;
- `iterate` и `unfold`, которые создают последовательность, получающуюся из многократного применения функции к начальному элементу или состоянию.

Таблица 24.15. Фабричные методы для трейтов Seq и Set

Что собой представляет	Результат работы
C.empty	Пустая коллекция
C(x, y, z)	Коллекция, состоящая из элементов x, y и z
C.concat(xs, ys, zs)	Коллекция, получаемая конкатенацией элементов коллекций xs, ys и zs

Продолжение

Таблица 24.15 *(продолжение)*

Что собой представляет	Результат работы
C.fill(n)(e)	Коллекция длиной n, где каждый элемент вычисляется выражением e
C.fill(m, n)(e)	Коллекция коллекций размерностью m × n, где каждый элемент вычисляется выражением e (существует также в более высоких измерениях)
C.tabulate(n)(f)	Коллекция длиной n, где элемент по каждому индексу i вычисляется путем вызова f(i)
C.tabulate(m, n)(f)	Коллекция коллекций размерностью m × n, где элемент по каждому индексу (i, j) вычисляется путем вызова f(i, j) (существует также в более высоких измерениях)
C.range(start, end)	Коллекция целых чисел start ... end – 1
C.range(start, end, step)	Коллекция целых чисел, начинающаяся со start и наращиваемая с шагом step до значения end, исключая само это значение
C.iterate(x, n)(f)	Коллекция длиной n с элементами x, f(x), f(f(x)), ...
C.unfold(init)(f)	Коллекция, которая использует функцию f для вычисления своего следующего элемента и состояния, начиная с состояния init

24.16. Преобразования между коллекциями Java и Scala

Как и в Scala, в Java есть богатая библиотека коллекций. Обе библиотеки имеют много общего. Например, и в той и в другой есть такие категории, как итераторы, итерируемые коллекции, множества, отображения и последовательности. Но есть и важные различия. В частности, в библиотеках Scala уделяется намного больше внимания неизменяемым коллекциям и предоставляется куда больше операций, выполняющих преобразование коллекции в новую коллекцию.

Иногда может понадобиться выполнить преобразование из одной среды в другую. Например, нужно обратиться к уже существующей Java-коллекции, как если бы это была Scala-коллекция. Или же следует передать одну из коллекций Scala методу Java, который ожидает получения Java-аналога. Сделать это не составит никакого труда, поскольку Scala предлагает в объекте `JavaConverters` неявные преобразования между всеми основными типами коллекций. В частности, двунаправленные преобразования имеются между следующими типами:

```
Iterator          ⇔     java.util.Iterator
Iterator          ⇔     java.util.Enumeration
Iterable          ⇔     java.lang.Iterable
Iterable          ⇔     java.util.Collection
```

```
mutable.Buffer   ⇔     java.util.List
mutable.Set      ⇔     java.util.Set
mutable.Map      ⇔     java.util.Map
```

Чтобы включить эти преобразования, нужно просто импортировать их:

```
scala> import collection.JavaConverters._
import collection.JavaConverters._
```

Это делает возможным неявное приведение между соответствующими коллекциями Scala и Java с помощью методов расширения `asScala` и `asJava`:

```
scala> import collection.mutable._
import collection.mutable._

scala> val jul: java.util.List[Int] = ArrayBuffer(1, 2, 3).asJava
jul: java.util.List[Int] = [1, 2, 3]

scala> val buf: Seq[Int] = jul.asScala
buf: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 2, 3)

scala> val m: java.util.Map[String, Int] =
          HashMap("abc" -> 1, "hello" -> 2).asJava
m: java.util.Map[String,Int] = {hello=2, abc=1}
```

Внутренний механизм этих преобразований работает за счет создания объекта-обертки, пересылающего все операции базовому объекту коллекции. Поэтому коллекции при преобразовании между Java и Scala никогда не копируются. Интересной особенностью является то, что при круговом преобразовании из, скажем, Java-типа в соответствующий Scala-тип и обратно, в тот же Java-тип, будет получен точно такой же объект коллекции, который имелся в самом начале.

Есть также ряд других востребованных Scala-коллекций, которые могут быть преобразованы в Java-типы, но для которых нет соответствующих преобразований в обратном направлении. К ним относятся:

```
Seq           ⇔     java.util.List
mutable.Seq   ⇔     java.util.List
Set           ⇔     java.util.Set
Map           ⇔     java.util.Map
```

Поскольку в Java изменяемые и неизменяемые коллекции по типам не различаются, преобразование из, скажем, `collection.immutable.List` выдаст коллекцию `java.util.List`. При всех попытках применить операции по внесению изменений в отношении этой коллекции будет генерироваться исключение `UnsupportedOperationException`, например:

```
scala> val jul: java.util.List[Int] = List(1, 2, 3)
jul: java.util.List[Int] = [1, 2, 3]

scala> jul.add(7)
java.lang.UnsupportedOperationException
        at java.util.AbstractList.add(AbstractList.java:131)
```

Резюме

Теперь вы получили более детальное представление об использовании коллекций Scala. В них применен подход, предоставляющий вам целый ряд не просто полезных специализированных методов, но по-настоящему эффективных строительных блоков. Сочетание двух или трех таких строительных блоков позволит провести множество полезных вычислений. Эффективность стиля, принятого в библиотеке, наиболее ярко проявляется благодаря имеющемуся в Scala облегченному синтаксису для функциональных литералов, а также благодаря тому, что сам язык предоставляет множество типов коллекций, сохраняющих постоянство и неизменяемость.

В данной главе мы показали коллекции с точки зрения программиста, использующего библиотеку коллекций. В следующей главе покажем, как создаются коллекции и как вы можете добавить собственные типы коллекций.

25 Архитектура коллекций Scala

В этой главе мы подробно опишем архитектуру фреймворка коллекций Scala. Продолжая изучать тему, начатую в главе 24, вы узнаете подробности внутренней работы фреймворка. Кроме того, мы покажем, как эта архитектура помогает определить ваши собственные коллекции всего в нескольких строках кода с учетом того, что подавляющая часть функциональных возможностей коллекции будет взята из этого фреймворка.

В главе 24 перечислялось большое количество операций над коллекциями, существующих в единой форме во многих реализациях различных коллекций. Реализация заново каждой операции над коллекцией для каждого типа коллекции привела бы к образованию огромного объема кода, основная часть которого была бы скопирована из другого места. Такое дублирование кода со временем может привести к несовместимости, вызванной тем, что операция добавляется или изменяется в одной части библиотеки коллекций, а остальные части остаются без изменений. Главная цель структуры фреймворка коллекций заключается в том, чтобы избежать дублирования, определяя каждую операцию в наименьшем количестве мест[1]. Подход к проектированию заключался в реализации большинства операций в «шаблонных трейтах», которые можно примешивать к отдельным базовым классам и классам реализации коллекций. В этой главе мы рассмотрим эти шаблоны наряду с другими классами и трейтами, составляющими строительные блоки фреймворка, а также поддерживаемые ими принципы конструирования.

25.1. Исключение общих операций

С точки зрения проектирования основная задача библиотеки коллекций состоит в предоставлении пользователям естественных типов, реализации которых имеют как можно больше общего кода. В частности, фреймворк коллекций

[1] В идеале все должно определяться в одном месте, но есть ряд исключений, где требуется переопределение.

Scala должен поддерживать следующие аспекты различных конкретных типов коллекций:

- некоторые операции преобразования возвращают тот же конкретный тип коллекции. Например, вызов `filter` для `List[Int]` возвращает `List[Int]`;
- некоторые операции преобразования возвращают тот же конкретный тип коллекций, но, возможно, с другим типом элементов. Например, вызов `map` для `List[Int]` может вернуть `List[String]`;
- некоторые разновидности коллекций, такие как `List[A]`, имеют один параметр типа, а другие — два, как в случае с `Map[K, V]`;
- некоторые операции над коллекциями возвращают результат, который зависит от типа элементов. Например, вызов `map` для `Map` возвращает другой экземпляр `Map`, если результатом функции отображения является пара «ключ — значение»; в противном случае возвращается `Iterable`;
- операции преобразования над определенными видами коллекций требуют дополнительных неявных параметров. Например, вызов `map` для `SortedSet` требует указать неявный параметр `Ordering`;
- наконец, некоторые коллекции, такие как `List`, являются строгими, а другие, включая `View` и `LazyList`, — нестрогими.

Шаблонные трейты фреймворка коллекций абстрагируют перечисленные выше отличия между разными коллекциями. В оставшейся части данного раздела вы узнаете, как это достигается.

Ранее мы уже упоминали, что одна из непростых задач, стоящих перед шаблонными трейтами, состоит в определении операций преобразования, которые возвращают конкретные типы коллекций, неизвестные заранее. Взгляните, к примеру, на сигнатуру операции `map` над `List[A]` и `Vector[A]`:

```
trait List[A] {
  def map[B](f: A => B): List[B]
}

trait Vector[A] {
  def map[B](f: A => B): Vector[B]
}
```

Тип, который возвращает `map`, зависит от конкретного типа коллекции. Метод `map` для `List` возвращает `List`, а для `Vector` он возвращает `Vector`. Эту особенность фреймворка коллекций называют принципом «того же результирующего типа»: везде, где это возможно, операция преобразования над коллекцией возвращает коллекцию того же типа. В случае с `map` шаблонные трейты должны абстрагировать конструктор результирующего типа коллекции[1].

Метод `filter` имеет немного другое требование. Взгляните на сигнатуру типа для `List[A]` и `Map[K, V]`:

[1] Конструкторы типов были описаны в разделе 19.3.

```
trait List[A] {
  def filter(p: A => Boolean): List[A]
}

trait Map[K, V] {
  def filter(p: ((K, V)) => Boolean): Map[K, V]
}
```

Чтобы обобщить сигнатуру типа метода `filter`, шаблонные трейты не могут просто абстрагировать конструктор типа коллекции, так как `List` принимает один параметр, а `Map` — два. Поэтому в данном случае шаблонные трейты должны абстрагировать весь результирующий тип коллекции.

Абстрагирование типов коллекций

Шаблонный трейт `IterableOps` реализует операции над типом коллекции `Iterable[A]`. Он имеет следующее определение:

```
trait IterableOps[+A, +CC[_], +C]
```

Трейт `IterableOps` объявляет три параметра типа с именами `A`, `CC` и `C`. Для типа `Iterable`, который расширяет `IterableOps`, `A` определяет тип элементов, `CC` — тип конструктора, а `C` — его полный тип. Сигнатуры методов `filter` и `map` для `IterableOps` выглядят так:

```
trait IterableOps[+A, +CC[_], +C] {
  def filter(p: A => Boolean): C
  def map[B](f: A => B): CC[B]
}
```

При примешивании в `IterableOps` конкретные типы в фреймворке коллекций передают подходящие параметры типов. Например, `List[A]` передает элемент типа `A` для `A`, `List` для `CC` и `List[A]` для `C`:

```
trait List[+A]
  extends Iterable[A]
    with IterableOps[A, List, List[A]]
```

Шаблонный трейт `SetOps` расширяет `IterableOps` и добавляет реализацию по умолчанию для методов, определенных в `Set`, которые не определены в `Iterable`. Вот определение `SetOps`:

```
trait SetOps[A, +CC[_], +C]
    extends IterableOps[A, CC, C]
```

Трейт `Set` примешивается в `SetOps`, передавая подходящие параметры типов:

```
trait Set[A] extends Iterable[A] with SetOps[A, Set, Set[A]]
```

С отображениями все немного сложнее. Трейт `IterableOps` работает для списков, однако сигнатура типа его операции `map` не работает с `Map`. Проблема в том,

что конструктор типа `Map[K, V]` принимает два параметра типов, `K` и `V`, а параметр типа `IterableOps CC[_]` принимает только один. В связи с этим фреймворк коллекций предоставляет шаблонный трейт `MapOps`, который содержит вторую, перегруженную версию метода `map`:

```
trait MapOps[K, +V, +CC[_, _], +C]
  extends IterableOps[(K, V), Iterable, C] {

  def map[K2, V2](f: ((K, V)) => (K2, V2)): CC[K2, V2]
}
```

Поскольку трейт `MapOps` расширяет `IterableOps`, операции, определенные в `IterableOps`, доступны и в `MapOps`. Следовательно, если учесть, что трейту `IterableOps` в качестве конструктора типа коллекции перадается `Iterable`, трейт `Map[K, V]` наследует две перегруженные версии операции `map`:

```
// наследуется от MapOps
def map[K2, V2](f: ((K, V)) => (K2, V2)): Map[K2, V2]

// наследуется от IterableOps
def map[B](f: ((K, V)) => B): Iterable[B]
```

Когда вы вызываете `map` в своем коде, компилятор выбирает одну из перегруженных версий. Если функция, переданная в `map` в качестве аргумента, возвращает пару значений, то подойдут обе версии. В этом случае приоритет отдается версии из `MapOps`, поскольку согласно правилам разрешения перегрузки она является более конкретной. Таким образом итоговой коллекцией будет `Map`. Если функция-аргумент не возвращает пару значений, то применимой оказывается только версия, которая определена в `IterableOps`. В этом случае итоговая коллекция будет иметь тип `Iterable`. Вот как принцип «того же результирующего типа» соблюдается на уровне проектирования.

Иногда бывает, что сигнатуры типов операций преобразования, которые определены в `IterableOps`, не соответствуют конкретным типам коллекций. Это, к примеру, относится к `SortedSet[A]`. В таких случаях сигнатура типа операции `map` требует, чтобы вы указали неявный параметр `Ordering` для типа элементов:

```
def map[B](f: A => B)
      (implicit ord: Ordering[B]): SortedSet[B]
```

Для `SortedSet`, как и для `Map`, нужен специализированный шаблонный трейт с перегруженными версиями операций преобразования. Для этого предусмотрен шаблонный трейт `SortedSetOps`. Вот как в нем определен метод `map`:

```
trait SortedSetOps[A, +CC[_], +C]
    extends SetOps[A, Set, C] {

  def map[B](f: A => B)(implicit ord: Ordering[B]): CC[B]
}
```

Типы коллекций, которые расширяют `SortedSetOps`, передают подходящие параметры типов. Например, `SortedSet[A]` передает свой элемент типа `A` для `A`, `SortedSet` для `CC` и `SortedSet[A]` для `C`:

```
trait SortedSet[A] extends
    SortedSetOps[A, SortedSet, SortedSet[A]]
```

Поскольку `SortedSetOps` расширяет `SetOps`, а в качестве конструктора типа коллекции в `SetOps` передается `Set`, то трейт `SortedSet[A]` наследует эти две перегруженные версии операции `map`:

```
// наследуется от SortedSetOps
def map[B](f: A => B)(implicit ord: Ordering[B]): SortedSet[B]

// наследуется от IterableOps, по пути SetOps
def map[B](f: A => B): Set[B]
```

Если при вызове `map` доступен неявный тип `Ordering[B]`, то по правилам разрешения перегрузки выберется версия из `SortedSetOps` и результатом будет `SortedSet[B]`. В противном случае используется версия, определенная в `IterableOps`, и результатом будет `Set[B]`.

Наконец, пятая по счету разновидность коллекций, которой нужен специальный шаблонный трейт, — `SortedMap[K, V]`. Она принимает два параметра типов и требует, чтобы для типа ключей был задан неявный порядок следования элементов. Соответствующие перегруженные методы предоставляет шаблонный трейт `SortedMapOps`.

На рис. 25.1 изображено отношение наследования между главными шаблонными трейтами и типами коллекций. Трейты, названия которых заканчиваются на `Ops`, являются шаблонными.

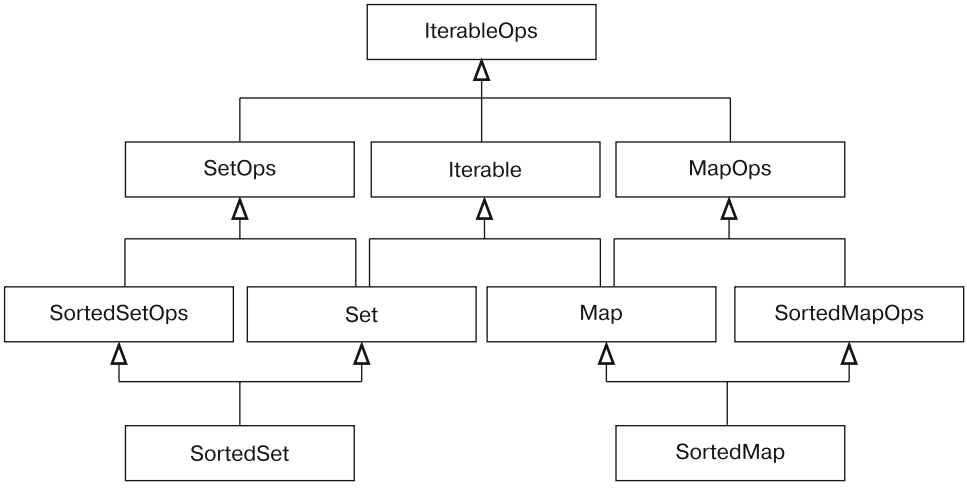


Рис. 25.1. Главные трейты фреймворка коллекций

Управление строгостью

В состав библиотеки коллекций `Scala` входят как *строгие*, так и *нестрогие* типы коллекций. Строгие, такие как `List` и `Set`, вычисляют значения своих элементов сразу. Нестрогие, такие как `LazyList` и `View`, откладывают вычисление до тех пор, пока не возникнет необходимость в том или ином элементе.

Следовательно, шаблонные трейты фреймворка коллекций должны поддерживать как строгое, так и нестрогое вычисление значений и позволять конкретным типам коллекций, которые расширяют шаблонные трейты, самостоятельно определять строгость. Например, реализация `map` по умолчанию должна быть строгой при наследовании списком и нестрогой при наследовании представлением.

Чтобы этого достичь, операции в шаблонных трейтах по умолчанию реализованы для нестрогих представлений. Представление (`View`) хранит «описание» операции, применяемой к коллекции, однако не вычисляет ее результат, пока все содержимое `View` не будет перебрано[1]. Как показано в листинге 25.1, трейт `View` определяется как `Iterable` с одним абстрактным методом, который предоставляет `Iterator` для обхода элементов `View`. Значения этих элементов будут вычислены только после прохождения `Iterator`.

Листинг 25.1. Упрощенное определение View

```
trait View[+A]
  extends Iterable[A]
    with IterableOps[A, View, View[A]] {
  def iterator: Iterator[A]
}
```

В листинге 25.2 показана реализация методов `filter` и `map` в `IterableOps`. Вначале метод `filter` создает экземпляр `View.Filter`; это представление, которое будет фильтровать элементы исходной коллекции, когда и если их будут перебирать. Этот экземпляр `View.Filter` передается в метод `fromSpecific`, который превратит представление в конкретную коллекцию типа `C`. Реализация `map` устроена похожим образом, только вместо `fromSpecific` она вызывает метод `from`, который использует в качестве своего параметра `source` экземпляр `Iterable` или итератор с произвольным типом элементов `E`.

Листинг 25.2. Реализация filter и map в IterableOps

```
trait IterableOps[+A, +CC[_], +C] {

  def filter(pred: A => Boolean): C =
    fromSpecific(new View.Filter(this, pred))

  def map[B](f: A => B): CC[B] =
    from(new View.Map(this, f))
```

[1] Представления были описаны в разделе 24.13.

```
  protected def fromSpecific(source: IterableOnce[A]): C
  protected def from[E](source: IterableOnce[E]): CC[E]
}
```

Реализация методов `from` и `fromSpecific` ложится на конкретные коллекции: они сами решают, как вычислять значения исходных элементов: строго или нестрого. Например, трейт `List` реализует метод `from`, немедленно вычисляя значения элементов коллекции `source`:

```
def from[E](source: IterableOnce[E]): List[E] =
  (new ListBuffer[E] ++= source).toList
```

Для сравнения, реализация `from` в `LazyList` (здесь она не показана) не вычисляет значения исходных элементов, пока их не начнут перебирать.

Еще один аспект реализации, о котором вам следует знать, состоит в том, что метод `from` на самом деле определен в объектах-компаньонах коллекций в качестве общего фабричного метода для создания экземпляров коллекции. Шаблонный трейт `IterableOps` просто переходит к нему через метод `iterableFactory`, который связывает классы коллекций с их объектами-компаньонами:

```
trait IterableOps[+A, +CC[_], +C] {

  def map[B](f: A => B): CC[B] =
    iterableFactory.from(new View.Map(this, f))

  def iterableFactory: IterableFactory[CC]
}

trait IterableFactory[+CC[_]] {

  def from[A](source: IterableOnce[A]): CC[A]
}
```

Наконец, фабричные трейты, как и шаблонные (такие как `IterableOps`, `MapOps` и `SortedSetOps`), могут иметь несколько вариантов. Например, в листинге 25.3 показаны важные в этом смысле участки кода реализации `map` из шаблонного трейта `MapOps`.

Листинг 25.3. Шаблонный трейт MapOps и связанный с ним трейт MapFactory

```
trait MapOps[K, +V, +CC[_, _], +C]
  extends IterableOps[(K, V), Iterable, C] {

  def map[K2, V2](f: ((K, V)) => (K2, V2)): CC[K2, V2] =
    mapFactory.from(new View.Map(this, f))

  def mapFactory: MapFactory[CC]
}

trait MapFactory[+CC[_, _]] {
  def from[K, V](source: IterableOnce[(K, V)]): CC[K, V]
}
```

Когда строгое вычисление предпочтительно или неизбежно

В предыдущих разделах мы объяснили, что строгость конкретных коллекций должна сохраняться за счет операций, реализованных по умолчанию. Однако в некоторых случаях это снижает эффективность реализаций.

Например, ленивая версия метода `partition` должна выполнить два обхода исходной коллекции. В других случаях, таких как `groupBy`, операция должна выполняться строго: ее попросту невозможно реализовать, не вычисляя значений элементов.

В таких ситуациях библиотека коллекций дает возможность реализовать операции в строгом стиле. Но при этом они будут основаны не на `View`, а на `Builder`. Вот как выглядит каркас трейта `Builder`:

```
package scala.collection.mutable

trait Builder[-A, +C] {

  def addOne(elem: A): this.type

  def result(): C

  def clear(): Unit
}
```

Трейт `Builder` — обобщенный как в контексте типов элементов, `A`, так и с точки зрения типа коллекции, `C`, которую он возвращает. Вы можете добавить элемент `x` в `Builder b` с помощью операции `b.addOne(x)` (или `b += x`). Метод `result` возвращает коллекцию из `Builder`. Состояние `Builder` после получения его результатов становится неопределенным, но его можно сбросить и заменить новым пустым состоянием с помощью метода `clear`.

По аналогии с `fromSpecific` и `from` шаблонные трейты предоставляют один метод с именем `newSpecificBuilder` в целях получения `Builder`, который возвращает коллекцию с элементами того же типа, и еще один, `newBuilder`, для получения `Builder` с тем же типом коллекции, но с элементами другого типа. В листинге 25.4 показаны участки кода `IterableOps` и `IterableFactory` для построения коллекций как в строгом, так и в нестрогом режиме.

Листинг 25.4. Строгие и нестрогие методы для построения экземпляров коллекций

```
trait IterableOps[+A, +CC[_], +C] {

  def iterableFactory: IterableFactory[CC]

  protected def fromSpecific(source: IterableOnce[A]): C

  protected def newSpecificBuilder: Builder[A, C]
}

trait IterableFactory[+CC[_]] {
```

```
  def from[A](source: IterableOnce[A]): CC[A]

  def newBuilder[A]: Builder[A, CC[A]]
}
```

Отметим, что в целом операции, которым *не нужно* быть строгими, следует реализовывать нестрого, иначе при использовании с нестрогими конкретными коллекциями они будут вести себя непредсказуемо. Тем не менее строгий режим зачастую оказывается более эффективным. Вот почему библиотека Scala предоставляет еще один набор шаблонных трейтов, операции в которых переопределены с помощью преимуществ, которые предоставляет строгий `Builder`. Названия этих шаблонных трейтов всегда начинаются с `StrictOptimized`. Вы должны использовать их в собственных коллекциях, если они являются строгими.

25.2. Интеграция новых коллекций

Что нужно сделать для интеграции нового класса коллекции, чтобы он воспользовался всеми предопределенными операциями над нужными типами? В этом разделе мы разберем три примера, посвященные таким темам: ограниченные последовательности, последовательности оснований РНК и префиксные отображения, реализованные с помощью базисных деревьев.

Ограниченные последовательности

Представьте, что хотите создать неизменяемую коллекцию, содержащую *максимум* `n` элементов: если количество добавленных элементов превышает максимальное, то удаляется первый элемент. Для начала нужно выбрать супертип коллекции: `Seq`, `Set`, `Map` или просто `Iterable`? В данном случае может возникнуть соблазн выбрать `Seq`, так как ваша коллекция может содержать дубликаты, а порядок итераций определяется порядком вставки. Однако отдельные свойства `Seq` не удовлетворяются. Например, размер последовательности `Seq`, которая является результатом конкатенации двух других последовательностей, должен быть суммой двух исходных размеров:

```
(xs ++ ys).size == xs.size + ys.size
```

В результате единственным разумным выбором в качестве базового типа для вашей новой коллекции будет `scala.collection.immutable.Iterable`.

Ограниченная коллекция, первая версия

В листинге 25.5 представлена первая версия ограниченной коллекции. Позже мы ее улучшим. У класса `Capped1` есть приватный конструктор, который принимает в качестве параметров емкость и длину коллекции, отступ (индекс первого элемента) и массив, в котором будут храниться элементы. Публичный конструктор

принимает только емкость коллекции, делая отступ нулевым и создавая неинициализированный массив для хранения элементов, длина которого равна заданной емкости.

Листинг 25.5. Коллекция Capped, первая версия

```
import scala.collection._

class Capped1[A] private (val capacity: Int, val length: Int,
    offset: Int, elems: Array[Any])
      extends immutable.Iterable[A] { self =>

  def this(capacity: Int) = this(capacity, length = 0,
    offset = 0, elems = Array.ofDim(capacity))

  def appended[B >: A](elem: B): Capped1[B] = {
    val newElems = Array.ofDim[Any](capacity)
    Array.copy(elems, 0, newElems, 0, capacity)
    val (newOffset, newLength) =
      if (length == capacity) {
        newElems(offset) = elem
        ((offset + 1) % capacity, length)
      } else {
        newElems(length) = elem
        (offset, length + 1)
      }
    new Capped1[B](capacity, newLength, newOffset, newElems)
  }

  @inline def :+ [B >: A](elem: B): Capped1[B] = appended(elem)

  def apply(i: Int): A =
    elems((i + offset) % capacity).asInstanceOf[A]

  def iterator: Iterator[A] = new AbstractIterator[A] {
    private var current = 0
    def hasNext = current < self.length
    def next(): A = {
      val elem = self(current)
      current += 1
      elem
    }
  }

  override def className = "Capped1"
}
```

Метод `appended` определяет, как элементы будут добавляться в заданную коллекцию `Capped1`: он создает новый массив элементов, копирует в него содержимое текущего массива и добавляет новый элемент. Если длина массива не превышает `capacity`, то новый элемент добавляется после предыдущего. Но в случае превы-

шения максимальной емкости новый элемент вставляется вместо первого элемента коллекции с индексом `offset`.

Метод `apply` реализует индексированный доступ: преобразует заданный индекс в соответствующий индекс внутреннего массива и возвращает элемент с этим индексом. Методы `appended` и `apply` делают коллекцию `Capped1` ограниченной.

Метод `iterator` делает обобщенные операции над коллекциями (такие как `foldLeft`, `count` и т. д.) совместимыми с `Capped1`. Он реализован с помощью индексированного доступа.

Наконец, мы переопределили метод `className`, чтобы он возвращал название коллекции, `Capped1`. Это название будет использоваться методом `toString`.

Вот несколько примеров взаимодействия с коллекцией `Capped1`:

```
scala> new Capped1(capacity = 4)
  res0: Capped1[Nothing] = Capped1()

scala> res0 :+ 1 :+ 2 :+ 3
  res1: Capped1[Int] = Capped1(1, 2, 3)

scala> res1.length
  res2: Int = 3

scala> res1.lastOption
  res3: Option[Int] = Some(3)

scala> res1 :+ 4 :+ 5 :+ 6
  res4: Capped1[Int] = Capped1(3, 4, 5, 6)

scala> res4.take(3)
  res5: collection.immutable.Iterable[Int] = List(3, 4, 5)
```

Как можно видеть в `res4`, в случае добавления в `Capped1` больше четырех элементов начальные элементы удаляются. Все операции ведут себя ожидаемым образом, за исключением последней: после вызова `take` вы получаете `List` вместо `Capped1`. Это объясняется тем, что `Capped1` расширяет трейт `Iterable` и наследует его метод `take`. Последний, в свою очередь, возвращает другой экземпляр `Iterable`, который по умолчанию реализован как `List`. Вот почему в последней строчке предыдущего примера вы получили `List`.

Разобравшись в ситуации, можно задать следующий вопрос: а как ее можно изменить? Один из способов предусматривает переопределение метода `take` в классе `Capped1`, возможно, таким вот образом:

```
override def take(count: Int): Capped1 = ...
```

Задача для `take` будет решена. А как же насчет `drop`, `filter` или `init`? Ведь существует более 50 методов работы с коллекциями, возвращающих другие коллекции. Если быть последовательными, то нужно переопределить все эти методы. Такой вариант все больше теряет свою привлекательность. К счастью, есть гораздо более простой способ достичь того же эффекта, показанный в следующем разделе.

Ограниченная коллекция, вторая версия

Класс ограниченной коллекции должен наследоваться не только от `Iterable`, но и от шаблонного трейта `IterableOps`. Это делает коллекция `Capped2`, представленная в листинге 25.6 и расширяющая `IterableOps[A, Capped2, Capped2[A]]`. Кроме того, ее член `iterableFactory` был переопределен и теперь возвращает `IterableFactory[Capped2]`. Как уже упоминалось в предыдущих разделах, трейт `IterableOps` реализует все конкретные методы `Iterable` в обобщенном виде. Например, результирующий тип методов `take`, `drop`, `filter` и `init` соответствует третьему параметру типа, переданному в `IterableOps`. Следовательно, в классе `Capped2` это `Capped2[A]`. Точно так же результирующий тип методов `map`, `flatMap` и `concat` определяется вторым параметром типа, переданным в `IterableOps`. В нашем случае это сам класс `Capped2`.

Листинг 25.6. Ограниченная коллекция, вторая версия

```
import scala.collection._

class Capped2[A] private (val capacity: Int, val length: Int,
    offset: Int, elems: Array[Any])
    extends immutable.Iterable[A]
    with IterableOps[A, Capped2, Capped2[A]] { self =>

  def this(capacity: Int) = // as before
  def appended[B >: A](elem: B): Capped2[B] = // as before
  @inline def :+ [B >: A](elem: B): Capped2[B] = // as before
  def apply(i: Int): A = // as before
  def iterator: Iterator[A] = // as before

  override def className = "Capped2"

  override val iterableFactory: IterableFactory[Capped2] =
    new Capped2Factory(capacity)

  override protected def fromSpecific(
      coll: IterableOnce[A]): Capped2[A] =
    iterableFactory.from(coll)

  override protected def newSpecificBuilder:
      mutable.Builder[A, Capped2[A]] =
    iterableFactory.newBuilder

  override def empty: Capped2[A] = iterableFactory.empty
}

class Capped2Factory(capacity: Int) extends
    IterableFactory[Capped2] {

  def from[A](source: IterableOnce[A]): Capped2[A] =
    (newBuilder[A] ++= source).result()
```

```
  def empty[A]: Capped2[A] = new Capped2[A](capacity)

  def newBuilder[A]: mutable.Builder[A, Capped2[A]] =
    new mutable.ImmutableBuilder[A, Capped2[A]](empty) {
      def addOne(elem: A): this.type = {
        elems = elems :+ elem
        this
      }
    }
}
```

Методы `fromSpecific` и `newSpecificBuilder` должны быть переопределены, поскольку реализации, унаследованные от супертрейта `Iterable`, возвращают `Iterable` вместо нужного нам типа `Capped2`. Еще одна операция, которая возвращает слишком общий тип, — `empty`. Мы переопределяем этот метод, чтобы он тоже возвращал `Capped2[A]`. Все эти переопределения попросту направляют вызовы к фабрике коллекций, на которую ссылается член `iterableFactory`, значением которого является экземпляр класса `Capped2Factory`.

Класс `Capped2Factory` предоставляет удобные фабричные методы для построения коллекций. В конечном счете они делегируют вызов операциям `empty`, которая создает пустой экземпляр `Capped2`, и `newBuilder`, использующей метод `appended` для наполнения коллекции `Capped2`.

В улучшенной реализации класса `Capped2` операции преобразования работают ожидаемым образом. Кроме того, класс `Capped2Factory` обеспечивает прозрачное приведение к `Capped2` других коллекций:

```
scala> object Capped extends Capped2Factory(capacity = 4)
  defined object Capped

scala> Capped(1, 2, 3)
  res0: Capped2[Int] = Capped2(1, 2, 3)

scala> res0.take(2)
  res1: Capped2[Int] = Capped2(1, 2)

scala> res0.filter(x => x % 2 == 1)
  res2: Capped2[Int] = Capped2(1, 3)

scala> res0.map(x => x * x)
  res3: Capped2[Int] = Capped2(1, 4, 9)

scala> List(1, 2, 3, 4, 5).to(Capped)
  res4: Capped2[Int] = Capped2(2, 3, 4, 5)
```

Теперь данная реализация ведет себя корректно, но мы можем еще немного улучшить ее. Наша коллекция строгая, поэтому мы можем добиться повышения производительности за счет использования строгих версий операций преобразования.

Ограниченная коллекция, итоговая версия

Итоговая версия класса `Capped` показана в листинге 25.7. Она примешивает трейт `StrictOptimizedIterableOps`, который переопределяет все операции преобразования, позволяя коллекции `Capped` пользоваться преимуществами строгих построителей. Эта версия переопределяет еще несколько операций для повышения производительности. Представление теперь задействует индексированный доступ, и ему делегируются вызовы итератора. Операция `knownSize` тоже переопределена, так как размер всегда известен.

Листинг 25.7. Коллекция Capped, итоговая версия

```
import scala.collection._

final class Capped[A] private (val capacity: Int, val length: Int,
    offset: Int, elems: Array[Any])
    extends immutable.Iterable[A]
    with IterableOps[A, Capped, Capped[A]]
    with StrictOptimizedIterableOps[A, Capped, Capped[A]] { self =>

  def this(capacity: Int) = this(capacity, length = 0,
    offset = 0, elems = Array.ofDim(capacity))

  def appended[B >: A](elem: B): Capped[B] = {
    val newElems = Array.ofDim[Any](capacity)
    Array.copy(elems, 0, newElems, 0, capacity)
    val (newOffset, newLength) =
      if (length == capacity) {
        newElems(offset) = elem
        ((offset + 1) % capacity, length)
      } else {
        newElems(length) = elem
        (offset, length + 1)
      }
    new Capped[B](capacity, newLength, newOffset, newElems)
  }

  @inline def :+ [B >: A](elem: B): Capped[B] = appended(elem)

  def apply(i: Int): A =
    elems((i + offset) % capacity).asInstanceOf[A]

  def iterator: Iterator[A] = view.iterator

  override def view: IndexedSeqView[A] = new IndexedSeqView[A] {
    def length: Int = self.length
    def apply(i: Int): A = self(i)
  }

  override def knownSize: Int = length
```

```
  override def className = "Capped"

  override val iterableFactory: IterableFactory[Capped] =
    new CappedFactory(capacity)

  override protected def fromSpecific(coll: IterableOnce[A]):
      Capped[A] = iterableFactory.from(coll)

  override protected def newSpecificBuilder:
      mutable.Builder[A, Capped[A]] =
    iterableFactory.newBuilder

  override def empty: Capped[A] = iterableFactory.empty
}

class CappedFactory(capacity: Int) extends IterableFactory[Capped] {

  def from[A](source: IterableOnce[A]): Capped[A] =
    source match {
      case capped: Capped[A] if capped.capacity == capacity => capped
      case _ => (newBuilder[A] ++= source).result()
    }

  def empty[A]: Capped[A] = new Capped[A](capacity)

  def newBuilder[A]: mutable.Builder[A, Capped[A]] =
    new mutable.ImmutableBuilder[A, Capped[A]](empty) {
      def addOne(elem: A): this.type = {
        elems = elems :+ elem
        this
      }
    }
}
```

Пример с классом `Capped` показал, что создание нового типа коллекций состоит из нескольких этапов. Помимо расширения подходящего типа, необходимо примешать нужный шаблон `Ops` и затем переопределить метод `iterableFactory` так, чтобы тот возвращал более конкретную фабрику. В завершение необходимо реализовать все абстрактные методы (такие как `iterator` в `Capped`).

Последовательности РНК

Возьмем еще один пример. Представьте, что вам нужно создать новый неизменяемый тип последовательности для цепочек РНК, представленных последовательностями оснований A (аденин), U (урацил), G (гуанин) и C (цитозин). Определения оснований можно легко сконфигурировать так, как показано в листинге 25.8.

Листинг 25.8. Основания РНК

```
abstract class Base
case object A extends Base
case object U extends Base
case object G extends Base
case object C extends Base

object Base {
  val fromInt: Int => Base = Array(A, U, G, C)
  val toInt: Base => Int = Map(A -> 0, U -> 1, G -> 2, C -> 3)
}
```

Каждое основание определено в виде `case`-объекта, унаследованного от общего абстрактного класса `Base`. У класса `Base` есть объект-компаньон, определяющий две функции, которые отображают основания на целые числа от 0 до 3. В наших примерах показаны два разных способа применения коллекций для реализации этих функций. Функция `toInt` реализована как отображение значений `Base` на целые числа. Обратная функция, `fromInt`, реализована в виде массива. Здесь мы используем тот факт, что и отображения, и массивы *являются* функциями, поскольку унаследованы от трейта `Function1`.

Дальше необходимо определить класс для цепочек РНК. На концептуальном уровне цепочка представляет собой всего лишь `Seq[Base]`. Однако цепочки РНК бывают довольно длинными, поэтому имеет смысл позаботиться об их компактном представлении. Поскольку оснований всего четыре, каждое из них можно определить с помощью двух бит, поэтому в целом числе можно хранить 16 двухбитных оснований. Идея в том, что мы должны создать специальный подкласс `Seq[Base]`, который будет использовать это упакованное представление.

Класс для цепочек РНК, первая версия

В листинге 25.9 показана первая версия этого класса, которая позже будет усовершенствована.

Листинг 25.9. Класс для цепочки РНК, первая версия

```
import collection.mutable
import collection.immutable.{IndexedSeq, IndexedSeqOps}

final class RNA1 private (val groups: Array[Int],
  val length: Int) extends IndexedSeq[Base]
  with IndexedSeqOps[Base, IndexedSeq, RNA1] {

  import RNA1._

  def apply(idx: Int): Base = {
    if (idx < 0 || length <= idx)
      throw new IndexOutOfBoundsException
    Base.fromInt(groups(idx / N) >> (idx % N * S) & M)
  }
```

```
  override def className = "RNA1"
  override protected def fromSpecific(
      source: IterableOnce[Base]): RNA1 =
    fromSeq(source.iterator.toSeq)
  override protected def newSpecificBuilder:
      mutable.Builder[Base, RNA1] =
    iterableFactory.newBuilder[Base].mapResult(fromSeq)
  override def empty: RNA1 = fromSeq(Seq.empty)
}

object RNA1 {

  // Количество битов, необходимое для представления группы
  private val S = 2
  // Количество групп, которые помещаются в Int
  private val N = 32 / S
  // Битовая маска для выделения группы
  private val M = (1 << S) 1

  def fromSeq(buf: collection.Seq[Base]): RNA1 = {
    val groups = new Array[Int]((buf.length + N - 1) / N)
    for (i <- 0 until buf.length)
      groups(i / N) |= Base.toInt(buf(i)) << (i % N * S)
    new RNA1(groups, buf.length)
  }

  def apply(bases: Base*) = fromSeq(bases)
}
```

У класса `RNA1` есть конструктор, который принимает в качестве своего первого аргумента целочисленный массив. Этот массив содержит упакованные данные РНК, с 16 основаниями в каждом элементе; исключение составляет последний элемент — массив, который может быть заполнен частично. Второй аргумент, `length`, определяет общее количество оснований в массиве (и последовательности). Класс `RNA1` расширяет `IndexedSeq[Base]` и `IndexedSeqOps[Base, IndexedSeq, RNA1]`. Эти трейты определяют два абстрактных метода, `length` и `apply`, которые должны быть реализованы в конкретных подклассах.

Класс `RNA1` реализует `length` автоматически, определяя параметрическое поле (см. раздел 10.3) с тем же именем. Для реализации индексирующего метода `apply` сначала извлекается целое значение из массива `groups`, а затем из этого значения берется подходящее двухбитное число с помощью сдвига вправо (`>>`) и маски (`&`). Приватные константы `S`, `N` и `M` предоставляются объектом-компаньоном `RNA1`. Константа `S` определяет размер каждого пакета (то есть два); `N` определяет количество двухбитных пакетов в каждом целом числе; а `M` — это битовая маска, которая выделяет в слове `S` младшие биты.

Класс `RNA1` также переопределяет методы `fromSpecific` и `newSpecificBuilder`, которые используются операциями преобразования, такими как `filter` и `take`. Реализация `fromSpecific` направляет вызовы к методу `fromSeq` объекта-компаньона `RNA1`.

Для реализации `newSpecificBuilder` используется построитель `IndexedSeq` по умолчанию, результат которого преобразуется в `RNA1` с помощью метода `mapResult`.

Обратите внимание: конструктор класса `RNA1` является приватным. Это значит, что клиенты не могут создавать последовательности `RNA1` путем вызова `new`; это логичное решение, которое позволяет скрыть от пользователя упакованные массивы, лежащие в основе `RNA1`. Если клиенты не могут видеть детали представления цепочек РНК, то это позволяет менять различные аспекты данного представления в любой момент времени, не нарушая работу клиентского кода.

Иными словами, данный подход обеспечивает хорошее разделение интерфейса цепочек РНК и их реализации. Однако если создание цепочки РНК с помощью ключевого слова `new` не разрешено, то вы должны предусмотреть какую-то альтернативу, иначе весь класс окажется довольно бесполезным.

На самом деле существует два альтернативных метода создания цепочки РНК, предоставляемые объектом-компаньоном `RNA1`. Первый метод, `fromSeq`, преобразует заданную последовательность оснований (то есть значение типа `Seq[Base]`) в экземпляр класса `RNA1`. Для этого метод `fromSeq` упаковывает все основания, содержащиеся в его последовательности аргументов, в массив и затем передает его в приватный конструктор `RNA1` вместе с длиной исходной последовательности. Здесь мы полагаемся на тот факт, что приватный конструктор класса виден только в его объекте-компаньоне.

Второй способ создания значения `RNA1` возможен благодаря методу `apply` объекта `RNA1`.Он принимает переменное количество аргументов типа `Base` и просто перенаправляет их в виде последовательности в метод `fromSeq`.

Вот как эти две схемы создания выглядят на практике:

```
scala> RNA1.fromSeq(List(A, G, U, A))
res1: RNA1 = RNA1(A, G, U, A)

scala> val rna1 = RNA1(A, U, G, G, C)
rna1: RNA1 = RNA1(A, U, G, G, C)
```

Примешивая `IndexedSeqOps`, класс `RNA1` указывает в качестве параметров типов `Base`, `IndexedSeq` и `RNA1`. Первый параметр определяет `Base` в качестве типа элементов. Второй выбирает `IndexedSeq` в качестве конструктора типа, который будет использоваться операциями преобразования, возвращающими коллекцию с разными типами элементов. Третий параметр определяет результирующий тип, возвращаемый операциями преобразования, то есть коллекции с элементами одного типа.

Обратите внимание: первый параметр — `IndexedSeq`, а второй — `RNA1`. Это значит, такие операции, как `take` или `filter`, будут возвращать `RNA1`:

```
scala> rna1.take(3)
res2: RNA1 = RNA1(A, U, G)

scala> rna1.filter(_ != U)
res3: RNA1 = RNA1(A, G, G, C)
```

А вот результатом операций вроде `map` или `++` будет `IndexedSeq`:

```
scala> rna1.map(base => base)
res7: IndexedSeq[Base] = Vector(A, U, G, G, C)

scala> rna1 ++ rna1
res8: IndexedSeq[Base] = Vector(A, U, G, G, C, A, U, G, G, C)
```

Каким образом следует адаптировать для цепочек РНК такие методы, как `map` и `++`? Мы хотим, чтобы при отображении оснований на основания или конкатенации двух цепочек с помощью `++` получалась новая цепочка РНК. С другой стороны, при отображении оснований на какой-то другой тип не может получиться еще одна цепочка РНК, поскольку новые элементы будут иметь не тот тип. Вместо этого должна возвращаться последовательность. Точно так же добавление к цепочке РНК элементов любого типа, кроме `Base`, создает общую последовательность, а не еще одну цепочку.

В контексте `RNA1` результаты выполнения `map` и `++` никогда не имеют тип `RNA1`, даже если в качестве типа элементов сгенерированной коллекции используется `Base`. Чтобы понять, как это исправить, взгляните на сигнатуру метода `map` (или `++`, который имеет похожую сигнатуру). Он изначально определен в классе `scala.collection.IterableOps` со следующей сигнатурой:

```
def map[B](f: A => B): CC[B]
```

В `RNA1` типом элементов коллекции является `A`, а `CC` — это конструктор типа, который передается в качестве второго параметра трейту `IterableOps`. В реализации `RNA1` конструктором типа `CC` является `IndexedSeq`. Вот почему в качестве результирующего типа `map` вы получаете `IndexedSeq`.

Класс для цепочек РНК, вторая версия

В целях улучшения ситуации можно перегрузить методы, которые возвращают `CC[B]`, и сделать так, чтобы они возвращали тип цепочки РНК, если известно, что `B` — это `Base`. Перегрузке подлежат методы `appended`, `appendedAll`, `concat`, `flatMap`, `map`, `prepended`, `prependedAll` и `++`. Класс `RNA2`, показанный в листинге 25.10, перегружает эти методы для цепочек РНК.

Листинг 25.10. Класс для цепочек РНК, вторая версия

```
import scala.collection.{View, mutable}
import scala.collection.immutable.{IndexedSeq, IndexedSeqOps}

final class RNA2 private (val groups: Array[Int],
    val length: Int) extends IndexedSeq[Base]
    with IndexedSeqOps[Base, IndexedSeq, RNA2] {

  import RNA2._

  def apply(idx: Int): Base = // as before
```

```
  override def className = "RNA2"

  override protected def fromSpecific(
    source: IterableOnce[Base]): RNA2 = // as before

  override protected def newSpecificBuilder:
    mutable.Builder[Base, RNA2] = // as before

  override def empty: RNA2 = // as before

  // Перегружаем методы, чтобы возвращать RNA2
  def appended(base: Base): RNA2 = fromSpecific(
    new View.Append(this, base))

  def appendedAll(suffix: IterableOnce[Base]): RNA2 =
    concat(suffix)

  def concat(suffix: IterableOnce[Base]): RNA2 =
    fromSpecific(this.iterator ++ suffix.iterator)

  def flatMap(f: Base => IterableOnce[Base]): RNA2 =
    fromSpecific(new View.FlatMap(this, f))

  def map(f: Base => Base): RNA2 =
    fromSpecific(new View.Map(this, f))

  def prepended(base: Base): RNA2 = fromSpecific(
    new View.Prepend(base, this))

  def prependedAll(prefix: IterableOnce[Base]): RNA2 =
    fromSpecific(prefix.iterator ++ this.iterator)

  // Символический псевдоним для 'concat'
  @inline final def ++ (suffix: IterableOnce[Base]): RNA2 =
    concat(suffix)
}
```

Теперь в нашей реализации эти методы ведут себя так, как нам нужно:

```
scala> val rna2 = RNA2(A, U, G, G, C)
rna2: RNA2 = RNA2(A, U, G, G, C)

scala> rna1.map(base => base)
res2: RNA2 = RNA2(A, U, G, G, C)

scala> rna1 ++ rna1
res3: RNA2 = RNA2(A, U, G, G, C, A, U, G, G, C)
```

Эта реализация лучше предыдущей, но в ней тоже можно поправить несколько вещей.

Наша коллекция строгая, поэтому мы можем воспользоваться преимуществами улучшенной производительности, которую предлагают строгие построители в операциях преобразования. Кроме того, если мы попытаемся преобразовать `Iterable[Base]` в `RNA2`, то у нас ничего не получится:

```
scala> val bases: Iterable[Base] = List(A, U, C, C)
bases: Iterable[Base] = List(A, U, C, C)

scala> bases.to(RNA2)
               ^
     error: type mismatch;
      found   : RNA2.type
      required: scala.collection.Factory[Base,?]
```

Класс для цепочек РНК, итоговая версия

Итоговая версия класса `RNA`, показанная в листинге 25.11, расширяет трейт `StrictOptimizedSeqOps`.

Листинг 25.11. Класс для цепочек РНК, итоговая версия

```
import scala.collection.{AbstractIterator,
  SpecificIterableFactory, StrictOptimizedSeqOps, View, mutable}
import scala.collection.immutable.{IndexedSeq, IndexedSeqOps}

final class RNA private (val groups: Array[Int],
    val length: Int) extends IndexedSeq[Base]
    with IndexedSeqOps[Base, IndexedSeq, RNA]
    with StrictOptimizedSeqOps[Base, IndexedSeq, RNA] { rna =>

  import RNA._

  // Обязательная реализация 'apply' в 'IndexedSeqOps'
  def apply(idx: Int): Base = {
    if (idx < 0 || length <= idx)
      throw new IndexOutOfBoundsException
    Base.fromInt(groups(idx / N) >> (idx % N * S) & M)
  }

  override def className = "RNA"

  override protected def fromSpecific(
      source: IterableOnce[Base]): RNA =
    RNA.fromSpecific(source)

  override protected def newSpecificBuilder:
      mutable.Builder[Base, RNA] =
    RNA.newBuilder

  override def empty: RNA = RNA.empty
```

```
  // Перегружаем методы, чтобы возвращать RNA
  def appended(base: Base): RNA =
    (newSpecificBuilder ++= this += base).result()

  def appendedAll(suffix: IterableOnce[Base]): RNA =
    strictOptimizedConcat(suffix, newSpecificBuilder)

  def concat(suffix: IterableOnce[Base]): RNA =
    strictOptimizedConcat(suffix, newSpecificBuilder)

  def flatMap(f: Base => IterableOnce[Base]): RNA =
    strictOptimizedFlatMap(newSpecificBuilder, f)

  def map(f: Base => Base): RNA =
    strictOptimizedMap(newSpecificBuilder, f)

  def prepended(base: Base): RNA =
    (newSpecificBuilder += base ++= this).result()

  def prependedAll(prefix: Iterable[Base]): RNA =
    (newSpecificBuilder ++= prefix ++= this).result()

  @inline final def ++ (suffix: IterableOnce[Base]): RNA =
    concat(suffix)

  // Необязательное переопределение реализации итератора
  // для повышения его эффективности
  override def iterator: Iterator[Base] =
    new AbstractIterator[Base] {
      private var i = 0
      private var b = 0
      def hasNext: Boolean = i < rna.length
      def next(): Base = {
        b = if (i % N == 0) groups(i / N) else b >>> S
        i += 1
        Base.fromInt(b & M)
      }
    }
}

object RNA extends SpecificIterableFactory[Base, RNA] {

  private val S = 2            // Количество битов в группе
  private val M = (1 << S) - 1 // Битовая маска для выделения группы
  private val N = 32 / S       // Количество групп в Int

  def fromSeq(buf: collection.Seq[Base]): RNA = {
    val groups = new Array[Int]((buf.length + N - 1) / N)
    for (i <- 0 until buf.length)
      groups(i / N) |= Base.toInt(buf(i)) << (i % N * S)
  new RNA(groups, buf.length)
}
```

```
  // Реализуем фабричные методы, которые требует
  // SpecificIterableFactory
  def empty: RNA = fromSeq(Seq.empty)

  def newBuilder: mutable.Builder[Base, RNA] =
    mutable.ArrayBuffer.newBuilder[Base].mapResult(fromSeq)

  def fromSpecific(it: IterableOnce[Base]): RNA = it match {
    case seq: collection.Seq[Base] => fromSeq(seq)
    case _ => fromSeq(mutable.ArrayBuffer.from(it))
  }
}
```

Трейт `StrictOptimizedSeqOps` переопределяет все операции преобразования, чтобы воспользоваться преимуществами строгих построителей. Более того, в этой версии используются вспомогательные операции, предоставляемые трейтом `StrictOptimizedSeqOps`, такие как `strictOptimizedConcat`, которые делают так, чтобы перегруженные методы преобразования возвращали коллекцию `RNA` (включая `map` и `++`) более эффективным способом.

Чтобы упростить превращение любой коллекции оснований в `RNA`, объект-компаньон этого класса расширяет `SpecificIterableFactory[Base, RNA]`. Благодаря этому объект-компаньон `RNA` можно использовать в качестве параметра для вызова `to` в контексте `Seq`:

```
scala> List(U, A, G, C).to(RNA)
res0: RNA(U, A, G, C)
```

Трейт `SpecificIterableFactory[Base, RNA]` определяет три абстрактных метода: `empty`, `newBuilder` и `fromSpecific`. Объект `RNA` их реализует, а класс `RNA` определяет в их контексте собственные методы построения `fromSpecific` и `newSpecific`.

До сих пор мы фокусировались на минимальном количестве определений для новых последовательностей, методы которых соответствуют определенным типам. Однако на практике может также возникнуть необходимость в добавлении новой функциональности или переопределении существующих методов для повышения их эффективности. Примером этого служит переопределенный метод `iterator` в классе `RNA`. Он сам по себе имеет большое значение, поскольку в нем реализуется перебор коллекций. Кроме того, многие другие методы коллекции реализованы на основе `iterator`. Поэтому имеет смысл приложить усилия к оптимизации реализации данного метода.

Стандартная реализация `iterator` в `IndexedSeq` станет просто выбирать каждый `i`-й элемент коллекции с помощью `apply`, где `i` будет относиться к диапазону от нуля до длины коллекции минус единица. То есть эта стандартная реализация для каждого элемента цепочки РНК выбирает элемент массива и распаковывает из него основание. Переопределение `iterator` в классе `RNA` ведет себя более рационально. Заданная функция тут же применяется ко всем содержащимся в выбранном элементе массива основаниям. Таким образом затраты на выбор из массива и распаковку битов существенно сокращаются.

Префиксные отображения

В качестве третьего примера вы освоите способ интеграции в фреймворк коллекций новой разновидности изменяемого отображения. Замысел заключается в реализации изменяемого отображения с `String` в качестве типа ключей дерева Patricia[1]. Термин *Patricia* — аббревиатура от Practical Algorithm to Retrieve Information Coded in Alphanumeric (практический алгоритм получения информации, закодированной алфавитно-цифровыми символами). Идея состоит в хранении множества или отображения в виде дерева, в котором последующие символы в ключе поиска уникальным образом определяют дерево-потомок.

Например, дерево Patricia, в котором хранятся пять строк: `"abc"`, `"abd"`, `"al"`, `"all"`, `"xy"`, будет выглядеть как дерево, показанное на рис. 25.2. Чтобы в этом дереве найти узел, соответствующий строке `"abc"`, нужно просто проследовать к поддереву с меткой `"a"`, перейти оттуда к дереву с меткой `"b"`, чтобы в конечном счете добраться до поддерева, имеющего метку `"c"`. Если дерево Patricia используется в качестве отображения, то значение, связанное с ключом, сохраняется в узлах, добраться до которых можно будет с помощью ключа. Если это множество, то просто сохраняется метка, сообщающая о том, что узел присутствует во множестве.

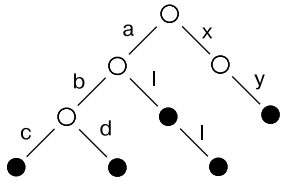


Рис. 25.2. Пример дерева Patricia

В деревьях Patricia поддерживаются весьма эффективные способы поиска и обновления. Еще одной приятной особенностью является то, что они поддерживают выбор подколлекции по заданному префиксу. Например, в дереве, изображенном на рис. 25.2, можно получить подколлекцию всех ключей, начинающихся с `"a"`, просто поставив после корня дерева ссылку `"a"`.

Теперь, основываясь на этих идеях, мы проведем вас через реализацию отображения в виде дерева Patricia. Отображение назовем `PrefixMap`, что будет означать: в нем предоставляется метод `withPrefix`, котрый выбирает подотображение всех ключей, начинающихся с заданного префикса.

Сначала определим префиксное отображение с ключами, показанными на рис. 25.2:

```
scala> val m = PrefixMap("abc" -> 0, "abd" -> 1, "al" -> 2,
  "all" -> 3, "xy" -> 4)
m: PrefixMap[Int] = Map((abc,0), (abd,1), (al,2), (all,3),
  (xy,4))
```

[1] *Morrison D. R.* PATRICIA — Practical Algorithm To Retrieve Information Coded in Alphanumeric [Электронный ресурс] / D. R. Morrison. J. ACM, 15(4):514–534, 1968. ISSN 0004-5411. — Режим доступа: doi:http://doi.acm.org/10.1145/321479.321481.

Затем вызов `withPrefix` в отношении `m` выдаст еще одно префиксное отображение:

```
scala> m withPrefix "a"
res14: PrefixMap[Int] = Map((bc,0), (bd,1), (l,2), (ll,3))
```

В листинге 25.12 показано определение `PrefixMap`. Оно связывает строки с любым типом значений, поэтому у него есть параметр типа `A` для значений отображения. Класс расширяет `mutable.Map` и `MapOps` по аналогии с тем, как это делалось в примере с цепочками РНК. Класс-реализация `MapOps` наследуется для того, чтобы получить подходящий результирующий тип в таких преобразованиях, как `filter` и `take`.

Листинг 25.12. Реализация префиксных отображений с помощью деревьев Patricia

```
import scala.collection._

class PrefixMap[A] extends mutable.Map[String, A]
    with mutable.MapOps[String, A, mutable.Map, PrefixMap[A]]
    with StrictOptimizedIterableOps
    [(String, A), mutable.Iterable, PrefixMap[A]] {

  private var suffixes: immutable.Map[Char, PrefixMap[A]] =
    immutable.Map.empty

  private var value: Option[A] = None

  def get(s: String): Option[A] =
    if (s.isEmpty) value
    else suffixes get (s(0)) flatMap (_.get(s substring 1))

  def addOne(kv: (String, A)): this.type = {
    withPrefix(kv._1).value = Some(kv._2)
    this
  }

  def subtractOne(s: String): this.type = {
    if (s.isEmpty) { val prev = value; value = None; prev }
    else suffixes get (s(0)) flatMap (_.remove(s substring 1))
    this
  }

  def withPrefix(s: String): PrefixMap[A] =
    if (s.isEmpty) this
    else {
      val leading = s(0)
      suffixes get leading match {
        case None =>
          suffixes = suffixes + (leading -> empty)
        case _ =>
      }
      suffixes(leading) withPrefix (s substring 1)
    }
```

```
  def iterator: Iterator[(String, A)] =
    (for (v <- value.iterator) yield ("", v)) ++
      (for ((chr, m) <- suffixes.iterator;
            (s, v) <- m.iterator) yield (chr +: s, v))

  override def className = "PrefixMap"

  // Перегружаем методы, чтобы возвращать PrefixMap
  def map[B](f: ((String, A)) => (String, B)): PrefixMap[B] =
    strictOptimizedMap(PrefixMap.newBuilder[B], f)

  def flatMap[B](f: ((String, A)) =>
      IterableOnce[(String, B)]): PrefixMap[B] =
    strictOptimizedFlatMap(PrefixMap.newBuilder[B], f)

  // Перегружаем метод concat, чтобы уточнить его возвращаемый тип
  override def concat[B >: A](suffix:
      Iterable[(String, B)]): PrefixMap[B] =
    strictOptimizedConcat(suffix, PrefixMap.newBuilder[B])

  // Члены, объявленные в scala.collection.mutable.Clearable
  def clear(): Unit = suffixes = immutable.Map.empty

  // Члены, объявленные в scala.collection.IterableOps
  override protected def fromSpecific(
      source: IterableOnce[(String, A)]): PrefixMap[A] =
    PrefixMap.from(coll)

  override protected def newSpecificBuilder:
      mutable.Builder[(String, A), PrefixMap[A]] =
    PrefixMap.newBuilder

  override def empty: PrefixMap[A] = new PrefixMap
}
```

Узел префиксного отображения имеет два изменяемых поля: `suffixes` и `value`. Поле `value` содержит связанное с этим узлом `Option`-значение. Оно инициализировано в `None`. Поле `suffixes` содержит отображение символов на `PrefixMap`-значения. Оно инициализировано пустым отображением. Может возникнуть вопрос: почему в качестве типа реализации `suffixes` было выбрано неизменяемое отображение? Не будет ли более подходящим изменяемое отображение, поскольку `PrefixMap` в целом является изменяемым? Дело в том, что работа с неизменяемыми отображениями, содержащими всего несколько элементов, очень эффективна как по расходу пространства памяти, так и по времени выполнения.

Например, отображения, содержащие менее пяти элементов, представляются в виде отдельного объекта. В отличие от этого, как говорилось в разделе 17.2, стандартным изменяемым отображением является `HashMap`, которое обычно занимает 80 байт памяти, даже будучи пустым. Следовательно, когда чаще всего используются небольшие коллекции, лучше делать выбор в пользу неизменяемых

отображений. В случае с деревом Patricia мы ожидаем, что большинство узлов, за исключением тех, которые находятся на самой верхушке дерева, будет иметь совсем немного потомков. Хранить эти потомки в неизменяемом отображении будет, скорее всего, более эффективно.

Теперь посмотрим на первый метод, который нужно реализовать для работы с отображением, — `get`. Алгоритм следующий: чтобы получить значение, связанное в префиксном отображении с пустой строкой, нужно просто выбрать сохраненное в корне дерева значение `Option`. В противном случае, если строка ключа не пуста, нужно попробовать выбрать подотображение, соответствующее первому символу строки. Если это выдаст отображение, то нужно продолжить поиск по тому, что останется в строке ключей данного отображения после ее первого символа. Если выбор окажется неудачным, то ключ не хранится в отображении и возвращается значение `None`. Комбинированный выбор в отношении значения `Option` можно вполне изящно выразить с помощью `flatMap`. В случае применения к `Option ov` и указания замыкания `f`, которое, в свою очередь, возвращает `Option`, выражение `ov flatMap f` будет успешно вычислено, если и `ov`, и `f` вернут определенное значение. В противном случае выражение `ov flatMap f` возвратит `None`.

Следующими двумя методами, реализуемыми для работы с изменяемым отображением, будут `addOne` и `subtractOne`. Метод `subtractOne` очень похож на `get`, за исключением того, что перед возвращением любого связанного с ним значения для поля, содержащего это значение, устанавливается значение `None`. Метод `addOne` сначала вызывает `withPrefix` для перехода по дереву к тому узлу, который нуждается в обновлении, а затем устанавливает для поля `value` этого узла заданное значение. Метод `withPrefix` выполняет переход по дереву, создавая при необходимости подотображения, если какой-то префикс символов еще не содержится в дереве в качестве пути.

Последний абстрактный метод, реализуемый для изменяемого отображения, — `iterator`. Он требует создания итератора, который выдает все пары «ключ — значение», хранящиеся в отображении. Для любого заданного префиксного отображения этот итератор состоит из следующих частей: если отображение содержит в поле `value` своего корня определенное значение `Some(x)`, то первым элементом, возвращенным из итератора, будет `("", x)`. Более того, итератор нуждается в проходе итераторов всех подотображений, сохраненных в поле `suffixes`, при этом нужно добавить символ перед каждой строкой ключа, возвращенной этими итераторами. Точнее, если `m` является подотображением, достигнутым от корня посредством символа `chr`, а `(s, v)` — элемент, возвращенный итератором `m.iterator`, то итератор корня возвратит `(chr +: s, v)`.

Эта логика в реализации метода `iterator`, показанной выше, в листинге 25.15, представлена довольно кратко в виде объединения двух выражений `for`. Первое выражение выполняет обход `value.iterator`. Здесь используется то обстоятельство, что в `Option`-значениях определяется метод итерации, который либо вообще не возвращает элемент, если значением `Option` является `None`, либо возвращает ровно один элемент `x`, если значением `Option` является `Some(x)`.

`PrefixMap` переопределяет `fromSpecific` и `newSpecificBuilder`, чтобы привести результат к более конкретному типу `PrefixMap`. По умолчанию реализации этих методов возвращают `mutable.Map` вместо нужного нам типа `PrefixMap`.

Теперь обратимся к объекту-компаньону `PrefixMap`, показанному в листинге 25.13. На самом деле абсолютной необходимости в определении этого объекта-компаньона не было, поскольку класс `PrefixMap` сам в состоянии справиться со всеми задачами. Главное предназначение объекта `PrefixMap` — определение некоторых удобных фабричных методов. Он также определяет прозрачное приведение к `Factory` для более удобного взаимодействия с другими коллекциями. Приведение, к примеру, осуществляется при записи `List("foo" -> 3).to(PrefixMap)`. Данная операция принимает `Factory` в качестве параметра, но при этом объект-компаньон `PrefixMap` не расширяет `Factory` (он не может этого делать, поскольку в `Factory` типы элементов коллекции зафиксированы, тогда как в `PrefixMap` типы значений полиморфны).

Листинг 25.13. Объект-компаньон для префиксных отображений

```
import scala.collection._
import scala.language.implicitConversions

object PrefixMap {
  def empty[A] = new PrefixMap[A]

  def from[A](source: IterableOnce[(String, A)]): PrefixMap[A] =
    source match {
      case pm: PrefixMap[A] => pm
      case _ => (newBuilder[A] ++= source).result()
    }

  def apply[A](kvs: (String, A)*): PrefixMap[A] = from(kvs)

  def newBuilder[A]: mutable.Builder[(String, A), PrefixMap[A]] =
    new mutable.GrowableBuilder[(String, A), PrefixMap[A]](empty)

  implicit def toFactory[A](
      self: this.type): Factory[(String, A), PrefixMap[A]] =
    new Factory[(String, A), PrefixMap[A]] {
      def fromSpecific(source:
        IterableOnce[(String, A)]): PrefixMap[A] =
        self.from(source)
      def newBuilder: mutable.Builder[(String, A), PrefixMap[A]] =
        self.newBuilder
    }
}
```

Двумя вспомогательными методами являются `empty` и `apply`. Во фреймворке коллекций Scala для всех других коллекций имеются точно такие же методы, поэтому был смысл определить их и в данном объекте. Располагая этими двумя методами, литералы `PrefixMap` можно записывать точно так же, как это делалось для любых других коллекций:

```
scala> PrefixMap("hello" -> 5, "hi" -> 2)
  res0: PrefixMap[Int] = PrefixMap(hello -> 5, hi -> 2)

scala> res0 += "foo" -> 3
  res1: res0.type = PrefixMap(hello -> 5, hi -> 2, foo -> 3)
```

Краткие выводы

Если вам нужна полная интеграция во фреймворк нового класса коллекции, то обратите внимание на следующие моменты.

1. Решите, какой должна быть коллекция: изменяемой или неизменяемой.
2. Подберите для коллекции подходящие базовые трейты.
3. Наследуйте от правильного шаблонного трейта, чтобы реализовать большинство операций коллекции.
4. Перегружайте необходимые операции, которые не возвращают по умолчанию максимально специализированные коллекции.

Резюме

Теперь вы увидели, как создаются коллекции Scala и как можно добавлять новые разновидности коллекций. Благодаря сильной поддержке абстракции в Scala каждый новый тип коллекции располагает большим количеством методов, не требуя их повторной реализации.

26 Экстракторы

Вы, должно быть, уже привыкли к тому, как лаконично можно выполнить декомпозицию и анализ данных, воспользовавшись сопоставлением с образцом. В этой главе мы покажем, как придать данной концепции более обобщенный характер. До сих пор паттерны конструкторов были связаны с `case`-классами. Например, `Some(x)` — вполне допустимый паттерн, поскольку `Some` — это `case`-класс. Вероятно, вы захотите создавать подобные паттерны без ассоциированных с ними `case`-классов. Вам может потребоваться возможность создания собственных разновидностей паттернов. Именно такая возможность и предоставляется экстракторами. В этой главе мы рассмотрим, что представляют собой экстракторы и как ими можно воспользоваться в целях определения паттернов, не связанных с представлением объекта.

26.1. Пример извлечения адресов электронной почты

Чтобы показать задачу, решаемую с помощью экстракторов, предположим, что нужно проанализировать строки, представляющие адреса электронной почты. Следует решить, есть ли в заданной строке адрес электронной почты, и если да, то получить доступ к частям адреса, представляющим собой имя пользователя и домен. Традиционный способ решения данной задачи предусматривает использование трех вспомогательных функций:

```
def isEMail(s: String): Boolean
def domain(s: String): String
def user(s: String): String
```

Располагая ими, можно проанализировать заданную строку `s`:

```
if (isEMail(s)) println(user(s) + " AT " + domain(s))
else println("not an email address")
```

Этот код работает, но выглядит несколько неуклюже. Более того, если понадобится объединить несколько таких проверок, то все станет намного сложнее.

Например, может понадобиться найти две смежные строки в списке при условии, что обе они являются адресами электронной почты одного и того же пользователя. Можете сами попробовать применить ранее определенные функции доступа, чтобы посмотреть, во что все это выльется.

В главе 15 уже было показано, что для решения таких задач идеально подходит сопоставление с образцом. Представим на минуту, что можно сопоставить строку со следующим паттерном:

```
EMail(user, domain)
```

Паттерн будет совпадать, если строка содержит символ «собачки» (@). В таком случае она привяжет переменную `user` к той части строки, которая находится перед @, а переменную `domain` — к части, находящейся после этого символа. Если сделать такой паттерн условием, то предыдущее выражение можно прописать более четко:

```
s match {
  case EMail(user, domain) => println(user + " AT " + domain)
  case _ => println("not an email address")
}
```

Более сложная задача нахождения двух смежных адресов электронной почты одного и того же пользователя будет сведена к следующему паттерну:

```
ss match {
  case EMail(u1, d1) :: EMail(u2, d2) :: _ if (u1 == u2) => ...
  ...
}
```

Его читать гораздо удобнее, чем все то, что можно было бы написать с помощью функций доступа. Но есть проблема: строки не являются `case`-классами — у них нет представления, соответствующего `EMail(user, domain)`. И тут на помощь приходят экстракторы Scala: они позволяют определять новые паттерны для существующих типов, где паттерн не обязан следовать внутреннему представлению типа.

26.2. Экстракторы

Экстрактор в Scala представляет собой объект, один из членов которого — метод `unapply`. Данный метод служит для сопоставления значения и деления его на части. Зачастую объект-экстрактор также определяет дуальный метод `apply` для создания значений, но это необязательное условие. В качестве примера в листинге 26.1 показан объект-экстрактор для адресов электронной почты.

Листинг 26.1. Объект-экстрактор Email-строк

```
object EMail {

  // Метод-инъектор (необязательный)
  def apply(user: String, domain: String) = user + "@" + domain
```

```
  // Метод-экстрактор (обязательный)
  def unapply(str: String): Option[(String, String)] = {
    val parts = str split "@"
    if (parts.length == 2) Some(parts(0), parts(1)) else None
  }
}
```

В этом объекте определены оба метода, и `apply`, и `unapply`. Метод `apply` имеет то же самое предназначение, что и всегда: превращает `EMail` в объект, который может быть применен к аргументам в круглых скобках точно так же, как применяется метод. Следовательно, создать строку `John@epfl.ch` можно с помощью записи `EMail("John", "epfl.ch")`. Чтобы прояснить ситуацию, можно также позволить `Email` воспользоваться наследованием из функционального типа Scala:

```
object EMail extends ((String, String) => String) { ... }
```

ПРИМЕЧАНИЕ

Та часть предыдущего объявления объекта, которая имеет вид (String, String) => String, означает то же самое, что и выражение Function2[String, String, String], объявляющее абстрактный метод apply, реализуемый EMail. В результате этого объявления можно будет, к примеру, передать EMail тому методу, который ожидает тип Function2[String, String, String].

Метод `unapply` и есть именно то, что превращает `EMail` в экстрактор. В определенном смысле он реверсирует процесс создания, выполняемый `apply`. Если `apply` получает две строки и формирует из них строку адреса электронной почты, то `unapply` получает адрес и потенциально возвращает две строки: `user` и `domain`, извлеченные из него. Но `unapply` должен также справляться со случаями, когда заданная строка не является адресом электронной почты. Именно поэтому `unapply` в отношении пары строк возвращает тип `Option`. Результат реализации данного метода — либо `Some(user, domain)`, если строка `str` представляет собой адрес электронной почты, с заданными частями, обозначающими пользователя и домен[1], либо `None`, если `str` не является адресом электронной почты. Ниже представлены несколько примеров:

```
unapply("John@epfl.ch") равно Some("John", "epfl.ch")
unapply("John Doe") равно None
```

Теперь каждый раз, когда сопоставление с образцом сталкивается с паттерном, ссылающимся на объект-экстрактор, в отношении выражения выбора вызывается принадлежащий экстрактору метод `unapply`. Например, выполнение этого кода:

```
selectorString match { case EMail(user, domain) => ... }
```

приведет к такому вызову:

```
EMail.unapply(selectorString)
```

[1] Как здесь показано, когда Some применяется к кортежу (user, domain), пару круглых скобок при передаче кортежа функции, принимающей один аргумент, можно не ставить. То есть Some(user, domain) означает то же самое, что и Some((user, domain)).

Как было показано ранее, этот вызов `EMail.unapply` вернет либо `None`, либо `Some(u, d)`, где `u` будет пользовательской частью адреса, а `d` — доменной. В случае `None` паттерн не подходит, и система попробует другой паттерн или даст сбой с генерацией исключения `MatchError`. В случае `Some(u, d)` паттерн подошел и его переменные будут привязаны к элементам возвращаемого значения. В предыдущем сопоставлении переменная `user` будет привязана к `u`, а переменная `domain` — к `d`.

В примере сопоставления с образцом для `EMail` тип `String` выражения селектора, `selectorString`, соответствует типу аргумента `unapply` (который в примере также имел тип `String`). Подобное случается довольно часто, но не является обязательным. Кроме того, можно было бы использовать `EMail`-экстрактор в целях сопоставления выражений для более общих типов. Например, чтобы определить, является ли произвольное значение `x` строкой адреса электронной почты, можно задействовать следующий код:

```
val x: Any = ...
x match { case EMail(user, domain) => ... }
```

С этим кодом сопоставление с образцом сначала проверяет, соответствует ли заданное значение `x` типу `String`, то есть типу параметра метода `unapply` принадлежащего `EMail`. Если да, то значение приводится к типу `String` и сопоставление с образцом продолжается в обычном режиме. Если не соответствует, то сопоставление тут же дает сбой.

Метод `apply` объекта `Email` называется *инъекцией*, поскольку этот метод получает некие аргументы и выдает элемент заданного множества (в нашем случае множества строк, являющихся адресами электронной почты). А работа метода `unapply` называется *экстракцией*, поскольку данный метод получает элемент из того же самого множества и извлекает некоторые его части (в нашем случае это подстроки `user` и `domain`). Инъекции и экстракции часто группируют вместе в одном объекте, поскольку тогда можно будет воспользоваться именем объекта как для конструктора, так и для паттерна. Таким образом имитируется соглашение, действующее в отношении сопоставления с образцом для `case`-классов. Но, помимо этого, в объекте можно определить экстракцию без соответствующей инъекции. Сам объект называется *экстрактором* независимо от того, содержит метод `apply` или нет.

Если метод инъекции включен, то должен быть дуален методу экстракции. Например, следующий вызов:

```
EMail.unapply(EMail.apply(user, domain))
```

должен возвращать:

```
Some(user, domain)
```

то есть точно такую же последовательность аргументов, заключенных в `Some`. Теперь пойдем в обратном направлении. Это значит, что, как показано в следующем коде, сначала мы запустим `unapply`, а затем `apply`:

```
EMail.unapply(obj) match {
  case Some(u, d) => EMail.apply(u, d)
}
```

В данном коде применение `match` в отношении `obj` пройдет успешно, и мы вправе ожидать получения обратно от `apply` того же самого объекта. Эти два условия дуальности для `apply` и `unapply` являются хорошими принципами конструирования. Они не навязываются языком Scala, но их рекомендуется придерживаться при создании собственных экстракторов.

26.3. Паттерны без переменных или с одной переменной

Метод `unapply` из предыдущего примера в случае удачного завершения возвращает пару элементов-значений. Его легко сделать более универсальным для паттернов из более чем двух переменных. Чтобы привязать `N` переменных, метод `unapply` будет возвращать `N`-элементный кортеж, заключенный в оболочку `Some`.

Но ситуация, при которой паттерн привязывает только одну переменную, рассматривается по-другому. В Scala нет кортежей из одного элемента. Чтобы возвращался лишь один элемент паттерна, метод `unapply` просто заключает в оболочку `Some` сам элемент. Например, объект-экстрактор, показанный в листинге 26.2, определяет `apply` и `unapply` для строк, состоящих из некой подстроки, которая появляется в строковой записи два раза подряд.

Листинг 26.2. Объект извлечения дважды встречающейся строки

```
object Twice {
  def apply(s: String): String = s + s
  def unapply(s: String): Option[String] = {
    val length = s.length / 2
    val half = s.substring(0, length)
    if (half == s.substring(length)) Some(half) else None
  }
}
```

Вдобавок существует вероятность того, что паттерн экстрактора вообще не привязывает никаких переменных. В таком случае соответствующий метод `unapply` возвращает булево значение `true` в случае успеха и `false` — при сбое. Например, объект-экстрактор, показанный в листинге 26.3, определяет строки, состоящие только из символов в верхнем регистре.

Листинг 26.3. Объект-экстрактор UpperCase

```
object UpperCase {
  def unapply(s: String): Boolean = s.toUpperCase == s
}
```

На этот раз в экстракторе определяется только метод `unapply`, а метод `apply` в нем отсутствует. Определять `apply` смысла нет, поскольку здесь нечего составлять.

Следующая функция `userTwiceUpper` в своем коде сопоставления с образцом применяет вместе все ранее определенные экстракторы:

```
def userTwiceUpper(s: String) = s match {
  case EMail(Twice(x @ UpperCase()), domain) =>
    "match: " + x + " in domain " + domain
  case _ =>
    "no match"
}
```

Первый паттерн этой функции соответствует строкам, являющимся адресами электронной почты, пользовательская часть которых состоит из двух одинаковых строк с символами в верхнем регистре, например:

```
scala> userTwiceUpper("DIDI@hotmail.com")
res0: String = match: DI in domain hotmail.com

scala> userTwiceUpper("DIDO@hotmail.com")
res1: String = no match

scala> userTwiceUpper("didi@hotmail.com")
res2: String = no match
```

Обратите внимание: `UpperCase` в функции `userTwiceUpper` получает пустой список параметров. Этот список нельзя отбросить, поскольку тогда будет проверяться равенство объекту `UpperCase`! Стоит также отметить, что, даже если сам метод `UpperCase()` не привязывает никаких переменных, связать переменную со всем соответствующим ей паттерном все же можно. Для этого используется стандартная схема привязки переменных, рассмотренная в разделе 15.2: форма `x @ UpperCase()` связывает переменную `x` с паттерном, соответствующим `UpperCase()`. Например, в показанном ранее первом вызове `userTwiceUpper` переменная была привязана к `"DI"`, поскольку это было значение, с которым сопоставлялся паттерн `UpperCase()`.

26.4. Экстракторы переменного количества аргументов

Все предыдущие методы экстракции для адресов электронной почты возвращали фиксированное количество элементов значений. Иногда подобный подход недостаточно гибок. Например, может понадобиться найти соответствие строке, представляющей доменное имя, чтобы каждая часть домена сохранялась в другом подпаттерне. Для этого потребуется выразить паттерны так:

```
dom match {
  case Domain("org", "acm") => println("acm.org")
  case Domain("com", "sun", "java") => println("java.sun.com")
  case Domain("net", _ * ) => println("a .net domain")
}
```

В этом примере все устроено так, что домены раскрываются в обратном порядке — от домена верхнего уровня к поддоменам. Сделано это с целью извлечь больше пользы от последовательно применяемых паттернов. В разделе 5.2 мы показали, что подстановочный паттерн сопоставления с последовательностью, `_*`, в конце списка аргументов соответствует любым оставшимся в ней элементам. От этого свойства больше пользы, если домен верхнего уровня стоит первым, поскольку тогда можно задействовать подстановочный паттерн сопоставления с последовательностью, чтобы искать соответствие поддоменам произвольной глубины вложения.

Остается открытым вопрос о том, каким образом экстрактор может поддерживать *сопоставление с переменным количеством аргументов*, как показано в предыдущем примере, при условии, что у паттернов может быть произвольное количество подпаттернов. Встречавшиеся до сих пор методы `unapply` здесь не подойдут, поскольку каждый из них в случае успешного завершения возвращает фиксированное количество подэлементов. Чтобы справиться с подобными обстоятельствами, Scala позволяет определять другой метод экстракции, специально предназначенный для сопоставления с переменным количеством аргументов. Этот метод называется `unapplySeq`. Разобраться в его конструкции можно, рассмотрев экстрактор `Domain`, показанный в листинге 26.4.

Листинг 26.4. Объект-экстрактор Domain-строк

```
object Domain {

  // Метод инъекции (необязательный)
  def apply(parts: String * ): String =
    parts.reverse.mkString(".")

  // Метод экстракции (обязательный)
  def unapplySeq(whole: String): Option[Seq[String]] =
    Some(whole.split("\\.").reverse.toSeq)
}
```

В объекте `Domain` определяется метод `unapplySeq`, который сначала разбивает строку на части, отделенные друг от друга точками. Это действие выполняет Java-метод `split`, применяемый к строкам, который получает в качестве своего аргумента регулярное выражение. Результатом выполнения метода `split` является массив из подстрок, а метода `unapplySeq` — массив со всеми элементами, следующими в обратном порядке, заключенный в оболочку `Some`.

Результирующий тип `unapplySeq` должен соответствовать `Option[Seq[T]]`, где тип элемента `T` может быть произвольным. Как было показано в разделе 17.1, `Seq` — весьма важный класс в иерархии коллекций Scala. Это общий суперкласс для нескольких классов, дающих описание различных видов коллекций: `List`, `Array`, `WrappedString` и некоторых других коллекций.

Для симметрии в объекте `Domain` также имеется метод `apply`, который выстраивает строку доменного имени из переменного числа параметров частей домена, начиная с имени домена верхнего уровня. Как всегда, наличие метода `apply` необязательно.

Экстрактор `Domain` можно использовать для получения более детализированной информации о строках адресов электронной почты. Например, для поиска адресов электронной почты с именем `"tom"` в каком-либо домене `.com`, можно задействовать следующую функцию:

```
def isTomInDotCom(s: String): Boolean = s match {
  case EMail("tom", Domain("com", _ * )) => true
  case _ => false
}
```

Этот код даст вполне ожидаемые результаты:

```
scala> isTomInDotCom("tom@sun.com")
res3: Boolean = true

scala> isTomInDotCom("peter@sun.com")
res4: Boolean = false

scala> isTomInDotCom("tom@acm.org")
res5: Boolean = false
```

Можно также наряду с переменной частью получить в качестве возвращаемого значения `unapplySeq` ряд фиксированных элементов. Это выражается в возвращении всех элементов в кортеже, где переменная часть, как обычно, будет последней. В примере, показанном в листинге 26.5, демонстрируется новый экстрактор для адресов электронной почты, где доменная часть уже разложена в последовательность.

Листинг 26.5. Объект-экстрактор ExpandedEMail

```
object ExpandedEMail {
  def unapplySeq(email: String)
      : Option[(String, Seq[String])] = {
    val parts = email split "@"
    if (parts.length == 2)
      Some(parts(0), parts(1).split("\\.").reverse.toSeq)
    else
      None
  }
}
```

Имеющийся в объекте `ExpandedEMail` метод `unapplySeq` возвращает представленное парой (`Tuple2`) значение `Option`. Первый элемент данной пары является пользовательской частью. А второй элемент — это последовательность имен, представляющих домен. Выполнять сопоставление можно как обычно:

```
scala> val s = "tom@support.epfl.ch"
s: String = tom@support.epfl.ch

scala> val ExpandedEMail(name, topdom, subdoms @ _ * ) = s
name: String = tom
topdom: String = ch
subdoms: Seq[String] = WrappedArray(epfl, support)
```

26.5. Экстракторы и паттерны последовательностей

В разделе 15.2 мы показали, что доступ к элементам списка или массива можно получить, используя следующие паттерны последовательностей:

```
List()
List(x, y, _ * )
Array(x, 0, 0, _)
```

Фактически все эти паттерны последовательностей реализованы с использованием экстракторов из стандартной библиотеки Scala. Например, применение паттернов вида `List(...)` стало возможным благодаря тому, что объект-компаньон `scala.List` является экстрактором, определяющим метод `unapplySeq`. Соответствующие определения показаны в листинге 26.6.

Листинг 26.6. Экстрактор, определяющий метод unapplySeq

```
package scala
object List {
  def apply[T](elems: T * ) = elems.toList
  def unapplySeq[T](x: List[T]): Option[Seq[T]] = Some(x)
  ...
}
```

В объекте `List` содержится метод `apply`, который получает переменное число аргументов. Это позволяет воспользоваться следующими выражениями:

```
List()
List(1, 2, 3)
```

В нем также содержится метод `unapplySeq`, возвращающий все элементы списка в виде последовательности. Именно он и поддерживает паттерны вида `List(...)`. Очень похожее определение есть в объекте `scala.Array`. Им поддерживаются аналогичные инъекции и экстракции в отношении массивов.

26.6. Сравнение экстракторов и case-классов

При всей немалой пользе, получаемой от `case`-классов, у них есть один существенный недостаток: с их помощью выявляется конкретное представление данных. Это значит, что имя класса в паттерне конструктора соответствует конкретному представлению типа объекта селектора. Если сопоставление вида:

```
case C(...)
```

будет выполнено успешно, то станет понятно, что выражение селектора является экземпляром класса C.

Экстракторы разрывают эту связь между представлением данных и паттернами. А в примерах этого раздела мы показали, что они допускают применение паттернов,

которые не имеют ничего общего с типом данных выбираемого объекта. Это свойство называется *независимостью представления*. В крупных открытых системах она играет весьма важную роль, поскольку позволяет изменять тип реализации, используемый в наборе компонентов, не затрагивая код тех, кто пользуется этими компонентами.

Если ваш компонент определил и экспортировал набор `case`-классов, то станет камнем преткновения, так как код клиента уже может содержать сопоставление с образцом в отношении этих же `case`-классов. Переименование некоторых `case`-классов или изменение иерархии классов повлияет на код клиента. Экстракторы не сталкиваются с подобной проблемой, поскольку представляют собой уровень косвенности между представлением данных и тем, как их видят клиенты. У вас сохраняется возможность изменять конкретное представление типа при условии, что наряду с этим будут обновлены все ваши экстракторы.

Независимость представления дает экстракторам существенные преимущества над `case`-классами. Но и у `case`-классов есть некоторые достоинства в сравнении с экстракторами. Во-первых, их намного проще настраивать и определять, и для них требуется меньше кода. Во-вторых, они зачастую приводят к более эффективной организации сопоставления с образцом, чем экстракторы, поскольку компилятор Scala может оптимизировать паттерны, применяемые к `case`-классам, намного лучше паттернов, применяемых к экстракторам. Дело в том, что механизмы `case`-классов фиксированы, а методы `unapply` или `unapplySeq` в экстракторе способны практически на все. В-третьих, если ваши `case`-классы являются наследниками `sealed` базовых классов, то компилятор Scala проверит ваше сопоставление с образцом на полноту и станет возражать, если какие-либо комбинации возможных значений не были охвачены паттерном. Для экстракторов любые проверки на полноту недоступны.

Какому же из двух методов сопоставления с образцом следует отдать предпочтение? Все зависит от конкретных обстоятельств. Если создается код для приватного приложения, то предпочтение обычно отдается `case`-классам, поскольку они обеспечивают преимущества в краткости, скорости и возможности статической проверки кода. Если позже будет принято решение об изменении иерархии классов, то приложение нужно будет реструктурировать, но обычно это не вызывает серьезных проблем. В то же время, при необходимости распространить тип среди неизвестных вам клиентов предпочтительным может стать применение экстракторов ввиду того, что они обеспечивают независимость представления.

К счастью, немедленно принимать решение не обязательно. Всегда можно начать с применения `case`-классов, а затем при необходимости заменить соответствующий код экстракторами. Паттерны для работы с `case`-классами и паттерны для работы с экстракторами выглядят в Scala совершенно одинаково, поэтому сопоставление с образцом в коде ваших клиентов не утратит своей работоспособности.

Разумеется, бывают ситуации, когда с самого начала понятно, что структура паттернов не соответствует представлению типов ваших данных. Примером могут послужить рассмотренные в данной главе адреса электронной почты. В таком случае единственным доступным вариантом останутся экстракторы.

26.7. Регулярные выражения

Одна из полезных сфер применения экстракторов — регулярные выражения. Как и в Java, в библиотеке Scala есть регулярные выражения, но экстракторы делают работу с ними намного более привлекательной.

Формирование регулярных выражений

Синтаксис, используемый в Scala для регулярных выражений, унаследован из Java, который, в свою очередь, унаследовал большинство свойств из Perl. Предполагаем, что данный синтаксис вам уже известен. Если же это не так, то есть множество доступных руководств, начиная с документации Javadoc, с описанием класса `java.util.regex.Pattern`. Приведем несколько примеров, которых будет достаточно, чтобы освежить память.

- `ab?` — `a`, за которым, возможно, следует `b`.
- `\d+` — число, состоящее из одной или нескольких цифр, представленных `\d`.
- `[a-dA-D]\w *` — слово, начинающееся с буквы в диапазоне от `a` до `d` в верхнем или нижнем регистре, за которым идет последовательность из нуля и более словообразующих символов, обозначенных `\w` (к таким символам относятся буква, цифра или знак подчеркивания).
- `(-)?(\d+)(\.\d * )?` — число, состоящее из необязательного знака минус, за которым следуют одна или несколько цифр, после которых не обязательно стоят точка и от нуля до нескольких цифр. Число состоит из трех *групп*, то есть из знака минус, той части, что идет до десятичной точки, и дробной части, включая десятичную точку. Группы заключены в круглые скобки.

Класс регулярных выражений находится в пакете `scala.util.matching`:

```
scala> import scala.util.matching.Regex
```

Новое значение регулярного выражения создается передачей строки конструктору `Regex`, например:

```
scala> val Decimal = new Regex("(-)?(\\d+)(\\.\\d * )?")
Decimal: scala.util.matching.Regex = (-)?(\d+)(\.\d * )?
```

Обратите внимание на то, что по сравнению с регулярным выражением для десятичных чисел, заданным ранее, в показанной выше строке каждый обратный слеш появляется дважды. Дело в том, что и в Java, и в Scala одиночный обратный слеш — в строковом литерале управляющий символ, а не действующий символ регулярного выражения, показываемый в строке. Поэтому для получения в строке действующего одиночного слеша нужно вместо символа `\` использовать пару символов `\\`.

Если в регулярном выражении содержится множество обратных слешей, его написание и чтение будет затруднено. Альтернатива — имеющаяся в Scala неформатированная строка. Как мы показали в разделе 5.2, неформатированной строкой называется последовательность символов, заключенная в тройные кавычки.

Неформатированная и простая строка различаются тем, что неформатированная отображается в точности так, как была набрана. Это относится и к обратным слешам, которые не считаются в ней управляющими символами. Таким образом, можно сделать равнозначную и более разборчивую запись:

```
scala> val Decimal = new Regex("""(-)?(\d+)(\.\d * )?""")
Decimal: scala.util.matching.Regex = (-)?(\d+)(\.\d * )?
```

Из выведенной интерпретатором информации видно, что сгенерированное значение результата для `Decimal` точно такое же, что и прежде.

Другой, еще более краткий способ написания регулярного выражения в Scala выглядит следующим образом:

```
scala> val Decimal = """(-)?(\d+)(\.\d * )?""".r
Decimal: scala.util.matching.Regex = (-)?(\d+)(\.\d * )?
```

Иначе говоря, нужно просто добавить к строке `.r`, чтобы получить регулярное выражение. Эту возможность обеспечивает наличие в классе `StringOps` метода `r`, который преобразует строку в регулярное выражение. Определение этого метода показано в листинге 26.7.

Листинг 26.7. Определение метода r в классе StringOps

```
package scala.runtime
import scala.util.matching.Regex

class StringOps(self: String) ... {
  ...
  def r = new Regex(self)
}
```

Поиск регулярных выражений

Определить наличие в строке соответствия регулярному выражению можно с помощью нескольких различных операторов:

- `regex findFirstIn str` — поиск первого соответствия регулярному выражению `regex` в строке `str`, результат возвращается в значении `Option`-типа;
- `regex findAllIn str` — поиск всех соответствий регулярному выражению `regex` в строке `str`, результат возвращается в значении типа `Iterator`;
- `regex findPrefixOf str` — поиск соответствия регулярному выражению `regex` в самом начале строки `str`, результат возвращается в значении `Option`-типа.

Например, можно определить входную последовательность, показанную далее, а затем искать в ней десятичные числа:

```
scala> val Decimal = """(-)?(\d+)(\.\d * )?""".r
Decimal: scala.util.matching.Regex = (-)?(\d+)(\.\d * )?

scala> val input = "for -1.0 to 99 by 3"
input: String = for -1.0 to 99 by 3
```

```
scala> for (s <- Decimal findAllIn input)
       println(s)
-1.0
99
3

scala> Decimal findFirstIn input
res7: Option[String] = Some(-1.0)

scala> Decimal findPrefixOf input
res8: Option[String] = None
```

Извлечение с помощью регулярных выражений

А еще каждое регулярное выражение в Scala определяет экстрактор. Он используется для определения подстрок, соответствующих группам регулярных выражений. Например, строку, которая обозначает десятичное число, можно разбить на составляющие так:

```
scala> val Decimal(sign, integerpart, decimalpart) = "-1.23"
sign: String = -
integerpart: String = 1
decimalpart: String = .23
```

В этом примере паттерн `Decimal(...)` используется в `val`-определении в соответствии с описанным в разделе 15.7. Здесь с помощью регулярного выражения определяется метод `unapplySeq`. Он соответствует каждой строке, которая соответствует синтаксису регулярных выражений для десятичных чисел. Если строка соответствует, то в качестве элементов паттерна возвращаются те части, соответствующие трем группам в регулярном выражении `(-)?(\d+)(\.\d * )?`, которые затем сопоставляются с тремя переменными паттерна: `sign`, `integerpart` и `decimalpart`. При отсутствии группы для элемента устанавливается значение `null`, в чем можно убедиться, рассмотрев следующий пример:

```
scala> val Decimal(sign, integerpart, decimalpart) = "1.0"
sign: String = null
integerpart: String = 1
decimalpart: String = .0
```

Кроме того, можно смешивать экстракторы с поиском с помощью регулярных выражений в `for`-выражениях. Например, представленное ниже выражение разбивает на составляющие все десятичные числа, найденные им в строке `input`:

```
scala> for (Decimal(s, i, d) <- Decimal findAllIn input)
         println("sign: " + s + ", integer: " +
             i + ", decimal: " + d)
sign: -, integer: 1, decimal: .0
sign: null, integer: 99, decimal: null
sign: null, integer: 3, decimal: null
```

Резюме

В этой главе мы показали, как с помощью экстракторов сделать сопоставление с образцом более универсальным. Экстракторы дают возможность определять собственные разновидности паттернов, которым не нужно соответствовать типу выбираемых выражений. Благодаря этому можно более гибко подходить к выбору видов паттернов, допускаемых к использованию в сопоставлении. Фактически это похоже на оперирование разнообразными представлениями одних и тех же данных. Кроме того, вы получаете промежуточный уровень между представлением типов и методом их представления клиентам. Это позволяет выполнять сопоставление с образцом, одновременно обеспечивая независимость представления, то есть получить весьма ценное свойство, которое особенно пригодится для больших программных систем.

В вашем арсенале инструментов экстракторы — еще одно средство, которое позволяет определять весьма гибкие библиотечные абстракции. Они довольно часто используются в библиотеках Scala, например, чтобы обеспечить возможность удобного поиска с применением регулярных выражений.

27 Аннотации

Аннотации представляют собой структурированную информацию, добавляемую к исходному коду программы. Как и комментарии, они могут быть разбросаны по всей программе и прикреплены к любым переменной, методу, выражению или другому элементу программы. В отличие от комментариев у них есть структура, что упрощает процесс их обработки.

В этой главе мы рассмотрим порядок применения аннотаций в Scala: их общий синтаксис и приемы использования ряда стандартных аннотаций.

Создание новых инструментальных средств для обработки аннотаций не относится к теме данной книги и не рассматривается в ней. В главе 31 мы покажем одну технологию, но она далеко не единственная. Вместо этого основное внимание в данной главе уделяется порядку использования аннотаций, поскольку применять их приходится гораздо чаще, чем определять новые обработчики аннотаций.

27.1. Зачем нужны аннотации

Помимо компиляции и выполнения, с программой можно делать и многое другое. Примерами могут послужить следующие действия.

1. Автоматическое создание документации, как это делается с помощью Scaladoc.
2. Красивый вывод кода в соответствии с предпочитаемым вами стилем.
3. Проверка кода на наличие самых распространенных ошибок, например таких, когда файл открывается, но при некоторых вариантах хода выполнения программы никогда не закрывается.
4. Экспериментальная проверка типов, например, для управления побочными эффектами или обеспечения свойств принадлежности.

Такие инструменты называются средствами *метапрограммирования*, поскольку являются программами, которые принимают на вход другие программы. Аннотации поддерживают эти средства, позволяя программистам расставлять директивы для определенного средства по всему исходному коду. Подобные директивы дают

средствам возможность работать более эффективно, чем при отсутствии пользовательского ввода. Например, аннотации могут улучшить ранее перечисленный инструментарий следующим образом.

1. Генератор документации можно настроить на документирование конкретных методов в качестве устаревших.
2. Красивый вывод кода можно настроить на пропуск тех частей программы, которые уже были тщательно отформатированы вручную.
3. Средство проверки неприватных файлов можно настроить на игнорирование конкретного файла, закрытие которого уже было проверено программистом.
4. Средство проверки побочных эффектов можно настроить на проверку отсутствия таковых у указанного метода.

Для всех этих случаев теоретически язык программирования должен содержать возможность обеспечить способы вставки дополнительной информации. Фактически большинство из этого непосредственно поддерживается в тех или иных языках. Но для одного языка непосредственной поддержки такого арсенала инструментальных средств будет многовато. Более того, вся эта информация игнорируется компилятором, от которого требуется лишь сделать код работоспособным.

В подобных случаях философия Scala состоит во включении разумного минимума средств, независимо друг от друга поддерживаемых в основном языке, чтобы можно было воспользоваться широким разнообразием средств метапрограммирования. В данном случае в качестве минимальной поддержки выступает система аннотаций. Компилятору важно только наличие аннотации, но значение каждой конкретной аннотации ему безразлично. Каждое средство метапрограммирования может затем определять и использовать собственные аннотации.

27.2. Синтаксис аннотаций

Типичное использование аннотации выглядит так:

```
@deprecated def bigMistake() = //...
```

Аннотацией здесь является `@deprecated`, и она применяется ко всему методу `bigMistake` (который не показан — стыдно). В данном случае метод помечен автором `bigMistake` как нечто нежелательное для использования. Возможно, из будущей версии кода `bigMistake` будет полностью изъят.

В предыдущем примере метод аннотирован как `@deprecated`. Аннотации можно применять и в других местах. Допускается использовать их в любых видах объявлений или определений, включая объявления `val`, `var`, `def`, `class`, `object`, `trait` и `type`. Аннотация применяется ко всему объявлению или определению, которое следует за ней:

```
@deprecated class QuickAndDirty {
  //...
}
```

Аннотации могут применяться и к выражениям, как в случае с аннотацией `@unchecked` для сопоставления с образцом (см. главу 15). Для этого после выражения ставится двоеточие (`:`), а затем записывается аннотация. Синтаксически это похоже на использование аннотации в качестве типа:

```
(e: @unchecked) match {
  // Неисчерпывающие варианты...
}
```

И наконец, аннотации могут помещаться на типах. Аннотированные типы мы рассмотрим в данной главе чуть позже.

Все показанные до сих пор аннотации выглядели просто как знак «собачка», после которого указывался класс аннотации. Такие простые аннотации употребляются довольно часто, но у аннотаций есть и более полная общая форма:

```
@annot(exp1, exp2, ...)
```

Часть `annot` указывает на класс аннотации. Его должны включать все аннотации. Части *`exp`* являются аргументами аннотации. Для аннотаций наподобие `@deprecated`, которым не нужны никакие аргументы, скобки обычно опускаются, но при желании можно пользоваться записью вида `@deprecated()`. Для аннотаций, содержащих аргументы, их следует заключать в круглые скобки, например `@serial(1234)`.

Точная форма аргументов, допустимых для передачи аннотации, зависит от конкретного класса аннотации. Большинство обработчиков аннотаций позволяют предоставлять только непосредственные константы наподобие `123` или `"hello"`. Но сам компилятор поддерживает произвольные выражения при условии, что они пройдут проверку типов.

Некоторые классы аннотаций могут этим воспользоваться, например, чтобы позволить вам ссылаться на другие переменные, находящиеся в области видимости:

```
@cool val normal = "Hello"
@coolerThan(normal) val fonzy = "Heeyyy"
```

Внутренне аннотация в Scala представлена просто как вызов конструктора класса аннотации: замените знак `@` на ключевое слово `new` — и получите допустимый экземпляр выражения создания. Это означает естественную поддержку именованных аргументов и аргументов по умолчанию для аннотаций, поскольку в Scala уже есть именованные аргументы и аргументы по умолчанию для вызовов методов и конструкторов. Еще одна сложность касается аннотаций, которые концептуально принимают в качестве аргументов другие аннотации, в чем есть потребность у некоторых фреймворков. Записывать аннотацию непосредственно как аргумент аннотации нельзя ввиду того, что аннотации не являются допустимыми выражениями. Как показано в следующем диалоге с интерпретатором, в таких случаях следует вместо знака `@` использовать ключевое слово `new`:

```
scala> import annotation._
import annotation._
```

```
scala> class strategy(arg: Annotation) extends Annotation
defined class strategy

scala> class delayed extends Annotation
defined class delayed

scala> @strategy(@delayed) def f() = {}
                 ^
       error: illegal start of simple expression

scala> @strategy(new delayed) def f() = {}
f: ()Unit
```

27.3. Стандартные аннотации

В Scala включен ряд стандартных аннотаций. Они используются достаточно широко, чтобы заслужить включение в спецификацию языка, однако недостаточно фундаментальны, чтобы заслужить собственный синтаксис. Со временем должен появиться новый набор аннотаций, точно так же добавляемых к стандарту.

Устаревание

Иногда со временем возникает желание избавиться от ранее созданного класса или метода. Но как только он стал доступным, метод может быть вызван кодом, созданным другими людьми. Следовательно, просто удалить метод невозможно, поскольку это приведет к прекращению компиляции кода, созданного сторонними разработчиками.

Устаревание позволяет вам аккуратно удалить ошибочно созданный метод или класс. Подобный метод или класс помечается как устаревший (deprecated), после чего все вызывающие его разработчики станут получать предупреждение об устаревании. Им лучше прислушаться к данному предупреждению и обновить свой код! Смысл заключается в том, что по истечении довольно продолжительного времени появляется уверенность, что все значимые клиенты перестали обращаться к устаревшему методу или классу и поэтому его можно совершенно свободно удалить.

Метод помечается как устаревший путем написания перед ним аннотации `@deprecated`, например:

```
@deprecated def bigMistake() = //...
```

Такая аннотация заставит компилятор Scala в случае обращения кода к методу выдать предупреждение об устаревании.

Если в качестве аргумента аннотации `@deprecated` предоставить строку, то она будет выдана вместе с предупреждением. Заключенное в эту строку сообщение

объяснит разработчикам, что именно им следует использовать взамен устаревшего метода:

```
@deprecated("use newShinyMethod() instead")
def bigMistake() = //...
```

Теперь все обращающиеся к данному методу разработчики получат такое сообщение:

```
$ scalac -deprecation Deprecation2.scala
Deprecation2.scala:33: warning: method bigMistake in object
Deprecation2 is deprecated: use newShinyMethod() instead
    bigMistake()
    ^
one warning found
```

Непостоянные поля

Программирование многопоточных приложений плохо сочетается с совместно используемым изменяемым состоянием. Поэтому механизм `Future` уделяет основное внимание поддержке многопоточного программирования на Scala и сведению к минимуму совместно используемого изменяемого состояния. Подробности рассматриваются в главе 32.

Тем не менее иногда программистам в их многопоточных программах нужно применить изменяемое состояние. В подобных случаях им поможет аннотация `@volatile`. Она проинформирует компилятор, что рассматриваемая переменная будет использоваться сразу несколькими потоками. Подобные переменные реализуются таким образом, чтобы чтение и запись в них выполнялись медленнее, а обращение из нескольких потоков велось более предсказуемо.

Ключевое слово `@volatile` на разных платформах дает разные гарантии. Однако на платформе Java получается такое же поведение, как и при написании поля в коде Java и пометке его Java-модификатором `volatile`.

Двоичная сериализация

Фреймворки двоичной *сериализации* включены во многие языки. Фреймворк сериализации помогает превращать объекты в потоки байтов и *наоборот*. Это пригодится для сохранения объектов на диске или отправки их по сети. Добиться тех же результатов можно с помощью XML (см. главу 28), но при этом проявляются различные недостатки, касающиеся скорости, использования пространства памяти, гибкости и переносимости.

У Scala нет собственного фреймворка сериализации. Поэтому придется пользоваться фреймворком базовой платформы. А язык Scala предоставляет две аннотации, применяемые для различных фреймворков. Кроме того, компилятор Scala для платформы Java интерпретирует эти аннотации тем же способом, что и Java (см. главу 31).

Для начала, большинство классов пригодно к сериализации, но бывают исключения. К примеру, не поддается сериализации обработка сокета или GUI-окна. По умолчанию класс не считается поддающимся сериализации. Если нужно, чтобы он проходил сериализацию, то к нему следует добавить трейт `scala.Serializable`, который является псевдонимом `java.io.Serializable` из JVM.

Аннотация `SerialVersionUID` помогает в работе с сериализуемыми классами, которые со временем претерпевают какие-либо изменения. К текущей версии класса можно прикрепить серийный номер, добавив аннотацию вида `@SerialVersionUID(1234)`, где вместо `1234` следует поставить выбранный вами серийный номер. Фреймворк должен сохранить его в создаваемом потоке байтов. Если впоследствии такой же поток байтов будет загружен заново и произойдет попытка превратить его в объект, то фреймворк сможет проверить, имеет ли текущая версия класса тот же серийный номер, что и версия, переданная в потоке байтов. Если нужно внести в класс несовместимые с прежней сериализацией изменения, то можно указать другой номер версии. После этого фреймворк автоматически откажется загружать старые экземпляры класса.

Scala также предоставляет аннотацию `@transient` для тех полей, которые вообще не должны подвергаться сериализации. Можно пометить поле аннотацией `@transient`, тогда фреймворк не станет его сохранять, даже если окружающий поле объект был сериализован. При загрузке объекта для такого поля, тип которого аннотирован как `@transient`, будет восстановлено исходное значение.

Автоматические методы get и set

Обычно код на Scala не нуждается в явном определении для полей методов `get` и `set`, поскольку Scala смешивает синтаксис для обращения к полям и для вызова методов. Но некоторые зависящие от платформы фреймворки ожидают наличия методов `get` и `set`. Чтобы решить этот вопрос, в Scala предоставляется аннотация `@scala.reflect.BeanProperty`. При добавлении этой аннотации к полю компилятор автоматически сгенерирует для него методы `get` и `set`. Если аннотируется поле по имени `crazy`, то `get`-метод будет назван `getCrazy`, а `set`-метод — `setCrazy`.

Сгенерированные методы `get` и `set` станут доступными только после завершения прохода компиляции. То есть вызывать эти `get`- и `set`-методы из кода, скомпилированного одновременно с аннотированными полями, невозможно. На практике это вряд ли станет проблемой, поскольку в коде на Scala к этим полям можно обращаться напрямую. Данное свойство предназначено для поддержки тех фреймворков, в которых ожидается наличие обычных методов `get` и `set`, и, как правило, сам фреймворк и код, использующий его, в одно и то же время не компилируются.

Хвостовая рекурсия

Аннотация `@tailrec` обычно добавляется к методу, который должен быть хвостовой рекурсией, например, из-за ожиданий, что иначе рекурсия будет слишком глубокой. Чтобы заставить компилятор Scala выполнить в отношении метода оптимизацию

с хвостовой рекурсией, рассмотренную в разделе 8.9, перед определением метода можно добавить аннотацию `@tailrec`. Если оптимизацию выполнить невозможно, то будет получено предупреждение с объяснением причин.

Unchecked

Аннотация `@unchecked` интерпретируется компилятором в ходе сопоставления с образцом. Благодаря ей компилятор перестает заботиться об отсутствии в выражении соответствия ряда вариантов. Подробности мы рассмотрели в разделе 15.5.

Native-методы

Аннотация `@native` информирует компилятор, что реализация метода предоставляется средой выполнения, а не кодом Scala. Компилятор установит на выходе соответствующие флаги, и разработчику достаточно будет обеспечить реализацию, используя такие механизмы, как JNI (Java Native Interface).

При использовании аннотации `@native` должно быть предоставлено тело метода, которое не будет отражено на выходе. Например, объявить, что метод `beginCountdown` будет предоставлен средой выполнения, можно следующим образом:

```
@native
def beginCountdown() = {}
```

Резюме

В этой главе мы рассмотрели платформонезависимые аспекты аннотаций, сведения о которых наиболее востребованы. В первую очередь мы обсудили синтаксис аннотаций, поскольку использовать их приходится гораздо чаще, чем создавать новые. Далее мы рассмотрели вопросы применения ряда аннотаций, поддерживаемых стандартным компилятором Scala, включая `@deprecated`, `@volatile`, `@serializable`, `@BeanProperty`, `@tailrec` и `@unchecked`.

Дополнительную информацию об аннотациях, специфических для Java, мы дадим в главе 31. Речь в ней идет об аннотациях, нацеленных только на применение в Java, мы раскрываем дополнительное значение аннотаций применительно к Java, рассматриваем вопросы взаимодействия с аннотациями, основанными на Java, и приемы использования Java-механизмов для определения и обработки аннотаций в Scala.

28 Работа с XML

В этой главе мы рассмотрим реализованную в Scala поддержку XML. После общего обзора слабоструктурированных данных в ней показаны основные функциональные возможности Scala по работе с XML: как с помощью литералов XML создаются узлы, как XML сохраняется в файлах и загружается из них и как выполняется разбор XML-узлов с использованием методов запросов и сопоставления с образцом. Эта глава — всего лишь краткое введение в возможности работы с XML, но изложенного в ней материала вполне достаточно для того, чтобы начать применять данную технологию на практике.

28.1. Слабоструктурированные данные

XML — форма *слабоструктурированных данных*. Они более структурированы, чем обычные строки, поскольку организуют данные в виде дерева. Но обычный XML менее структурирован, чем объекты языков программирования, так как между тегами в нем допускается использование текста в свободной форме, а система типов отсутствует[1].

Слабоструктурированные данные очень хорошо подходят для тех случаев, когда нужно сериализовать (то есть превратить в последовательность битов) данные программы в целях сохранения их в файле или передачи по сети. Вместо преобразования структурированных данных вплоть до байтов их преобразуют в слабоструктурированные данные и обратно. Преобразования между слабоструктурированными и двоичными данными выполняются с помощью уже существующих библиотечных процедур, что позволяет сэкономить время для решения более важных задач.

Существует множество форм слабоструктурированных данных, но наиболее широкое распространение в Интернете получил язык XML. Средства XML есть в большинстве операционных систем, а доступные библиотеки XML имеют многие

[1] Для XML существуют системы типов, такие как XML Schema, но в данной книге они не рассматриваются.

языки программирования. Его популярность растет сама. Чем больше средств и библиотек разрабатывается в ответ на рост популярности XML, тем выше шансы, что разработчики программных систем выберут XML как часть своих форматов. Если вы пишете программы, общающиеся через Интернет, то рано или поздно столкнетесь с какими-либо сервисами, которые используют XML.

Принимая во внимание все эти обстоятельства, для работы с XML в Scala включили специальную поддержку. В данной главе мы рассмотрим реализованную в Scala поддержку создания XML-кода, его обработку с помощью обычных методов и сопоставления с образцом. Вдобавок обсудим ряд общих идиом использования XML в Scala.

28.2. Краткий обзор XML

Структура XML выстраивается из двух базовых элементов — текста и тегов[1]. Текст, как всегда, представлен последовательностью символов. Теги, записываемые как `<pod>`, состоят из символа «меньше», буквенно-цифровой метки и символа «больше». Теги могут быть *открывающими* и *закрывающими*. Закрывающий тег выглядит практически так же, как и открывающий, за исключением использования слеша перед меткой тега, например `</pod>`.

Открывающий и закрывающий теги должны соответствовать друг другу, что напоминает применение круглых скобок. За любым открывающим тегом должен в конечном итоге следовать закрывающий с той же самой меткой. Поэтому использование вот такого кода недопустимо:

```
// Недопустимый код XML
One <pod>, two <pod>, three <pod> zoo
```

Далее, содержимое, которое находится внутри двух соответствующих друг другу тегов, должно быть допустимым кодом XML. В коде не может быть двух пар соответствующих друг другу пересекающихся тегов:

```
// Этот код также недопустим
<pod>Three <peas> in the </pod></peas>
```

Но вполне допустима следующая запись:

```
<pod>Three <peas></peas> in the </pod>
```

Из-за принятого правила соответствия тегов структурирование XML заключается в соблюдении *вложенности* элементов. Элемент формируется парой соответствующих друг другу открывающего и закрывающего тегов, и элементы могут быть вложены друг в друга. В примере выше весь код `<pod>Three <peas></peas> in the </pod>` в целом является элементом, а `<peas></peas>` — вложенным в него элементом.

[1] В действительности все немного сложнее, но для полноценной работы с XML вполне достаточно и этих сведений.

Это основа основ. Следует знать и о двух других особенностях. Во-первых, существует сокращенная система записи для случая, когда сразу же за открывающим тегом стоит соответствующий ему закрывающий. Нужно просто записать один тег со слешем, помещаемым сразу же после метки тега. Такой тег содержит *пустой элемент*. Пустой элемент позволяет записать предыдущий пример следующим образом:

```
<pod>Three <peas/> in the </pod>
```

Во-вторых, к открывающему тегу могут быть прикреплены *атрибуты*. Атрибутом называется пара «имя — значение», со знаком равенства посередине. Имя атрибута является простым неструктурированным текстом, а значение заключается в двойные (`""`) или одинарные (`''`) кавычки. Выглядят атрибуты так:

```
<pod peas="3" strings="true"/>
```

28.3. Литералы XML

Scala позволяет набирать код XML в виде литерала везде, где допускается использование выражения. Нужно просто набрать открывающий тег, а затем продолжить набор XML-содержимого. Компилятор перейдет в режим XML-ввода и станет считывать содержимое XML до тех пор, пока ему не попадется закрывающий тег, соответствующий открывающему тегу, с которого начинался литерал XML:

```
scala> <a>
         Это код XML.
         Вот тег: <atag/>
       </a>
res0: scala.xml.Elem =
<a>
  Это код XML.
  Вот тег: <atag/>
</a>
```

Результат данного выражения будет относиться к типу `Elem`, что означает следующее: это XML-элемент с меткой «a» и дочерним элементом («Это код XML...» и т. д.). К другим важным XML-классам относятся:

- класс `Node` — абстрактный суперкласс всех классов XML-узлов;
- класс `Text` — узел, содержащий только текст. Например, часть `stuff` литерала `<a>stuff</a>` относится к классу `Text`;
- класс `NodeSeq` содержит последовательность узлов. Многие методы в библиотеке `XML` обрабатывают `NodeSeq` в тех местах, в которых от них можно ожидать обработки отдельных `Node`. Но такие методы можно использовать и с отдельно взятыми узлами, поскольку `Node` расширяет `NodeSeq`. Данное обстоятельство может показаться немного странным, но в отношении XML все это срабатывает. Отдельно взятый `Node` можно рассматривать как одноэлементный `NodeSeq`.

Вы не ограничены в написании точного XML-кода посимвольно. Используя в качестве управляющих символов фигурные скобки, в середине XML-литерала можно выполнять вычисления кода Scala. Вот простой пример:

```
scala> <a> {"hello" + ", world"} </a>
res1: scala.xml.Elem = <a> hello, world </a>
```

Внутри управляющих символов в виде фигурных скобок может помещаться произвольное Scala-содержимое, включая дополнительные XML-литералы. Таким образом, по мере увеличения количества вложений в вашем коде может перемежаться XML-код и обычный код Scala, например:

```
scala> val yearMade = 1955
yearMade: Int = 1955

scala> <a> { if (yearMade < 2000) <old>{yearMade}</old>
             else xml.NodeSeq.Empty }
       </a>
res2: scala.xml.Elem =
<a> <old>1955</old>
 </a>
```

Если код внутри фигурных скобок вычисляется либо в XML-узел, либо в последовательность XML-узлов, то такие узлы вставляются непосредственно в получившемся в результате вычисления виде. Если в показанном ранее примере `yearMade` меньше `2000`, то значение `yearMade` заключается в теги `<old>`, и все это добавляется к элементу `<a>`. В противном случае не добавляется ничего. Относительно данного примера следует заметить, что «ничего» в виде XML-узла обозначается `xml.NodeSeq.Empty`.

Выражение внутри управляющих символов в виде фигурных скобок не должно обязательно вычисляться в XML-узел. Оно может вычисляться в любое значение Scala. В таком случае результат преобразуется в строку и вставляется в качестве текстового узла:

```
scala> <a> {3 + 4} </a>
res3: scala.xml.Elem = <a> 7 </a>
```

При выводе узла на стандартное устройство любые встречающиеся в тексте знаки (`<`, `>` и `&`) будут превращены в управляющие последовательности символов:

```
scala> <a> {"</a>потенциальная брешь в системе безопасности<a>"} </a>
res4: scala.xml.Elem = <a> &lt;/a&gt;потенциальная брешь
в системе безопасности&lt;a&gt; </a>
```

В отличие от этого, если создавать XML с помощью низкоуровневых строковых операций, то можно угодить в следующую ловушку:

```
scala> "<a>" + "</a>потенциальная брешь в системе безопасности<a>" + "</a>"
res5: String = <a></a>потенциальная брешь в системе безопасности<a></a>
```

Здесь заданная пользователем строка была включена в XML-теги в неизменном виде — в данном случае вместе с и <a>. Этот прием может преподнести программисту ряд неприятных сюрпризов, поскольку позволяет пользователю влиять на получающееся в итоге дерево XML за пределами пространства, предоставляемого пользователю внутри элемента <a>. Если конструировать XML только с применением XML-литералов, отказавшись от добавления строк, то весь этот класс проблем можно предотвратить.

28.4. Сериализация

Разобравшись в достаточной степени в поддержке XML в Scala, можно приступать к написанию первой части сериализатора, которая превращает внутренние структуры данных в XML. Для этого понадобятся лишь XML-литералы и их управляющие символы — скобки.

Предположим, к примеру, что вы реализуете базу данных для систематизации своей обширной коллекции старинных термометров с логотипами Coca-Cola. Чтобы хранить записи каталога, можно создать следующий внутренний класс:

```
abstract class CCTherm {
  val description: String
  val yearMade: Int
  val dateObtained: String
  val bookPrice: Int      // в центах США
  val purchasePrice: Int  // в центах США
  val condition: Int      // от 1 до 10

  override def toString = description
}
```

Это обычный нагруженный данными класс, содержащий различные сведения о том, к примеру, где термометр был изготовлен, когда приобретен и по какой цене.

Чтобы превратить экземпляры этого класса в XML, следует просто добавить метод `toXML`, который использует XML-литералы и управляющие скобки:

```
abstract class CCTherm {
  ...
  def toXML =
    <cctherm>
      <description>{description}</description>
      <yearMade>{yearMade}</yearMade>
      <dateObtained>{dateObtained}</dateObtained>
      <bookPrice>{bookPrice}</bookPrice>
      <purchasePrice>{purchasePrice}</purchasePrice>
      <condition>{condition}</condition>
    </cctherm>
}
```

А вот как выглядит работа этого метода:

```
scala> val therm = new CCTherm {
         val description = "hot dog #5"
         val yearMade = 1952
         val dateObtained = "March 14, 2006"
         val bookPrice = 2199
         val purchasePrice = 500
         val condition = 9
       }
therm: CCTherm = hot dog #5

scala> therm.toXML
res6: scala.xml.Elem =
<cctherm>
              <description>hot dog #5</description>
              <yearMade>1952</yearMade>
              <dateObtained>March 14, 2006</dateObtained>
              <bookPrice>2199</bookPrice>
              <purchasePrice>500</purchasePrice>
              <condition>9</condition>
            </cctherm>
```

ПРИМЕЧАНИЕ

Выражение new CCTherm в примере работает, несмотря на то что CCTherm является абстрактным классом, поскольку этот синтаксис в действительности создает экземпляр анонимного подкласса класса CCTherm. Анонимные классы мы рассматривали в разделе 20.5.

Кстати, если в качестве текста XML в код на Scala нужно включить фигурную скобку (`'{'` или `'}'`) вместо того, чтобы использовать ее в качестве управляющего символа, то нужно просто в одной строке записать две фигурные скобки подряд:

```
scala> <a> {{{{скобки как таковые!}}}} </a>
res7: scala.xml.Elem = <a> {{скобки как таковые!}} </a>
```

28.5. Разбор XML

Среди множества методов, доступных для работы с XML-классами, есть три метода, о которые непременно следует упомянуть. Они позволяют разбирать XML-данные на части, не обращая особого внимания на конкретный способ представления XML в Scala. Эти методы основаны на языке XPath, предназначенном для обработки данных в XML-формате. Как часто бывает при работе с языком Scala, их можно вводить в код на Scala напрямую, не привлекая какое-либо внешнее средство.

- **Извлечение текста.** Весь содержащийся в узле текст за вычетом любых тегов элемента можно получить, вызвав в отношении любого XML-узла метод `text`:

```
scala> <a>Sounds <tag/> good</a>.text
res8: String = Sounds good
```

Все закодированные символы автоматически раскодируются:

```
scala> <a> input ---&gt; output </a>.text
res9: String = " input ---> output "
```

- **Извлечение подэлементов.** Если нужно найти подэлемент по имени его тега, следует просто вызвать в отношении этого имени метод \:

```
scala> <a><b><c>hello</c></b></a> \ "b"
res10: scala.xml.NodeSeq = NodeSeq(<b><c>hello</c></b>)
```

Можно выполнить глубокий поиск и просмотреть подподэлементы и т. д., для чего вместо метода \ следует воспользоваться оператором \\:

```
scala> <a><b><c>hello</c></b></a> \ "c"
res11: scala.xml.NodeSeq = NodeSeq()

scala> <a><b><c>hello</c></b></a> \\ "c"
res12: scala.xml.NodeSeq = NodeSeq(<c>hello</c>)

scala> <a><b><c>hello</c></b></a> \ "a"
res13: scala.xml.NodeSeq = NodeSeq()

scala> <a><b><c>hello</c></b></a> \\ "a"
res14: scala.xml.NodeSeq =
NodeSeq(<a><b><c>hello</c></b></a>)
```

ПРИМЕЧАНИЕ

В Scala вместо имеющихся в XPath методов / и // используются методы \ и \\. Дело в том, что с пары символов // в Scala начинаются комментарии! Поэтому пришлось воспользоваться каким-то другим символом и для успешной работы применить другую разновидность слешей.

- **Извлечение атрибутов.** Атрибуты тега можно извлечь, используя все те же методы \ и \\. Просто перед именем атрибута следует поставить знак @:

```
scala> val joe = <employee
          name="Joe"
          rank="code monkey"
          serial="123"/>
joe: scala.xml.Elem = <employee name="Joe" rank="code monkey"
serial="123"/>

scala> joe \ "@name"
res15: scala.xml.NodeSeq = Joe

scala> joe \ "@serial"
res16: scala.xml.NodeSeq = 123
```

28.6. Десериализация

Теперь, используя рассмотренные методы разбора XML-данных на части, можно создать вторую часть сериализации, а именно анализатор, вновь переводящий XML-данные в вашу внутреннюю структуру данных. Например, данный код позволяет выполнить обратный анализ `CCTherm`-экземпляра:

```
def fromXML(node: scala.xml.Node): CCTherm =
  new CCTherm {
    val description   = (node \ "description").text
    val yearMade      = (node \ "yearMade").text.toInt
    val dateObtained  = (node \ "dateObtained").text
    val bookPrice     = (node \ "bookPrice").text.toInt
    val purchasePrice = (node \ "purchasePrice").text.toInt
    val condition     = (node \ "condition").text.toInt
  }
```

Этот код просматривает входной XML-узел по имени `node` с целью найти каждую из шести частей, необходимую для определения данных типа `CCTherm`. Данные, в смысле текст, извлекаются с помощью метода `.text` и оставляются как есть. Работа этого метода показана в следующем примере:

```
scala> val node = therm.toXML
node: scala.xml.Elem =
<cctherm>
              <description>hot dog #5</description>
              <yearMade>1952</yearMade>
              <dateObtained>March 14, 2006</dateObtained>
              <bookPrice>2199</bookPrice>
              <purchasePrice>500</purchasePrice>
              <condition>9</condition>
            </cctherm>

scala> fromXML(node)
res17: CCTherm = hot dog #5
```

28.7. Загрузка и сохранение

Осталась еще одна, последняя часть, необходимая для написания сериализатора данных, — преобразование между XML и потоками байтов. Реализовать эту часть легче всего, поскольку существуют библиотечные процедуры, которые все сделают за вас. Вам останется лишь вызвать для данных подходящую процедуру.

Чтобы преобразовать XML в строку, нужно всего лишь вызвать метод `toString`. Именно наличие работоспособного метода `toString` позволяет экспериментировать с XML в оболочке Scala.

Но лучше все же прибегнуть к библиотечной процедуре и полностью преобразовать все в байты. Здесь в получающийся на выходе XML нужно включить

директиву, которая указывает на используемую кодировку символов. Если вы сами кодируете строки в байты, то и ответственность за отслеживание кодировки символов лежит на вас.

Чтобы преобразовать XML в файл, состоящий из байтов, можно воспользоваться командой `XML.save`. Нужно указать имя файла и сохраняемый в нем узел:

```
scala.xml.XML.save("therm1.xml", node)
```

После запуска команды выше получившийся в результате этого файл `therm1.xml` примет такой вид:

```
<?xml version='1.0' encoding='UTF-8'?>
<cctherm>
             <description>hot dog #5</description>
             <yearMade>1952</yearMade>
             <dateObtained>March 14, 2006</dateObtained>
             <bookPrice>2199</bookPrice>
             <purchasePrice>500</purchasePrice>
             <condition>9</condition>
           </cctherm>
```

Загрузку выполнить проще, чем сохранение, поскольку файл включает все, что нужно знать загрузчику. Нужно всего лишь вызвать метод `XML.loadFile` в отношении имени файла:

```
scala> val loadnode = xml.XML.loadFile("therm1.xml")
loadnode: scala.xml.Elem =
<cctherm>
      <description>hot dog #5</description>
      <yearMade>1952</yearMade>
      <dateObtained>March 14, 2006</dateObtained>
      <bookPrice>2199</bookPrice>
      <purchasePrice>500</purchasePrice>
      <condition>9</condition>
    </cctherm>

scala> fromXML(loadnode)
res14: CCTherm = hot dog #5
```

Таковы основные необходимые вам методы. У этих методов загрузки и сохранения существует множество вариантов, включая методы для чтения и записи в разнообразные системы чтения и записи, входные и выходные потоки.

28.8. Сопоставление с образцом XML

До сих пор мы рассматривали методы разбора XML с помощью метода `text` и XPath-подобных методов `\` и `\\`. Хорошо, когда есть точные сведения о разновидности разбираемой XML-структуры. Но иногда в работе могут находиться сразу несколько из существующих XML-структур. Возможно, это

будут несколько разновидностей записей внутри данных, которые появились по причине того, что, помимо термометров, вы начали коллекционировать часы и тарелки. Может быть, вам захочется пропустить все пустые места между тегами. Какой бы ни была причина, отфильтровать все возможности поможет сопоставление с образцом.

XML-паттерн выглядит практически так же, как и XML-литерал. Основное отличие состоит в том, что при вставке управляющих символов {} код внутри них является не выражением, а паттерном. В паттерне, заключенном в символы {}, может использоваться полноценный язык паттернов Scala, включая привязку новых переменных, выполнение проверок соответствия типов и игнорирование содержимого с помощью паттернов и _ *. Рассмотрим простой пример:

```
def proc(node: scala.xml.Node): String =
  node match {
    case <a>{contents}</a> => "Это a: " + contents
    case <b>{contents}</b> => "Это b: " + contents
    case _ => "Это нечто иное."
}
```

В данной функции реализуется сопоставление с образцом, включающим три варианта. Первый ведет поиск <a>-элемента, содержимое которого состоит из одиночного подузла. Он привязывает это содержимое к переменной по имени contents, а затем вычисляет код, находящийся справа от соответствующей стрелки вправо (=>). Второй вариант делает то же самое, но ведет поиск не <a>-, а <b>-элемента, а третий вариант соответствует всему, что не соответствует любым другим вариантам. Посмотрим на данную функцию в действии:

```
scala> proc(<a>apple</a>)
res18: String = Это a: apple

scala> proc(<b>banana</b>)
res19: String = Это b: banana

scala> proc(<c>cherry</c>)
res20: String = Это нечто иное.
```

Скорее всего, данная функция не отвечает в полной мере вашим потребностям, поскольку ищет исключительно содержимое, которое находится внутри одиночного подузла <a> или <b>. Поэтому со следующими примерами она не справится:

```
scala> proc(<a>a <em>red</em> apple</a>)
res21: String = Это нечто иное.

scala> proc(<a/>)
res22: String = Это нечто иное.
```

Если нужно, чтобы функция находила соответствия в вариантах, подобных этому, можно вести поиск соответствия последовательности узлов, а не одному

узлу. Паттерн для любой последовательности XML-узлов записывается в виде _ *. Визуально этот паттерн последовательности похож на подстановочный паттерн (_), за которым следует звезда Клини в стиле регулярных выражений (*). Обновленная функция поиска соответствия не одиночному подэлементу, а последовательности подэлементов выглядит так:

```
def proc(node: scala.xml.Node): String =
  node match {
    case <a>{contents @ _ * }</a> => "Это a: " + contents
    case <b>{contents @ _ * }</b> => "Это b: " + contents
    case _ => "Это нечто иное."
}
```

Обратите внимание: в результате применения _ * с помощью @-паттерна, рассмотренного в разделе 15.2, выполняется привязка к переменной `contents`. А вот как новая версия выглядит в работе:

```
scala> proc(<a>a <em>red</em> apple</a>)
res23: String = It's an a: ArrayBuffer(a , <em>red</em>,
apple)

scala> proc(<a/>)
res24: String = It's an a: WrappedArray()
```

И заключительный совет: имейте в виду, что XML-паттерны отлично подходят для выражений `for` как способ обхода некоторых частей XML-дерева и игнорирования других его частей. Предположим, что вам нужно пропустить пустые места между записями в следующей XML-структуре:

```
val catalog =
  <catalog>
    <cctherm>
      <description>hot dog #5</description>
      <yearMade>1952</yearMade>
      <dateObtained>March 14, 2006</dateObtained>
      <bookPrice>2199</bookPrice>
      <purchasePrice>500</purchasePrice>
      <condition>9</condition>
    </cctherm>
    <cctherm>
      <description>Sprite Boy</description>
      <yearMade>1964</yearMade>
      <dateObtained>April 28, 2003</dateObtained>
      <bookPrice>1695</bookPrice>
      <purchasePrice>595</purchasePrice>
      <condition>5</condition>
    </cctherm>
  </catalog>
```

Визуально похоже на то, что внутри элемента `<catalog>` есть два узла. Однако в действительности их там пять. До, после и между двумя элементами имеются

пустые места! Если не принимать их во внимание, то можно некорректно обработать записи о термометрах:

```
catalog match {
  case <catalog>{therms @ _ * }</catalog> =>
    for (therm <- therms)
      println("processing: " +
              (therm \ "description").text)
}
```

```
processing:
processing: hot dog #5
processing:
processing: Sprite Boy
processing:
```

Обратите внимание на все строки, которые пытаются обработать пустые места, словно они являются настоящими записями о термометрах. В действительности же вам захочется проигнорировать пустые места и обработать только те подузлы, которые находятся внутри элемента `<cctherm>`. Это подмножество можно описать с помощью паттерна `<cctherm>{_ * }</cctherm>`, и можно ограничить выражение `for` обходом лишь тех записей, которые соответствуют этому паттерну:

```
catalog match {
  case <catalog>{therms @ _ * }</catalog> =>
    for (therm @ <cctherm>{_ * }</cctherm> <- therms)
      println("processing: " +
              (therm \ "description").text)
}
```

```
processing: hot dog #5
processing: Sprite Boy
```

Резюме

В этой главе мы представили краткий обзор тех возможностей, которые предоставляются вам для работы с XML. Существует множество других расширений, библиотек и инструментальных средств, и все они доступны для изучения. Одни из них настроены на работу со Scala, другие разработаны для Java, но могут применяться и в Scala, а третьи нейтральны по отношению к используемым языкам программирования. Целью данной главы было рассказать вам о способах, которые позволяют использовать слабоструктурированные данные для обмена информацией и получать доступ к таким данным с помощью поддержки XML, предоставляемой в Scala.

29 Модульное программирование с использованием объектов

В главе 1 мы утверждали, что один из способов придать программам на Scala масштабируемость основывается на возможности использования одних и тех же технологий для создания как малых, так и больших программ. До сих пор в книге основное внимание уделялось *программированию с прицелом на малое* — разработке и реализации совсем небольших фрагментов программ, на основе которых можно создавать гораздо более объемные программы[1]. Обратной стороной медали выступает *программирование с прицелом на большое* — организация и сборка маленьких фрагментов в более крупные программы, приложения или системы. Мы касались этой темы, когда в главе 13 говорили о пакетах и модификаторах доступа. Если коротко, то пакеты и модификаторы доступа позволяют организовать большую программу, используя пакеты в качестве *модулей*, под которым понимается небольшой фрагмент программы с четко определенным интерфейсом и скрытой реализацией.

Хотя само по себе деление программ на пакеты крайне полезно, оно ограниченно, поскольку не допускает абстракций. Невозможно перенастроить пакет для работы двумя разными способами в одной и той же программе, как нет возможности и организовать наследование между пакетами. Пакет неизменно содержит один абсолютно четкий перечень компонентов, который останется фиксированным до тех пор, пока не будет изменен сам код.

В этой главе мы обсудим, как с помощью объектно-ориентированных свойств Scala повышать модульность программ. Сначала покажем, как в качестве модуля может применяться простой объект-одиночка. Затем рассмотрим способы использования трейтов и классов как абстракций над модулями. Эти абстракции

---

[1] Эта терминология была введена в книге: *DeRemer F., Kron H.* Programming-in-the large versus programming-in-the-small // Proceedings of the international conference on Reliable software. — New York: ACM, 1975. — P. 114–121.

могут перенастраиваться на работу с множеством модулей и даже множество раз в рамках одной и той же программы. И наконец, покажем практичную технологию применения трейтов для деления модуля на несколько файлов.

29.1. Суть проблемы

По мере разрастания программы повышается актуальность задачи ее организации в виде модулей. Во-первых, возможность раздельной компиляции различных модулей, составляющих систему, помогает разным командам работать независимо друг от друга. Кроме того, возможность отключать одну реализацию модуля и подключать другую будет полезной потому, что позволит использовать различные конфигурации системы в разных контекстах, например при модульном тестировании на компьютере разработчика, интеграционном тестировании, установке и развертывании.

Например, вы можете работать с приложением, которое задействует базу данных и сервис сообщений. По мере написания кода может появиться потребность в запуске модульных тестов на том компьютере, где используются имитаторы как базы данных, так и сервиса сообщений, которые подходящим для тестирования образом имитируют работу этих сервисов, не требуя обращений по сети к совместно используемым ресурсам. В ходе интеграционного тестирования может потребоваться применить имитатор сервиса сообщений, но вживую обратиться к базе данных разработки. В ходе установки и, конечно же, развертывания в вашей организации предпочтение будет отдано использованию живых версий как базы данных, так и сервиса сообщений.

Любая технология, предназначением которой является создание такого рода модульности, должна обладать рядом обязательных свойств. Во-первых, иметь конструктор модулей, обеспечивающий качественное разделение интерфейса и реализации. Во-вторых, иметь способ заменить один модуль другим, с точно таким же интерфейсом, не внося изменений или не проводя повторную компиляцию тех модулей, работа которых зависит от заменяемого. И наконец, иметь возможность соединять модули в единое целое. Задача соединения должна восприниматься как *конфигурирование* системы.

Один из подходов к решению этой задачи — *внедрение зависимостей* (dependency injection), то есть использование технологии на платформе Java с применением таких фреймворков, как Spring и Guice, которые популярны у корпоративного сообщества Java[1]. К примеру, Spring позволяет, по сути, представлять интерфейс модуля в виде Java-интерфейса, а реализацию модуля — в виде Java-классов. Можно указывать зависимости между модулями и связывать приложение в единое целое с помощью внешних конфигурационных файлов в формате XML. Spring

[1] *Fowler M.* Inversion of Control Containers and the Dependency Injection pattern. January 2004 [Электронный ресурс]. — Режим доступа: martinfowler.com/articles/injection.html.

можно использовать со Scala и при этом применять заложенный в Spring подход к достижению модульности ваших Scala-программ на системном уровне, однако, располагая самим языком Scala, вы получаете ряд предоставляемых им альтернативных вариантов. Далее в главе мы покажем способы использования объектов в качестве модулей, позволяющие достичь модульности по-крупному без применения внешних фреймворков.

29.2. Приложение для работы с рецептами

Предположим, вы создаете корпоративное веб-приложение, позволяющее пользователям управлять рецептами, и хотите распределить программные средства по уровням, включая *уровень предметной области* и *уровень приложения*. На уровне предметной области будут определены *объекты предметной области*, которые будут включать бизнес-правила и понятия, а также будет инкапсулироваться состояние, сохраняемое во внешней реляционной базе данных. На уровне приложения будет предоставляться API, сведенный к понятиям тех сервисов, которые приложение предлагает клиентам (включая уровень пользовательского интерфейса). Уровень приложения будет нацелен на реализацию этих сервисов путем координации задач и делегирования работы с объектами уровню предметной области[1].

Хотелось бы на каждом из этих уровней иметь возможность подключать реальные версии или имитаторы того или иного объекта, чтобы проще было создавать модульные тесты для вашего приложения. Чтобы воплотить эти замыслы в жизнь, объекты, поведение которых нужно имитировать, следует рассматривать как модули. В Scala не возникает потребности в «измельчении» объектов, равно как и в использовании неких модулеподобных конструкций в целях их укрупнения. Масштабируемость Scala в немалой степени определяется тем обстоятельством, что одни и те же конструкции применяются как для мелких, так и для крупных структур.

Например, один из модулей будет сделан на основе одного из имитируемых компонентов уровня предметной области, который является объектом, представляющим собой базу данных. На уровне приложения объект «обозреватель базы данных» будет считаться модулем. В базе будут храниться все рецепты, собранные каким-либо человеком. Обозреватель поможет в поиске в базе данных и в просмотре ее записей, к примеру, с целью найти все рецепты, которые включают имеющийся у вас ингредиент.

Сначала нужно смоделировать продукты и рецепты. Чтобы ничего не усложнять, у продукта, как показано в листинге 29.1, есть только название. А у рецепта,

---

[1] Названия этих уровней соответствуют терминологии, принятой в публикации: *Evans E.* Domain-Driven Design: Tackling Complexity in the Heart of Software. — Addison-Wesley Professional, 2003.

как показано в листинге 29.2, будут название, список ингредиентов и ряд инструкций.

Листинг 29.1. Простой класс-сущность Food

```
package org.stairwaybook.recipe

abstract class Food(val name: String) {
  override def toString = name
}
```

Листинг 29.2. Простой класс-сущность Recipe

```
package org.stairwaybook.recipe

class Recipe(
  val name: String,
  val ingredients: List[Food],
  val instructions: String
) {
  override def toString = name
}
```

Классы `Food` и `Recipe`, показанные в листингах 29.1 и 29.2, представляют *сущности*, которые будут храниться в базе данных[1]. В листинге 29.3 показаны экземпляры-одиночки этих классов, которые могут применяться при написании тестов.

Листинг 29.3. Примеры Food и Recipe для использования в тестах

```
package org.stairwaybook.recipe

object Apple extends Food("Apple")
object Orange extends Food("Orange")
object Cream extends Food("Cream")
object Sugar extends Food("Sugar")

object FruitSalad extends Recipe(
  "fruit salad",
  List(Apple, Orange, Cream, Sugar),
  "Stir it all together."
)
```

Для модулей в Scala используются объекты, поэтому сборку вашей программы можно начать с создания двух объектов-одиночек, которые в ходе тестирования послужат имитаторами реализации модулей базы данных и обозревателя. Это будут

[1] Эти классы-сущности упрощены, чтобы не засорять примеры обилием подробностей из реального мира. Но превращение этих классов, к примеру, в сущности, которые могут храниться с помощью Hibernate или Java Persistence Architecture, потребует всего нескольких изменений, таких как добавление закрытого поля id типа Long и конструктора без аргументов, аннотирование полей с помощью scala.reflect.BeanProperty, спецификация соответствующих отображений через аннотации или отдельный XML-файл и т. д.

имитаторы, так что модуль базы данных основан на простом хранящемся в памяти списке. Реализации этих объектов показаны в листинге 29.4.

Листинг 29.4. Модули-имитаторы базы данных и обозревателя

```
package org.stairwaybook.recipe

object SimpleDatabase {
  def allFoods = List(Apple, Orange, Cream, Sugar)

  def foodNamed(name: String): Option[Food] =
    allFoods.find(_.name == name)

  def allRecipes: List[Recipe] = List(FruitSalad)
}

object SimpleBrowser {
  def recipesUsing(food: Food) =
    SimpleDatabase.allRecipes.filter(recipe =>
      recipe.ingredients.contains(food))
}
```

Использовать эти базу данных и обозреватель можно следующим образом:

```
scala> val apple = SimpleDatabase.foodNamed("Apple").get
apple: Food = Apple

scala> SimpleBrowser.recipesUsing(apple)
res0: List[Recipe] = List(fruit salad)
```

Все может стать еще интереснее. Для этого предположим, что в базе данных продукты отсортированы по категориям. Чтобы реализовать это, можно, как показано в листинге 29.5, добавить класс `FoodCategory` и список всех категорий в базе данных. Обратите внимание: в этом примере ключевое слово `private`, столь полезное для реализации классов, пригодилось и для реализации модулей. Элементы, обозначенные как приватные (`private`), являются частью реализации модуля, и поэтому их особенно просто изменять, не оказывая никакого влияния на все остальные модули.

На данном этапе могут быть добавлены и многие другие объекты, но главное сейчас — ухватить саму идею. Программы можно разделить на объекты-одиночки, о которых можно думать, что это модули. В этом нет ничего особенного, но такой подход к делу окажется весьма полезен при обдумывании абстракции, рассматриваемой далее.

Листинг 29.5. Модули базы данных и обозревателя с добавлением категорий

```
package org.stairwaybook.recipe

object SimpleDatabase {
  def allFoods = List(Apple, Orange, Cream, Sugar)

  def foodNamed(name: String): Option[Food] =
    allFoods.find(_.name == name)
```

```
  def allRecipes: List[Recipe] = List(FruitSalad)

  case class FoodCategory(name: String, foods: List[Food])

  private var categories = List(
    FoodCategory("fruits", List(Apple, Orange)),
    FoodCategory("misc", List(Cream, Sugar)))

  def allCategories = categories
}

object SimpleBrowser {
  def recipesUsing(food: Food) =
    SimpleDatabase.allRecipes.filter(recipe =>
      recipe.ingredients.contains(food))

  def displayCategory(category: SimpleDatabase.FoodCategory) = {
    println(category)
  }
}
```

29.3. Абстракция

С помощью показанных до сих пор примеров ваше приложение удалось разбить на отдельные модули базы данных и обозревателя, однако конструкция все еще не обладает достаточной модульностью. Дело в том, что в ней имеется по сути жесткая ссылка из модуля обозревателя на модуль базы данных

```
SimpleDatabase.allRecipes.filter(recipe => ...
```

В модуле `SimpleBrowser` упоминается по имени модуль `SimpleDatabase`, поэтому вы не сможете подключить другую реализацию модуля базы данных, не внося изменений и не выполнив повторную компиляцию модуля обозревателя. Кроме того, несмотря на то, что из `SimpleDatabase` на модуль `SimpleBrowser` жесткой ссылки нет[1], не вполне понятно, как, например, позволить уровню пользовательского интерфейса быть перенастроенным на использование другой реализации модуля обозревателя.

Но при повышении подключаемости модулей важно избегать дублирования, поскольку различные реализации одного и того же модуля будут, скорее всего, совместно использовать большой объем кода. Предположим, к примеру, что для поддержки баз данных с рецептами требуется одна и та же кодовая база, но при этом вам нужна возможность создать для каждой из них отдельный обозреватель. Для каждого из этих экземпляров вам захочется повторно задействовать код обо-

[1] И в этом нет ничего плохого, поскольку каждый из этих архитектурных уровней должен зависеть только от уровней ниже.

зревателя, поскольку обозреватели отличаются друг от друга лишь той базой, к которой они обращаются. Кроме реализации, касающейся БД, весь остальной код может быть использован повторно без малейших изменений. Как построить программу таким образом, чтобы свести дублирование кода к минимуму? Как сделать код перенастраиваемым, чтобы его можно было настроить под использование любой реализации базы данных?

Ответ известен: если модуль — это объект, то шаблоном для модуля выступает класс. Точно так же, как класс может содержать описание общих частей всех своих экземпляров, он может описывать части модуля, являющиеся общими для всех его возможных конфигураций.

Поэтому определение обозревателя становится классом, а не объектом, а указание на используемую базу данных является, как показано в листинге 29.6, абстрактным членом класса. База также становится классом, который включает по максимуму все общее, что есть у всех баз данных, и объявляет недостающие части, которые должна определять база. В этом случае во всех модулях баз данных должны определяться методы для `allFoods`, `allRecipes` и `allCategories`, но в них может использоваться произвольное определение, поэтому в классе `Database` методы должны оставаться абстрактными. В отличие от них метод `foodNamed`, как показано в листинге 29.7, может быть определен в абстрактном классе `Database`.

Листинг 29.6. Класс Browser с абстрактной val-переменной database

```
abstract class Browser {
  val database: Database

  def recipesUsing(food: Food) =
    database.allRecipes.filter(recipe =>
      recipe.ingredients.contains(food))

  def displayCategory(category: database.FoodCategory) = {
    println(category)
  }
}
```

Листинг 29.7. Класс Database с абстрактными методами

```
abstract class Database {
  def allFoods: List[Food]
  def allRecipes: List[Recipe]

  def foodNamed(name: String) =
    allFoods.find(f => f.name == name)

  case class FoodCategory(name: String, foods: List[Food])
  def allCategories: List[FoodCategory]
}
```

Объект `SimpleDatabase` нужно обновить, чтобы он, как показано в листинге 29.8, стал наследником абстрактного класса `Database`.

Листинг 29.8. Объект SimpleDatabase как подкласс класса Database

```
object SimpleDatabase extends Database {
  def allFoods = List(Apple, Orange, Cream, Sugar)

  def allRecipes: List[Recipe] = List(FruitSalad)

  private var categories = List(
    FoodCategory("fruits", List(Apple, Orange)),
    FoodCategory("misc", List(Cream, Sugar)))

  def allCategories = categories
}
```

Затем, как показано в листинге 29.9, с помощью создания экземпляра класса `Browser` и указания, какую именно базу данных использовать, создается конкретный модуль-обозреватель.

Листинг 29.9. Объект SimpleBrowser как подкласс класса Browser

```
object SimpleBrowser extends Browser {
  val database = SimpleDatabase
}
```

Эти более пригодные к подключению модули можно использовать как и прежде:

```
scala> val apple = SimpleDatabase.foodNamed("Apple").get
apple: Food = Apple

scala> SimpleBrowser.recipesUsing(apple)
res1: List[Recipe] = List(fruit salad)
```

Вот теперь можно приступить к созданию имитатора второй базы данных и воспользоваться для него, как показано в листинге 29.10, тем же классом обозревателя.

Листинг 29.10. База данных студентов и обозреватель

```
object StudentDatabase extends Database {
  object FrozenFood extends Food("FrozenFood")

  object HeatItUp extends Recipe(
    "heat it up",
    List(FrozenFood),
    "Microwave the 'food' for 10 minutes.")

  def allFoods = List(FrozenFood)
  def allRecipes = List(HeatItUp)
  def allCategories = List(
    FoodCategory("edible", List(FrozenFood)))
}

object StudentBrowser extends Browser {
  val database = StudentDatabase
}
```

29.4. Разбиение модулей на трейты

Зачастую модуль оказывается слишком большим, чтобы комфортно уместиться в одном файле. В таком случае разбить модуль на несколько отдельных файлов можно с помощью трейтов. Предположим, к примеру, что вам нужно переместить код категоризации за пределы основного файла `Database`, поместив его в собственный файл. Для этого кода, как показано в листинге 29.11, можно создать трейт.

Листинг 29.11. Трейт для категорий продуктов

```
trait FoodCategories {
  case class FoodCategory(name: String, foods: List[Food])
  def allCategories: List[FoodCategory]
}
```

Теперь, как показано в листинге 29.12, вместо того чтобы определять `FoodCategory` и `allCategories` в классе `Database`, он может примешать трейт `FoodCategories`.

Листинг 29.12. Класс Database, примешивающий трейт FoodCategories

```
abstract class Database extends FoodCategories {
  def allFoods: List[Food]
  def allRecipes: List[Recipe]
  def foodNamed(name: String) =
    allFoods.find(f => f.name == name)
}
```

Можно попробовать разбить `SimpleDatabase` на два трейта: один для продуктов, а другой для рецептов. Это позволит определить `SimpleDatabase` так, как показано в листинге 29.13.

Листинг 29.13. Объект SimpleDatabase, состоящий исключительно из примесей

```
object SimpleDatabase extends Database
with SimpleFoods with SimpleRecipes
```

Трейт `SimpleFoods` может выглядеть так, как показано в листинге 29.14.

Листинг 29.14. Трейт SimpleFoods

```
trait SimpleFoods {
  object Pear extends Food("Pear")
  def allFoods = List(Apple, Pear)
  def allCategories = Nil
}
```

Пока вроде бы все получается, но, к сожалению, при попытке определить трейт для `SimpleRecipe` следующим образом возникнет проблема:

```
trait SimpleRecipes { // Не пройдет компиляцию
  object FruitSalad extends Recipe(
    "fruit salad",
    List(Apple, Pear), // Опа!
```

```
    "Mix it all together."
  )
  def allRecipes = List(FruitSalad)
}
```

Дело в том, что `Pear` находится в другом трейте, а не в том, который его использует, следовательно, пребывает вне области видимости. У компилятора нет никаких догадок насчет того, что `SimpleRecipes` когда-либо смешивается с `SimpleFoods`.

Но компилятору можно сообщить об этом. Именно для такой ситуации в Scala имеется *собственный тип*. С технической точки зрения он является предполагаемым типом для `this`, когда тот упоминается в пределах класса. С прагматической — указывает требования к любому конкретному классу, который примешивает данный трейт. При наличии трейта, который когда-либо примешивается с другим трейтом или трейтами, можно указать, что эти другие трейты должны предполагаться. В нашем варианте, как показано в листинге 29.15, достаточно указать собственный тип `SimpleFoods`.

Листинг 29.15. Трейт SimpleRecipes с собственным типом

```
trait SimpleRecipes {
  this: SimpleFoods =>

  object FruitSalad extends Recipe(
    "fruit salad",
    List(Apple, Pear), // Теперь Pear находится в области видимости
    "Mix it all together."
  )
  def allRecipes = List(FruitSalad)
}
```

Теперь благодаря новому собственному типу `Pear` находится в области видимости. Подразумевается, что ссылка на `Pear` должна выглядеть как `this.Pear`. Эта ссылка безопасна, поскольку любой конкретный класс, примешивающий `SimpleRecipes`, должен также быть подтипом `SimpleFoods`, следовательно, `Pear` будет членом подтипа. Абстрактные подклассы и трейты не должны придерживаться этого ограничения, но, так как создать их экземпляры с помощью ключевого слова `new` невозможно, риск того, что ссылка `this.Pear` даст сбой, отсутствует.

29.5. Компоновка во время выполнения

Модули Scala могут быть скомпонованы во время выполнения программы, и в зависимости от вычислений в ходе выполнения программы можно решить, какие именно модули должны быть скомпонованы. Например, в листинге 29.16 показана небольшая программа, которая выбирает базу данных, а затем выводит на стандартное устройство все имеющиеся в ней яблочные рецепты.

Листинг 29.16. Приложение, динамически выбирающее реализацию модуля

```
object GotApples {
  def main(args: Array[String]) = {
    val db: Database =
      if(args(0) == "student")
        StudentDatabase
      else
        SimpleDatabase

    object browser extends Browser {
      val database = db
    }

    val apple = SimpleDatabase.foodNamed("Apple").get

    for(recipe <- browser.recipesUsing(apple))
      println(recipe)
  }
}
```

Теперь, если обратиться к простой базе данных, то найдется рецепт фруктового салата. Если же воспользоваться базой данных студентов, то не найдется вообще никаких рецептов с яблоками:

```
$ scala GotApples simple
fruit salad
$ scala GotApples student
$
```

Конфигурирование с помощью кода Scala

Объект `GotApples`, показанный в листинге 29.16, содержит жесткие ссылки как на `StudentDatabase`, так и на `SimpleDatabase`, поэтому может возникнуть вопрос: не отступаем ли мы от решения проблемы жестких ссылок в исходных примерах данной главы? Разница в том, что жесткие ссылки локализованы в одном файле, который можно заменить.

Каждое модульное приложение нуждается в каком-либо способе, позволяющем указывать на ту действующую реализацию модуля, которая используется в конкретной ситуации. Это действие конфигурирования приложения будет по определению затрагивать имена конкретных реализаций модуля. Например, в приложении Spring конфигурирование путем указания имен реализаций осуществляется во внешнем XML-файле. В Scala конфигурирование можно выполнять с помощью самого кода Scala. Одно из преимуществ использования для конфигурирования исходного кода на Scala над XML заключается в том, что процесс прогона вашего конфигурационного файла через компилятор Scala вскроет в нем все опечатки еще до того, как вы начнете применять его на практике.

29.6. Отслеживание экземпляров модулей

Несмотря на использование одного и того же кода, различные модули обозревателей и баз данных, созданные в предыдущем разделе, в действительности являются отдельными модулями. Это значит, у каждого модуля есть собственное содержимое, включая любые вложенные классы. К примеру, `FoodCategory` в `SimpleDatabase` — это класс, который отличается от `FoodCategory` в `StudentDatabase`!

```
scala> val category = StudentDatabase.allCategories.head
category: StudentDatabase.FoodCategory =
FoodCategory(edible,List(FrozenFood))

scala> SimpleBrowser.displayCategory(category)
                                     ^

       error: type mismatch;
        found    : StudentDatabase.FoodCategory
         required: SimpleBrowser.database.FoodCategory
```

Но если вы, напротив, предпочитаете, чтобы все `FoodCategory` представляли собой одно и то же, то добиться этого можно, переместив определение `FoodCategory` за пределы любого класса или трейта.

Выбор за вами, но в том виде, в котором это уже написано, каждый класс `Database` получает собственный уникальный класс `FoodCategory`.

Два класса `FoodCategory`, показанные в данном примере, действительно разные, и потому компилятор вправе пожаловаться. Но иногда можно столкнуться со случаем, когда два типа одинаковы, но компилятор не в состоянии проверить это. Вы увидите: компилятор жалуется на то, что два типа разные, притом что вы как программист знаете, что они одинаковы.

В таких случаях зачастую удается решить задачу, используя *типы-одиночки*. К примеру, в программе `GotApples` система проверки типов не знает, что `db` и `browser.database` — это одно и то же. Попытка передавать категории между двумя объектами повлечет за собой возникновение ошибок несоответствия типов:

```
object GotApples {
  // Одинаковые определения...

  for (category <- db.allCategories)
    browser.displayCategory(category)

  // ...
}
```

```
GotApples2.scala:14: error: type mismatch;
 found : db.FoodCategory
 required: browser.database.FoodCategory
        browser.displayCategory(category)
                                ^
one error found
```

Чтобы предотвратить эту ошибку, нужно проинформировать систему проверки типов о том, что они являются одним и тем же объектом. Сделать это можно, изменив определение `browser.database`, как показано в листинге 29.17.

Листинг 29.17. Использование типа-одиночки

```
object browser extends Browser {
  val database: db.type = db
}
```

Это определение такое же, как и прежде, за исключением того, что у `database` имеется странно выглядящий тип `db.type`. Часть `.type` в конце означает, что это *тип-одиночка*, который имеет совершенно особую природу и содержит только один объект, в данном случае именно тот, на который ссылается `db`. Обычно такие типы слишком специфичны, чтобы быть полезными, именно поэтому компилятор не хочет вставлять их автоматически. Но в данном случае тип-одиночка позволяет компилятору узнать, что `db` и `browser.database` — это один и тот же объект, то есть дает достаточно информации, чтобы избежать появления ошибки несоответствия типов.

Резюме

В этой главе мы рассмотрели использование объектов Scala в качестве модулей. Данный подход раскрывает перед вами разнообразные способы создания не только простых статических модулей, но и абстрактных, перенастраиваемых модулей. В действительности существует гораздо больше технологий создания абстракций, чем мы показали, поскольку все, что работает с обычными классами, работает и с классами, применяемыми для реализации модуля. По обыкновению, в какой мере воспользоваться этим широким разнообразием возможностей — дело вкуса.

Модули — часть программирования с прицелом на большое, и поэтому с ними сложно экспериментировать. Чтобы реально оценить ситуацию, понадобится какая-нибудь крупная программа. И все же, прочитав главу, вы узнали, каким свойствам Scala следует уделить внимание при желании перейти к программированию в модульном стиле. Теперь вы можете поразмышлять о применении этих технологий при написании собственных крупных программ и распознать показанные здесь программные шаблоны, когда они встретятся в чьем-нибудь программном коде.

30 Равенство объектов

Сравнение двух значений на равенство встречается в программировании повсеместно. Выполнить эту операцию порой не так просто, как кажется на первый взгляд. В данной главе мы подробно рассмотрим равенство объектов и дадим ряд рекомендаций по разработке собственных тестов на равенство.

30.1. Понятие равенства в Scala

Как упоминалось в разделе 11.2, способы определить равенство в Scala и в Java различаются. В Java их два: задействовать оператор `==`, применяемый обычно для определения равенства типов значений и идентичности объектов ссылочных типов, и прибегнуть к методу `equals` — для выявления классического равенства (определяемого пользователем) ссылочных типов. Это положение не лишено проблем, поскольку более естественный символ `==` не всегда соответствует естественному понятию равенства. При программировании на Java новички часто спотыкаются на сравнении объектов с применением оператора `==`, в то время как проверять их на равенство следует с помощью метода `equals`. Например, сравнение двух строк, `x` и `y`, с использованием выражения `x == y` может в Java выдать значение `false`, даже если `x` и `y` содержат одинаковые символы, расположенные в одном и том же порядке.

В Scala также есть метод равенства, обозначающий идентичность объектов, однако он используется нечасто. Эта разновидность равенства, записываемая в виде `x eq y`, вычисляется в `true`, если `x` и `y` ссылаются на один и тот же объект. Знак равенства `==` зарезервирован в Scala для обозначения естественного равенства для каждого типа. Для типов значений `==` служит показателем, как и в Java, сравнения значений. Для ссылочных типов `==` является в Scala аналогом метода `equals`. Поведение метода `==` для новых типов можно переопределить путем переопределения метода `equals`, который всегда наследуется из класса

Any. Унаследованный метод equals, работающий до переопределения, выявляет, как и в Java, идентичность объектов. Следовательно, equals (а с ним заодно и ==) изначально выступает аналогом eq, но вы можете изменить его поведение, переопределив метод equals в определяемых вами классах. Переопределить == напрямую невозможно, поскольку он в классе Any определен как final-метод. То есть в Scala метод == рассматривается так, словно был определен в классе Any следующим образом:

```
final def == (that: Any): Boolean =
  if (null eq this) {null eq that} else {this equals that}
```

30.2. Написание метода равенства

Как должен быть определен метод equals? Оказывается, правильно написать метод равенства в объектно-ориентированных языках удивительно сложно. Изучив большой объем Java-кода, авторы статьи, вышедшей в 2007 году, пришли к выводу, что почти все реализации методов equals не лишены недостатков[1]. Проблема в том, что равенство лежит в основе многих других вещей. Скажем, дефектный метод равенства для типа C может означать, что не получится гарантированно поместить объект типа C в коллекцию.

Например, у вас есть два относящихся к типу C элемента, elem1 и elem2, равных друг другу (то есть выражение elem1 equals elem2 выдает true). Несмотря на это, при часто допускаемой дефектной реализации метода equals можно все же столкнуться с таким поведением:

```
var hashSet: Set[C] = new collection.immutable.HashSet
hashSet += elem1
hashSet contains elem2 // Возвращает false!
```

Неверное поведение при переопределении метода equals может быть вызвано четырьмя основными ловушками[2].

1. Определение equals с неверной сигнатурой.
2. Изменение equals без изменения hashCode.
3. Определение equals в терминах изменяемых полей.
4. Ошибка в определении equals как отношения эквивалентности.

Далее нам предстоит рассмотреть эти четыре ловушки более подробно.

---

[1] *Vaziri M., Tip F., Fink S., Dolby J.* Declarative Object Identity Using Relation Types // Proc. ECOOP 2007, 2007. — P. 54–78.

[2] Все ловушки, за исключением третьей, рассмотрены в контексте Java в книге: *Блох Дж.* Java. Эффективное программирование. 3-е изд. — М.: Вильямс, 2018.

Ловушка № 1. Определение equals с неверной сигнатурой

Рассмотрим добавление метода равенства к следующему классу простых точек:

```
class Point(val x: Int, val y: Int) { ... }
```

Кажущийся очевидным, но неверный способ — определить его так:

```
// Абсолютно ошибочное определение equals
def equals(other: Point): Boolean =
  this.x == other.x && this.y == other.y
```

Что не так с этим методом? На первый взгляд, все работает довольно успешно:

```
scala> val p1, p2 = new Point(1, 2)
p1: Point = Point@37d7d90f
p2: Point = Point@3beb846d

scala> val q = new Point(2, 3)
q: Point = Point@e0cf182

scala> p1 equals p2
res0: Boolean = true

scala> p1 equals q
res1: Boolean = false
```

Но неприятности начнутся, как только вы решите поместить точки в коллекцию:

```
scala> import scala.collection.mutable
import scala.collection.mutable

scala> val coll = mutable.HashSet(p1)
coll: scala.collection.mutable.HashSet[Point] =
Set(Point@37d7d90f)

scala> coll contains p2
res2: Boolean = false
```

Чем объяснить, что `coll` не содержит `p2`, несмотря на то что объект `p1` был добавлен в коллекцию, а `p1` и `p2` являются равными объектами? Причина становится понятной при следующем взаимодействии, где точный тип одной из сравниваемых точек маскируется. Определите в качестве псевдонима `p2` переменную `p2a`, но вместо типа `Point` укажите для нее тип `Any`:

```
scala> val p2a: Any = p2
p2a: Any = Point@3beb846d
```

Теперь при повторе первого сравнения, но уже с псевдонимом `p2a`, а не `p2` будет получен следующий результат:

```
scala> p1 equals p2a
res3: Boolean = false
```

Что пошло не так? Используемая прежде версия `equals` не переопределила стандартный метод `equals`, поскольку ее тип отличается. Вот каким является тип метода `equals`, определенного в корневом классе `Any`[1]:

```
def equals(other: Any): Boolean
```

Поскольку метод `equals` в `Point` получает в качестве аргумента не `Any`, а `Point`, то метод `equals` в `Any` не переопределяется. Вместо этого просто перегружается альтернативный вариант. Перегрузка в Scala и Java разрешается за счет аргумента статического типа, а не типа, определяемого во время выполнения программы. А при условии, что статическим типом аргумента является `Point`, вызывается метод `equals`, который определен в `Point`. Но так как статический аргумент относится к типу `Any`, вместо этого вызывается метод `equals`, который определен в `Any`. Данный метод не был переопределен, поэтому по-прежнему реализован на основе определения идентичности объектов.

Именно поэтому сравнение `p1 equals p2a` выдает значение `false`, хотя у точек `p1` и `p2a` одинаковые значения `x` и `y`. Кроме того, по этой причине метод `contains` в `HashSet` возвращает `false`. Поскольку данный метод работает с универсальными множествами, то вызывает универсальный метод `equals`, который определен в классе `Object`, а не перегруженный вариант, определенный в `Point`. А вот как выглядит более подходящий метод `equals`:

```
// Уже лучше, но еще не идеально
override def equals(other: Any) = other match {
  case that: Point => this.x == that.x && this.y == that.y
  case _ => false
}
```

Теперь у `equals` правильный тип. Метод получает в качестве параметра значение типа `Any` и выдает результат типа `Boolean`. В реализации этого метода используется сопоставление с образцом. Сначала в нем проверяется принадлежность других объектов к тому же типу `Point`. Если она подтвердится, то сравниваются координаты двух точек и возвращается результат. В противном случае результатом становится значение `false`.

В данном случае можно допустить просчет, определив `==` с неверной сигнатурой. Обычно, если попытаться переопределить `==` с правильной сигнатурой, получающей аргумент типа `Any`, то компилятор выдаст ошибку, поскольку это будет рассматриваться как попытка перезаписать финальный метод типа `Any`.

Новички в Scala порой допускают сразу две ошибки: пытаются переопределить `==` и дают этому методу неверную сигнатуру, например:

```
def ==(other: Point): Boolean = // Не делайте этого!
```

---

[1] При наличии большой практики программирования на Java вы, возможно, ожидали, что аргумент для этого метода будет относиться к типу Object, а не к типу Any. Напрасно беспокоитесь: это все тот же метод equals. Компилятор просто делает вид, что он относится к типу Any.

В данном случае определенный пользователем метод == рассматривается в качестве перегруженного варианта метода с таким же именем в классе Any, и программа проходит компиляцию, но вести себя будет так же сомнительно, как при определении equals с неверной сигнатурой.

Ловушка № 2. Изменение equals без изменения hashCode

Продолжим рассматривать пример из ловушки № 1. Если повторить сравнение p1 и p2a с самым последним определением Point, то будет получен, как и ожидалось, результат true. Но если повторить тест HashSet.contains, то, наверное, опять будет получен результат false:

```
scala> val p1, p2 = new Point(1, 2)
p1: Point = Point@122c1533
p2: Point = Point@c23d097

scala> collection.mutable.HashSet(p1) contains p2
res4: Boolean = false
```

Но подобный исход вероятен не на все 100 %. В ходе эксперимента можно получить и true. Если результат именно такой, то можно поэкспериментировать с какими-нибудь другими точками с координатами 1 и 2. Со временем найдется одна, не присутствующая во множестве. Дело в том, что в Point метод equals переопределен без сопутствующего ему переопределения метода hashCode.

Учтите, что коллекция в данном примере относится к типу HashSet. Следовательно, элементы коллекции помещаются в хеш-корзины, определяемые их хеш-кодом. Тест contains сначала определяет хеш-корзину, в которой следует искать, а затем сравнивает заданные элементы со всеми ее элементами. Далее, последняя версия класса Point обновила equals, но в то же самое время не обновила hashCode. Следовательно, hashCode остался таким же, каким был в своей версии в классе AnyRef, — выполняющим некоторые преобразования адреса размещаемого объекта.

Хеш-коды p1 и p2 практически всегда разные, даже притом, что поля обеих точек имеют одинаковые значения. А разные хеш-коды с высокой степенью вероятности означают разные хеш-корзины во множестве. Тест contains станет искать соответствующие элементы в корзине с хеш-кодом, который соответствует хеш-коду p2. В большинстве случаев точка p1 будет в другой корзине, следовательно, никогда не будет найдена. Точки p1 и p2 могут случайно оказаться и в одной и той же корзине. Тогда тест возвратит значение true. Проблема в том, что эта полезная реализация Point нарушила соглашение в отношении hashCode, определенное для класса Any[1]: *Если два объекта равны с точки зрения метода equals, то вызов метода hashCode в отношении каждого из двух объектов должен выдавать один и тот же целочисленный результат.*

[1] Текст соглашения метода hashCode класса Any навеян документацией Javadoc класса java.lang.Object.

Фактически в Java хорошо известно, что `hashCode` и `equals` должны всегда переопределяться вместе. Более того, `hashCode` может зависеть только от тех полей, от которых зависит `equals`. Для класса `Point` подходящее определение `hashCode` могло бы быть выражено так:

```
class Point(val x: Int, val y: Int) {
  override def hashCode = (x, y).##
  override def equals(other: Any) = other match {
    case that: Point => this.x == that.x && this.y == that.y
    case _ => false
  }
}
```

Это всего лишь одна из множества возможных реализаций `hashCode`. Следует напомнить, что метод `##` является сокращенной формой записи метода для вычисления хеш-кодов, работающего с примитивными значениями, ссылочными типами и `null`. При вызове в отношении коллекции или кортежа он вычисляет смешанный хеш, чувствительный к хеш-кодам всех элементов в коллекции. Дополнительные указания по написанию кода метода `hashCode` будут даны чуть позже.

Добавление `hashCode` решает проблему равенства при определении таких классов, как `Point`, но есть и другие тревожные моменты, которые нужно отслеживать.

Ловушка № 3. Определение equals в терминах изменяемых полей

Рассмотрим следующую небольшую вариацию класса `Point`:

```
class Point(var x: Int, var y: Int) { // Проблемное место
  override def hashCode = (x, y).##
  override def equals(other: Any) = other match {
    case that: Point => this.x == that.x && this.y == that.y
    case _ => false
  }
}
```

Единственное различие заключается в том, что теперь поля `x` и `y` относятся к `var`-, а не к `val`-переменным. Теперь методы `equals` и `hashCode` определены относительно этих изменяемых полей, следовательно, их результаты изменяются при изменении полей. Если поместить точки в коллекции, то это может привести к странным последствиям:

```
scala> val p = new Point(1, 2)
p: Point = Point@5428bd62

scala> val coll = collection.mutable.HashSet(p)
coll: scala.collection.mutable.HashSet[Point] =
Set(Point@5428bd62)

scala> coll contains p
res5: Boolean = true
```

Теперь, если изменить поле в точке p, то будет ли коллекция по-прежнему содержать ее? Посмотрим:

```
scala> p.x += 1

scala> coll contains p
res7: Boolean = false
```

Выглядит странно. Куда же делась p? Еще более странные результаты будут получены, если проверить, содержит ли p итератор множества:

```
scala> coll.iterator contains p
res8: Boolean = true
```

Итак, есть множество, не содержащее p, но тем не менее p находится среди элементов множества! Получилось, что после изменения поля x точка p оказалась в другой хеш-корзине множества coll. То есть исходная хеш-корзина больше не соответствует новому значению хеш-кода точки. Иначе говоря, точка p исчезла из поля зрения множества coll, хотя все еще принадлежит его элементам.

Из этого примера нужно извлечь урок: когда equals и hashCode зависят от изменяемого состояния, возникают проблемы для потенциальных пользователей. Если поместить подобные объекты в коллекцию, придется проявлять осмотрительность и никогда не изменять зависимое состояние, а сделать это не слишком просто. Если возникнет потребность в сравнении, которое принимает в расчет текущее состояние объекта, то его следует назвать как-то иначе, но ни в коем случае не equals.

Относительно последнего определения Point: было бы куда более предпочтительно избавиться от переопределения hashCode и назвать метод сравнения equalContents или каким-нибудь другим именем, отличающимся от equals. Тогда Point унаследует исходную реализацию equals и hashCode, а p будет размещаться в coll даже после изменения его поля x.

Ловушка № 4. Ошибка в определении equals как отношения эквивалентности

В соглашении по методу equals в scala.Any указывается, что в equals должны быть реализованы отношения эквивалентности для отличных от null объектов[1], выражающиеся в совокупности следующих свойств:

- *рефлексивности* — для любого отличного от null значения x выражение x.equals(x) должно возвращать true;
- *симметричности* — для любых отличных от null значений x и y выражение x.equals(y) должно возвращать true тогда и только тогда, когда выражение y.equals(x) возвращает true;

[1] Как и в случае с hashCode, имеющееся в Any соглашение относительно метода equals основано на соглашении по методу equals, определяемому в java.lang.Object.

- *транзитивности* — если для любых отличных от `null` значений `x`, `y` и `z` `x.equals(y)` возвращает `true` и `y.equals(z)` возвращает `true`, то и `x.equals(z)` должно возвращать `true`;
- *постоянства* — для любых отличных от `null` значений `x` и `y` множественные вызовы `x.equals(y)` должны неизменно возвращать `true` или же неизменно возвращать `false` при условии, что информация в сравнениях объектов на равенство не была изменена;
- для любого отличного от `null` значения `x` выражение `x.equals(null)` должно возвращать `false`.

Определение `equals`, разработанное для класса `Point`, на данный момент удовлетворяет соглашению по `equals`. Но все усложняется, как только дело касается подклассов. Предположим существование подкласса `ColoredPoint` класса `Point`, который добавляет поле `color`, имеющее тип `Color`. Допустим также, что `Color` определен как перечисление, представленное в разделе 20.9:

```
object Color extends Enumeration {
  val Red, Orange, Yellow, Green, Blue, Indigo, Violet = Value
}
```

В `ColoredPoint` метод `equals` переопределяется, чтобы учитывалось новое поле `color`:

```
class ColoredPoint(x: Int, y: Int, val color: Color.Value)
    extends Point(x, y) {   // Проблема: утрачена симметричность равенства

  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
      this.color == that.color && super.equals(that)
    case _ => false
 }
}
```

Именно такой код, скорее всего, напишут многие программисты. Обратите внимание: в таком случае классу `ColoredPoint` не нужно переопределять метод `hashCode`. Поскольку новое определение `equals` в `ColoredPoint` строже, чем переделанное определение в `Point` (то есть он определяет равенство меньшего количества пар объектов), соглашение по поводу `hashCode` остается ненарушенным. Если две цветные точки равны, то должны иметь и одинаковые координаты, следовательно, гарантируется равенство и их хеш-кодов.

Если ограничиться рассмотрением лишь класса `ColoredPoint` как такового, то в имеющемся в нем определении `equals` вроде бы все в порядке. Но соглашение по `equals` нарушено, поскольку смешиваются простые и цветные точки. Рассмотрим следующий диалог с интерпретатором:

```
scala> val p = new Point(1, 2)
p: Point = Point@5428bd62

scala> val cp = new ColoredPoint(1, 2, Color.Red)
cp: ColoredPoint = ColoredPoint@5428bd62
```

```
scala> p equals cp
res9: Boolean = true

scala> cp equals p
res10: Boolean = false
```

Сравнение `p equals cp` вызывает метод `equals` для объекта `p` класса `Point`. Этот метод берет в расчет только координаты двух точек. Следовательно, сравнение выдает `true`. В то же время сравнение `cp equals p` вызывает метод `equals` для объекта `cp` класса `ColoredPoint`. Этот метод возвращает `false`, поскольку `p` не относится к типу `ColoredPoint`. Следовательно, отношение, определенное методом `equals`, несимметрично.

Утрата симметричности может привести к неожиданным последствиям для коллекций. Рассмотрим следующий пример:

```
scala> collection.mutable.HashSet[Point](p) contains cp
res11: Boolean = true

scala> collection.mutable.HashSet[Point](cp) contains p
res12: Boolean = false
```

Даже если `p` и `cp` равны, один объект проходит проверку на принадлежность к коллекции `contains` успешно, а второй — нет.

Как изменить определение `equals`, чтобы добиться симметричности? Есть два способа. Отношение можно либо превратить в более общее, либо сделать более строгим. Превращение означает, что пара объектов, `x` и `y`, считается равной, если сравнение `x` с `y` или `y` с `x` выдает `true`. Рассмотрим код, реализующий этот замысел:

```
class ColoredPoint(x: Int, y: Int, val color: Color.Value)
    extends Point(x, y) { // Проблема: нарушена транзитивность равенства

  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
      (this.color == that.color) && super.equals(that)
    case that: Point =>
      that equals this
    case _ =>
      false
  }
}
```

В новом определении `equals` в `ColoredPoint` на один `case`-вариант больше, чем в старом: если другой объект относится к `Point`, а не к `ColoredPoint`, то метод передает управление методу `equals` класса `Point`. Тем самым достигается желаемый эффект симметричности `equals`. Теперь оба выражения — и `cp equals p`, и `p equals cp` — выдают результат `true`. Но соглашение относительно `equals` по-прежнему нарушено. Дело в том, что новое отношение больше не обладает транзитивностью!

Вот как выглядит последовательность инструкций, демонстрирующая это. Определите простую точку и две точки разного цвета так, чтобы все они были в одной и той же позиции:

```
scala> val redp = new ColoredPoint(1, 2, Color.Red)
redp: ColoredPoint = ColoredPoint@5428bd62

scala> val bluep = new ColoredPoint(1, 2, Color.Blue)
bluep: ColoredPoint = ColoredPoint@5428bd62
```

По отдельности `redp` равна `p`, а `p` равна `bluep`:

```
scala> redp == p
res13: Boolean = true

scala> p == bluep
res14: Boolean = true
```

А вот сравнение `redp` и `bluep` выдает `false`:

```
scala> redp == bluep
res15: Boolean = false
```

Следовательно, условие транзитивности в соглашении по `equals` нарушено.

Похоже, если сделать отношения равенства более общими, то это заведет нас в тупик. Вместо этого попробуем сделать их более строгими. Один из способов повышения строгости достигается неизменным рассмотрением объектов разных классов в качестве разных объектов. Добиться этого можно за счет изменения методов `equals` в классах `Point` и `ColoredPoint`. В классе `Point` можно добавить еще одно сравнение, проверяющее, что класс во время выполнения программы другого `Point`-объекта такой же, как и класс данного `Point`-объекта:

```
// Технически состоятельный, но не удовлетворяющий нас метод equals
class Point(val x: Int, val y: Int) {
  override def hashCode = (x, y).##
  override def equals(other: Any) = other match {
    case that: Point =>
      this.x == that.x && this.y == that.y &&
      this.getClass == that.getClass
    case _ => false
  }
}
```

Затем можно вернуть реализацию класса `ColoredPoint` назад — к той версии, которая ранее была забракована из-за несоблюдения требований симметричности[1]:

```
class ColoredPoint(x: Int, y: Int, val color: Color.Value)
    extends Point(x, y) {

  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
      (this.color == that.color) && super.equals(that)
    case _ => false
  }
}
```

[1] Благодаря новой версии equals в Point эта версия ColoredPoint больше не нарушает требование симметричности.

Здесь экземпляр класса Point считается равным какому-либо другому экземпляру того же класса лишь при условии, что у объекта имеются такие же координаты и одинаковый класс во время выполнения, что говорит о возвращении методом getClass в отношении обоих объектов одного и того же значения. Новое определение отвечает требованиям симметричности и транзитивности, поскольку теперь каждое сравнение объектов разных классов выдает false. Следовательно, цветная точка никогда не будет равна простой точке. Это соглашение представляется вполне резонным, но кое-кто может и оспорить излишнюю строгость нового определения.

Рассмотрим следующий непрямой способ определения точки с координатами (1, 2):

```
scala> val pAnon = new Point(1, 1) { override val y = 2 }
pAnon: Point = $anon$1@5428bd62
```

Будет ли объект pAnon равен объекту p? Нет, поскольку объекты java.lang.Class, ассоциируемые с p и pAnon, разные. Для p это Point, а для pAnon — анонимный подкласс Point. Но вполне очевидно, что pAnon является всего лишь еще одной точкой с координатами (1, 2). Неразумно считать ее отличной от p.

Похоже, мы зашли в тупик. Существует ли разумный способ переопределить равенство на нескольких уровнях иерархии классов, соблюдая все условия соглашения? Фактически такой способ существует, но для него нужен еще один метод, переопределяемый вместе с equals и hashCode. Идея такова: как только в классе переопределяется equals (и hashCode), следует также прямо указать, что объекты данного класса не равны объектам какого-либо суперкласса, в котором реализован другой метод равенства. Это достигается добавлением к каждому классу, в котором переопределен equals, метода canEqual. Данный метод имеет следующую сигнатуру:

```
def canEqual(other: Any): Boolean
```

Метод должен вернуть либо true, если объект other является экземпляром класса, в котором определен (или переопределен) метод canEqual, либо false в противном случае. Его вызывают из equals с целью убедиться, что объекты можно сравнивать в обоих направлениях. Новое (и окончательное) определение класса Point, придерживающееся этого принципа, показано в листинге 30.1.

Листинг 30.1. Метод equals суперкласса, вызывающий метод canEqual

```
class Point(val x: Int, val y: Int) {
  override def hashCode = (x, y).##
  override def equals(other: Any) = other match {
    case that: Point =>
      (that canEqual this) &&
      (this.x == that.x) && (this.y == that.y)
    case _ =>
     false
  }
  def canEqual(other: Any) = other.isInstanceOf[Point]
}
```

В этой версии метода `equals` класса `Point` содержится дополнительное требование: другой объект *может быть равен* данному объекту согласно определению в методе `canEqual`. В реализации `canEqual` в классе `Point` утверждается, что равными могут быть все экземпляры класса `Point`.

В листинге 30.2 показана соответствующая реализация `ColoredPoint`. Можно показать, что новые определения `Point` и `ColoredPoint` придерживаются соглашения по `equals`. Равенство симметрично и транзитивно. Сравнение экземпляров `Point` с экземплярами `ColoredPoint` всегда выдает `false`. Разумеется, для любой точки `p` и цветной точки `cp` при вычислении выражения `p equals cp` будет возвращено значение `false`, поскольку при вычислении `cp canEqual p` будет возвращено значение `false`. Обратное сравнение, `cp equals p`, тоже приведет к возвращению `false`, так как `p` не относится к типу `ColoredPoint`, следовательно, первое сопоставление с образцом в теле `equals` в классе `ColoredPoint` завершится неудачей.

Листинг 30.2. Метод equals подкласса, вызывающий метод canEqual

```
class ColoredPoint(x: Int, y: Int, val color: Color.Value)
    extends Point(x, y) {

  override def hashCode = (super.hashCode, color).##
  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
      (that canEqual this) &&
      super.equals(that) && this.color == that.color
    case _ =>
      false
  }
  override def canEqual(other: Any) =
    other.isInstanceOf[ColoredPoint]
}
```

В то же время экземпляры различных подклассов `Point` могут быть равны при условии, что ни в одном из них метод равенства не был переопределен. Так, после определения новых классов сравнение `p` и `pAnon` выдаст `true`. Рассмотрим несколько примеров:

```
scala> val p = new Point(1, 2)
p: Point = Point@5428bd62

scala> val cp = new ColoredPoint(1, 2, Color.Indigo)
cp: ColoredPoint = ColoredPoint@e6230d8f

scala> val pAnon = new Point(1, 1) { override val y = 2 }
pAnon: Point = $anon$1@5428bd62

scala> val coll = List(p)
coll: List[Point] = List(Point@5428bd62)
```

```
scala> coll contains p
res16: Boolean = true

scala> coll contains cp
res17: Boolean = false

scala> coll contains pAnon
res18: Boolean = true
```

В этих примерах показано, что если реализация `equals` в суперклассе определена и в ней вызывается метод `canEqual`, то программисты, реализующие подклассы, имеют возможность решать, могут ли их подклассы быть равны экземплярам суперкласса. К примеру, в `ColoredPoint` метод `canEqual` переопределяется, поэтому цветная точка никогда не может быть равна обычной. Но поскольку в анонимном подклассе, на который ссылается `pAnon`, метод `canEqual` не переопределяется, то его экземпляр может быть равен экземпляру класса `Point`.

Подход с применением `canEqual` можно критиковать за нарушение принципа подстановки Лисков (LSP). В качестве примера нарушения LSP можно привести технологию реализации `equals` путем сравнения классов во время выполнения программы, приводящей к невозможности определить подкласс, экземпляры которого могут быть равны экземплярам суперкласса[1]. Согласно положениям LSP должна существовать возможность использования (подстановки) экземпляра подкласса там, где требуется экземпляр суперкласса.

Но в предыдущем примере `coll contains cp` выдает `false`, несмотря на то что имеющиеся у `cp` значения `x` и `y` соответствуют таким же значениям точки в коллекции. Таким образом, это может быть похоже на нарушение LSP, поскольку вы не можете использовать `ColoredPoint` там, где ожидается `Point`. Полагаем, что это неверная интерпретация: в LSP не требуется, чтобы подкласс вел себя точно так же, как его суперкласс, просто он ведет себя в соответствии с соглашениями, принятыми в отношении своего суперкласса.

Проблема, касающаяся написания метода `equals`, который сравнивает классы во время выполнения программы, не в том, что нарушается LSP, а в том, что вы не получаете способа создания подкласса, чьи экземпляры могут быть равны экземплярам суперкласса. Например, если бы в предыдущем примере мы воспользовались технологией класса времени выполнения программы, то выражение `coll contains pAnon` выдало бы `false`, что не соответствовало бы нашему желанию. В отличие от этого нам хотелось, чтобы выражение `coll contains cp` выдало `false`, поскольку путем переопределения `equals` в `ColoredPoint` мы в основном добивались, чтобы фиолетовая точка с координатами (1, 2) *не была эквивалентна* бесцветной точке с координатами (1, 2). Следовательно, в предыдущем примере у нас была возможность передать имеющемуся в коллекции методу `contains` два различных экземпляра подкласса `Point` и получить обратно два разных ответа, оба правильные.

---

[1] *Блох Дж.* Java. Эффективное программирование. 3-е изд. — М.: Вильямс, 2018.

30.3. Определение равенства для параметризованных типов

Все методы `equals` в предыдущих примерах начинались с сопоставления с образцом, с помощью которого проверялось, подходит ли тип операнда к типу класса, содержащего метод `equals`. При параметризации классов эта схема нуждается в некоторой адаптации.

Рассмотрим в качестве примера двоичные деревья. В иерархии классов, показанной в листинге 30.3, для двоичного дерева определяется абстрактный класс `Tree` с двумя альтернативными реализациями: объектом `EmptyTree` и классом `Branch`, представляющим непустые деревья. Непустое дерево составляется из элемента `elem` и из левого и правого дочерних деревьев. Тип его элемента задается параметром типа `T`.

Листинг 30.3. Иерархия для двоичных деревьев

```
trait Tree[+T] {
  def elem: T
  def left: Tree[T]
  def right: Tree[T]
}

object EmptyTree extends Tree[Nothing] {
  def elem =
    throw new NoSuchElementException("EmptyTree.elem")
  def left =
    throw new NoSuchElementException("EmptyTree.left")
  def right =
    throw new NoSuchElementException("EmptyTree.right")
}

class Branch[+T](
  val elem: T,
  val left: Tree[T],
  val right: Tree[T]
) extends Tree[T]
```

Теперь добавим к этим классам методы `equals` и `hashCode`. Для самого класса `Tree` делать ничего не нужно, поскольку предполагается, что реализация этих методов выполняется отдельно для каждой реализации абстрактного класса. Для объекта `EmptyTree` тоже ничего делать не нужно, так как исходная реализация `equals` и `hashCode`, наследуемая классом `EmptyTree` из `AnyRef`, сохраняет свою работоспособность. Помимо всего прочего, объект `EmptyTree` равен только самому себе, поэтому равенство должно быть равенством ссылок, а оно унаследовано из `AnyRef`.

А вот над добавлением `equals` и `hashCode` к `Branch` придется потрудиться. Значение типа `Branch` должно быть равно лишь другим `Branch`-значениям и только если

у двух значений есть равные поля `elem`, `left` и `right`. Вполне естественно будет применить схему для `equals`, которая была разработана в предыдущих разделах этой главы. Тогда получится следующий код:

```
class Branch[T](
  val elem: T,
  val left: Tree[T],
  val right: Tree[T]
) extends Tree[T] {

  override def equals(other: Any) = other match {
    case that: Branch[T] => this.elem == that.elem &&
                            this.left == that.left &&
                            this.right == that.right
    case _ => false
  }
}
```

Но при компиляции этого примера выдается `unchecked`-предупреждение. Повторная компиляция с ключом `-unchecked` выявляет следующую проблему:

```
 $ fsc -unchecked Tree.scala
Tree.scala:14: warning: non variable type-argument T in type
pattern is unchecked since it is eliminated by erasure
    case that: Branch[T] => this.elem == that.elem &&
               ^
```

В предупреждении говорится, что имеется сопоставление с образцом в отношении типа `Branch[T]`, а система может лишь проверить, что другой ссылкой выступает некая разновидность `Branch` — но не может проверить, что типом элемента дерева является `T`. Причины этого мы уже объясняли в главе 19: типы элементов параметризованных типов уничтожаются на этапе затирания, проводимого компилятором, проверить их во время выполнения программы невозможно.

Что же можно сделать? К счастью, оказывается, что проверять при сравнении двух `Branch`-объектов наличие у них одинаковых типов элементов совсем не обязательно. Вполне возможно, что два `Branch`-объекта с разными типами элементов будут равны при условии одинаковости их полей. Простым примером этому может послужить `Branch`-объект, состоящий из одного `Nil`-элемента и двух пустых поддеревьев. Вероятно, любые такие `Branch`-объекты можно считать равными независимо от имеющегося у них статического типа:

```
scala> val b1 = new Branch[List[String]](Nil,
           EmptyTree, EmptyTree)
b1: Branch[List[String]] = Branch@9d5fa4f

scala> val b2 = new Branch[List[Int]](Nil,
           EmptyTree, EmptyTree)
b2: Branch[List[Int]] = Branch@56cdfc29

scala> b1 == b2
res19: Boolean = true
```

Положительный результат данного сравнения был получен с помощью показанной ранее реализации `equals` в `Branch`. Тем самым показано, что тип элемента `Branch`-объекта не был проверен. В противном случае результатом было бы значение `false`.

Можно поспорить о том, какой из двух возможных исходов сравнения будет более естественным. В конечном счете это зависит от задуманной модели представления классов. В той модели, где параметры типа присутствуют только во время компиляции, вполне естественно рассматривать два `Branch`-значения, `b1` и `b2`, в качестве равных. В альтернативной модели, где параметр типа формирует часть значения объекта, так же естественно будет считать их различными. В Scala принята модель затирания типа, поэтому параметры типа во время выполнения программы отсутствуют, так что `b1` и `b2` вполне естественно считать равными.

Нужно всего лишь внести маленькое изменение в формулировку метода `equals`, в результате чего `unchecked`-предупреждение выдаваться не будет. Вместо обозначения элемента типа `T` воспользуйтесь буквой в нижнем регистре, например `t`:

```
case that: Branch[t] => this.elem == that.elem &&
                        this.left == that.left &&
                        this.right == that.right
```

Вспомним: в разделе 15.2 говорилось, что параметр типа в паттерне, начинающийся с буквы в нижнем регистре, представляет неизвестный тип. Теперь сопоставление с образцом вида:

```
case that: Branch[t] =>
```

будет выполняться успешно для `Branch`-значений любого типа. Параметр типа `t` представляет неизвестный тип элемента `Branch`. Его можно заменить и знаком подчеркивания, как в следующем примере, эквивалентном предыдущему:

```
case that: Branch[_] =>
```

Теперь осталось лишь определить для класса `Branch` два других метода, `hashCode` и `canEqual`, которые сопутствуют методу `equals`. Вот так выглядит возможная реализация метода `hashCode`:

```
override def hashCode: Int = (elem, left, right).##
```

Это лишь одна из множества возможных реализаций. Как было показано ранее, принцип заключается в получении `hashCode`-значений всех полей и в их объединении. А вот так выглядит в классе `Branch` реализация метода `canEqual`:

```
def canEqual(other: Any) = other match {
  case that: Branch[_] => true
  case _ => false
}
```

В реализации метода `canEqual` используется типизированное сопоставление с образцом. Можно также сформулировать код метода с помощью `isInstanceOf`:

```
def canEqual(other: Any) = other.isInstanceOf[Branch[_]]
```

Если вы склонны критически оценивать все здесь увиденное, к чему мы вас, собственно, и подталкиваем, то у вас может появиться вопрос о предназначении знака подчеркивания в показанном ранее типе. Ведь в конце концов `Branch[_]` с технической точки зрения является не типом паттерна, а принадлежащим методу типом параметра. Тогда как же можно оставить некоторые его части неопределенными?

Ответ на данный вопрос будет рассматриваться в следующей главе. `Branch[_]` — сокращенная форма записи так называемого *подстановочного типа* (wildcard type), который, если не вдаваться в подробности, является типом с неизвестными частями. Поэтому, даже притом, что технически знак подчеркивания имеет два предназначения в сопоставлении с образцом и в параметре типа, используемом в коде вызова метода, по сути, предназначение одно и то же: знак позволяет пометить нечто неизвестное. Окончательная версия `Branch` показана в листинге 30.4.

Листинг 30.4. Параметризованные типы с equals и hashCode

```
class Branch[T](
  val elem: T,
  val left: Tree[T],
  val right: Tree[T]
) extends Tree[T] {

  override def equals(other: Any) = other match {
    case that: Branch[_] => (that canEqual this) &&
                            this.elem == that.elem &&
                            this.left == that.left &&
                            this.right == that.right
    case _ => false
  }

  def canEqual(other: Any) = other.isInstanceOf[Branch[_]]

  override def hashCode: Int = (elem, left, right).##
}
```

30.4. Рецепты для equals и hashCode

В этом разделе мы представим пошаговые рецепты создания методов `equals` и `hashCode`, которых должно быть достаточно для большинства ситуаций. В качестве иллюстрации воспользуемся методами класса `Rational`, показанного в листинге 30.5.

Листинг 30.5. Класс Rational с equals и hashCode

```
class Rational(n: Int, d: Int) {

  require(d != 0)
```

```
  private val g = gcd(n.abs, d.abs)
  val numer = (if (d < 0) -n else n) / g
  val denom = d.abs / g

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  override def equals(other: Any): Boolean =
    other match {

      case that: Rational =>
        (that canEqual this) &&
        numer == that.numer &&
        denom == that.denom
     case _ => false
   }

  def canEqual(other: Any): Boolean =
    other.isInstanceOf[Rational]

  override def hashCode: Int = (numer, denom).##

  override def toString =
    if (denom == 1) numer.toString else s"$numer/$denom"
}
```

Создадим этот класс, удалив методы математических операторов из версии класса `Rational`, показанной в листинге 6.5. Внесем также небольшие усовершенствования в метод `toString` и изменим инициализаторы `numer` и `denom` для нормализации всех дробей, чтобы у них был положительный знаменатель (то есть чтобы преобразовать 1/–2 в –1/2).

Рецепт для equals

Рассмотрим рецепт для переопределения `equals`.

1. Для переопределения `equals` в не `final`-классе создайте метод `canEqual`. В определении `equals`, унаследованном из `AnyRef` (то есть в `equals`, не переопределенном где-нибудь выше в иерархии классов), определение `canEqual` должно обновиться, иначе будет переопределено определение метода с тем же именем. Единственное исключение из этого требования касается `final`-классов, в которых переопределяется метод `equals`, унаследованный из `AnyRef`. Для них не могут возникать аномалии подклассов, рассмотренные в разделе 30.2, следовательно, в них не нужно определять метод `canEqual`. Типом объекта, переданного `canEqual`, должен быть `Any`:

   ```
   def canEqual(other: Any): Boolean =
   ```

2. Метод `canEqual` должен выдавать `true`, если объект-аргумент является экземпляром текущего класса (то есть класса, в котором определен `canEqual`), в противном случае он должен выдавать `false`:

   ```
   other.isInstanceOf[Rational]
   ```

3. В методе `equals` нужно удостовериться, что типом передаваемого объекта объявлен `Any`:

   ```
   override def equals(other: Any): Boolean =
   ```

4. Тело метода `equals` следует записать в виде одного выражения `match`. Селектором `match` должен быть объект, переданный `equals`:

   ```
   other match {
     // ...
   }
   ```

5. У выражения `match` должно быть два варианта. В первом должен объявляться типизированный паттерн для типа класса, в котором определяется метод `equals`:

   ```
   case that: Rational =>
   ```

6. В теле этого варианта следует записать выражение, объединяющее с помощью логического «И» отдельные выражения, которые должны вычисляться в `true` для объектов, которые должны быть равными. Если переопределяемый метод `equals` отличается от метода в `AnyRef`, то, скорее всего, возникнет желание включить вызов метода `equals` из суперкласса:

   ```
   super.equals(that) &&
   ```

   Если определяется метод `equals` для класса, в котором впервые появляется метод `canEqual`, то нужно вызвать `canEqual` в отношении аргумента метода равенства, передав в качестве аргумента ключевое слово `this`:

   ```
   (that canEqual this) &&
   ```

   В переопределении `equals` следует также включать вызов `canEqual`, если только в них не содержится вызов `super.equals`. В последнем варианте проверка с помощью `canEqual` уже должна выполняться путем вызова суперкласса. И наконец, для каждого поля, имеющего отношение к равенству, следует проверить, что поле в объекте `this` равно соответствующему полю в переданном объекте:

   ```
   numer == that.numer &&
   denom == that.denom
   ```

7. Для второго варианта следует воспользоваться подстановочным паттерном, выдающим `false`:

   ```
   case _ => false
   ```

Если придерживаться этого рецепта для `equals`, то равенство гарантированно будет отношением эквивалентности в соответствии с требованием соглашения по `equals`.

Рецепт для hashCode

Обычно удовлетворительные результаты для `hashCode` достигаются при использовании следующего рецепта, похожего на рецепты, рекомендованные для классов Java в книге Effective Java[1]. Возьмите в расчет все поля в своем объекте, которые служат для определения равенства в методе `equals` (значимые поля). Создайте кортеж, содержащий значения всех этих полей. Затем вызовите для получившегося кортежа метод `##`.

Например, чтобы реализовать хеш-код для объекта, имеющего пять значимых полей с именами `a`, `b`, `c`, `d` и `e`, нужно сделать следующую запись:

```
override def hashCode: Int = (a, b, c, d, e).##
```

Если метод `equals` в качестве части своих вычислений вызывает `super.equals(that)`, то вычисление в вашем методе `hashCode` должно начинаться с вызова `super.hashCode`. Например, если в методе `equals` класса `Rational` вызывается `super.equals(that)`, то его метод `hashCode` должен иметь следующий вид:

```
override def hashCode: Int = (super.hashCode, numer, denom).##
```

При написании методов `hashCode` с использованием данного подхода следует иметь в виду, что ваш хеш-код будет не хуже хеш-кодов, создаваемых за его пределами, а именно тех, которые получаются путем вызова `hashCode` в отношении значимых полей вашего объекта. Иногда для получения приемлемого хеш-кода для такого поля одного только вызова `hashCode` в отношении этого поля может оказаться недостаточно. Например, если одно из полей является коллекцией, то, скорее всего, для него понадобится хеш-код, основанный на всех содержащихся в коллекции элементах. Если поле относится к типу `Vector`, `List`, `Set`, `Map` или является кортежем, то его можно просто включить в список хешируемых элементов, поскольку `equals` и `hashCode` переопределены в данных классах таким образом, чтобы брать в расчет содержащиеся в них элементы. Этого нельзя сказать о полях, относящихся к типу `Array`, в которых при вычислении хеш-кода элементы в расчет не берутся. Что касается массивов, то здесь нужно рассматривать каждый элемент массива как отдельное поле вашего объекта, вызывая оператор `##` в отношении каждого элемента в явном виде или передавая массив одному из методов `hashCode` в объекте-одиночке `java.util.Arrays`.

И наконец, если обнаружится, что конкретно взятое вычисление хеш-кода отрицательно влияет на производительность вашей программы, то рассмотрите вариант кэширования хеш-кода. Если объект неизменяемый, то можно вычислить хеш-код при создании объекта и сохранить его в поле. Сделать это можно с помощью простого переопределения `hashCode`, указав `val` вместо `def`:

```
override val hashCode: Int = (numer, denom).##
```

Этот подход связан с дополнительным расходом памяти в процессе вычисления, поскольку каждый экземпляр неизменяемого класса будет иметь на одно поле больше для сохранения кэшируемого значения хеш-кода.

---

[1] *Блох Дж.* Java. Эффективное программирование. 3-е изд. — М.: Вильямс, 2018.

Резюме

Оглядываясь назад, можно сказать, что правильная реализация `equals` оказалась на удивление непростым занятием. Нужно было проявить осмотрительность в отношении сигнатуры типа, переопределить `hashCode`, обойти зависимости от изменяемого состояния и, если класс был не `final`, реализовать и использовать метод `canEqual`.

Столкнувшись со сложностями реализации корректного метода равенства, можно отдать предпочтение определению классов сопоставимых объектов в виде `case`-классов. Тогда компилятор Scala автоматически добавит методы `equals` и `hashCode`, имеющие нужные свойства.

31 Сочетание кода на Scala и Java

Зачастую код Scala используется в тандеме с довольно объемными программами и фреймворками на Java. Scala обладает высокой степенью совместимости с Java, поэтому в большинстве случаев удается вполне успешно без особой оглядки сочетать два языка. Например, доподлинно известно, что со Scala вполне успешно работают такие стандартные фреймворки, как Swing, Servlets и JUnit. И тем не менее иногда при сочетании Java и Scala можно столкнуться с некоторыми проблемами.

В этой главе мы опишем два аспекта сочетания Java и Scala. Сначала рассмотрим вопрос перевода Scala в Java, приобретающий особую важность, если код Scala вызывается из Java. Затем обсудим применение аннотаций Java в Scala, играющее важную роль при необходимости использования Scala с существующими фреймворками Java.

31.1. Использование Scala из Java

В основном Scala можно рассматривать на уровне исходного кода. Но более полное представление о работе системы можно получить, разобравшись с трансляцией ее кода. Более того, если код Scala вызывается из Java, то следует знать, как именно код Scala выглядит с точки зрения Java.

Основные правила

Scala реализуется в виде трансляции в стандартные байт-коды Java. Насколько это возможно, средства Scala отображаются напрямую на эквивалентные им средства Java. Например, классы, методы, строки и исключения Scala компилируются в тот же самый байт-код Java, что и их Java-аналоги.

Для этого иногда требовалось принимать трудные решения относительно дизайна Scala. Например, было бы неплохо реализовывать разрешение перегруженных методов во время выполнения программы, а не во время ее компиляции. Но такая

конструкция нарушала бы положения, принятые в Java, существенно усложняя смешивание кода Java и Scala. В данном случае Scala придерживается разрешения перегрузки, принятого в Java, благодаря чему методы Scala и вызовы методов могут напрямую отображаться на методы и вызовы методов Java.

Для других свойств в Scala есть собственный дизайн. Например, эквивалента трейтам в Java не существует. Аналогично, несмотря на то что обобщенные типы есть как в Scala, так и в Java, их особенности в этих системах сильно разнятся. Для подобных свойств языка код Scala не может быть напрямую отображен на конструкции Java, следовательно, должен быть преобразован с помощью каких-либо комбинаций имеющихся в Java структур.

Для таких косвенно отображаемых свойств преобразование не имеет фиксированных форм. Максимальное упрощение подобных преобразований требует постоянных усилий, поэтому на момент чтения данной книги некоторые особенности, существовавшие во времена ее написания, могли измениться. Какие именно преобразования использует ваш компилятор Scala, можно определить, изучив файлы с расширением `.class` с помощью инструментального средства, подобного `javap`.

Таковы общие правила. А теперь рассмотрим некоторые особые случаи.

Типы значений

Такой тип значения, как `Int`, может быть преобразован в Java двумя различными способами. При первой же возможности для получения наивысшей производительности компилятор преобразует Scala `Int` в Java `int`. Но иногда сделать это не удается, поскольку компилятор не уверен, каким именно преобразованием он занимается, `Int` или какого-либо другого типа данных. Например, конкретный `Array[Any]` может содержать только `Int`-значения, но у компилятора нет возможности убедиться в этом.

Когда компилятор не уверен, относится объект к типу значений или нет, он использует объекты и полагается на классы-оболочки. Например, такие классы-оболочки, как `java.lang.Integer`, позволяют типу значений быть заключенным в Java-объект, в силу чего обрабатываться кодом, требующим объекты[1].

Объекты-одиночки

В Java нет точного эквивалента объекту-одиночке, но в нем имеются статические методы. В выполняемом Scala преобразовании объектов-одиночек используется сочетание статических методов и методов экземпляра. Для каждого объекта-одиночки Scala компилятор создаст класс Java с именем, соответствующим имени объекта со знаком доллара в конце. Например, для объекта-одиночки по имени `App`

[1] Реализацию типов значений мы подробно рассмотрели в разделе 11.2.

компилятор создаст Java-класс по имени App$. Он будет иметь все методы и поля объекта-одиночки Scala. В Java-классе также будет одно статическое поле по имени MODULE$, предназначенное для хранения одного экземпляра класса, создаваемого во время выполнения программы.

Чтобы привести полноценный пример, предположим, что компилируется следующий объект-одиночка:

```
object App {
  def main(args: Array[String]) = {
    println("Hello, world!")
  }
}
```

Scala создаст Java-класс App$ со следующими полями и методами:

```
$ javap App$
Compiled from "App.scala"
public final class App$ {
    public static final App$ MODULE$;
    public static {};
    public void main(java.lang.String[]);
}
```

Это общий вариант преобразования. Важный особый случай — наличие «обособленного» объекта-одиночки, для которого нет класса с таким же именем. Например, у вас может быть объект-одиночка по имени App и не быть класса с именем App. В таком случае компилятор создаст Java-класс по имени App, имеющий статический метод перенаправления к каждому методу объекта-одиночки Scala:

```
$ javap App
Compiled from "App.scala"
public class App {
    public static void main(java.lang.String[]);
    public App();
}
```

В отличие от этого, если имелся класс по имени App, то Scala создаст соответствующий Java-класс App для хранения членов определенного вами класса App. В этот класс не будут добавляться методы перенаправления для одноименного объекта-одиночки, и код Java должен будет обращаться к объекту-одиночке с помощью поля MODULE$.

Трейты в качестве интерфейсов

При компиляции любого трейта создается Java-интерфейс с таким же именем. Этот интерфейс можно использовать как тип Java, и он позволяет вам вызывать методы в отношении объектов Scala через переменные этого типа.

Реализация трейта в Java — совсем другая история. Вообще-то это не практикуется, но один особый случай все же имеет важное значение. Если создается

Scala-трейт, включающий только абстрактные методы, то будет напрямую преобразован в Java-интерфейс без какого-либо кода, за который стоило бы переживать. По сути, это значит, что при желании можно создавать интерфейс Java в синтаксисе Scala.

31.2. Аннотации

Основную систему аннотаций Scala мы рассмотрели в главе 27. В этом разделе изучим те аспекты аннотаций, которые имеют непосредственное отношение к Java.

Дополнительные эффекты от стандартных аннотаций

Некоторые аннотации заставляют компилятор выдавать дополнительную информацию с прицелом на платформу Java. Когда компилятору встречается такая аннотация, он сначала обрабатывает ее в соответствии с общими правилами, действующими в Scala, а затем выполняет дополнительные действия для Java.

- **Устаревание.** Для любого метода или класса с меткой `@deprecated` компилятор добавит к выдаваемому коду Java аннотацию устаревания. Поэтому компиляторы Java могут выдавать предупреждения об устаревании при обращении кода на Java к устаревшим методам Scala.
- **Непостоянные поля.** По аналогии с предыдущим описанием любое поле с меткой `@volatile` в коде Scala получает в выдаваемом коде Java-модификатор `volatile`. Благодаря этому непостоянные поля в Scala ведут себя в точном соответствии с семантикой Java, и обращение к непостоянным полям выполняется в последовательности, точно соответствующей правилам, определенным для `volatile`-полей в модели памяти Java.
- **Сериализация.** Оба имеющихся в Scala стандарта аннотирования сериализации преобразуются в их Java-эквиваленты. К классу с меткой `@serializable` добавляется Java-интерфейс `Serializable`. Аннотация `@SerialVersionUID(1234L)` превращается в следующее определение поля Java:

  ```
  // Используемый в Java маркер серийного номера версии
  private final static long SerialVersionUID = 1234L
  ```

  Любая переменная с меткой `@transient` получает Java-модификатор `transient`.

Сгенерированные исключения

В Scala перехват сгенерированных исключений не контролируется. Поэтому в нем нет эквивалента имеющемуся в Java объявлению методов с ключевым словом

`throws`. Все методы Scala переводятся в методы Java, в которых не объявляется генерация исключений[1].

Последнее свойство было убрано из Scala потому, что опыт работы с ним, накопленный в Java, имеет несколько негативный оттенок. Аннотирование методов с помощью `throws` вызывает трудности, поэтому многие разработчики создают код, подавляющий и сбрасывающий исключения, просто чтобы получить код для компиляции без добавления всех этих `throws`-описаний. Разработчики могут хотеть улучшить систему обработки исключений чуть позже, однако опыт показывает, что ни один из зачастую ограниченных во времени программистов никогда не вернется назад и не добавит соответствующую обработку исключений. Обратная сторона медали выглядит так, что это реализуемое с благими намерениями свойство зачастую снижает надежность кода. Огромная масса промышленного кода Java подавляет и скрывает генерацию исключений во время выполнения программ, чтобы удовлетворить компилятор.

Но иногда при разработке интерфейса к Java может потребоваться создать код Scala, имеющий дружественные по отношению к Java аннотации с описаниями того, какие именно исключения могут выдавать ваши методы. К примеру, каждый метод в удаленном интерфейсе RMI требует в его описании `throws` упоминания о `java.io.RemoteException`. То есть, если нужно создать удаленный интерфейс RMI в виде Scala-трейта с абстрактными методами, для этих методов нужно будет в описании `throws` упомянуть `RemoteException`. Для этого понадобится всего лишь пометить свои методы аннотацией `@throws`. Например, в Scala-классе, показанном в листинге 31.1, есть метод с меткой, сообщающей о том, что он выдает исключение `IOException`.

Листинг 31.1. Метод Scala, объявляющий Java-описание throws

```
import java.io._
class Reader(fname: String) {
  private val in =
    new BufferedReader(new FileReader(fname))

  @throws(classOf[IOException])
  def read() = in.read()
}
```

Вот как это выглядит в случае с Java:

```
$ javap Reader
Compiled from "Reader.scala"
public class Reader {
  public int read() throws java.io.IOException;
  public Reader(java.lang.String);
}
```

Обратите внимание: метод `read` показывает наряду с Java-объявлением `throws`, что может быть выдано исключение `IOException`.

---

[1] Код все равно работает, поскольку верификатор байт-кода Java вообще не проверяет объявления! Этим занимается компилятор Java.

Аннотации Java

Аннотации, имеющиеся во фреймворках Java, могут использоваться в коде Scala напрямую. Они будут видны в любом Java-фреймворке точно так же, как будто были написаны в коде Java.

Аннотации используются множеством различных пакетов Java. Рассмотрим в качестве примера JUnit 4. JUnit — фреймворк для написания и запуска автоматизированных тестов. Самая последняя версия, JUnit 4, использует аннотации, чтобы показать, какие части кода тестируются. Замысел в том, что вы написали для своего кода множество тестов, которые запускаются после каждого изменения исходного кода. Поэтому, если при изменении допущена новая ошибка, то один из тестов не будет пройден и ошибка тут же обнаружится.

Написать тест не составляет труда. Просто в классе верхнего уровня создается метод, выполняющий ваш код, а для того, чтобы обозначить его как тест, используется аннотация. Выглядит это следующим образом:

```
import org.junit.Test
import org.junit.Assert.assertEquals

class SetTest {

  @Test
  def testMultiAdd = {
    val set = Set() + 1 + 2 + 3 + 1 + 2 + 3
    assertEquals(3, set.size)
  }
}
```

Метод `testMultiAdd` — тест. Он добавляет несколько элементов во множество и убеждается в том, что каждый из них был добавлен только один раз. Метод `assertEquals`, получаемый в составе JUnit API, проверяет, что два его аргумента равны друг другу. Если они отличаются друг от друга, то тест не будет пройден. В таком случае он проверяет, что повторно добавляемые одинаковые числа не увеличили размер множества.

Тест помечается аннотацией `org.junit.Test`. Обратите внимание: эта аннотация была импортирована, следовательно, на нее можно сослаться просто как на `@Test`, не прибегая к более длинной ссылке `@org.junit.Test`.

Вот, собственно, и все. Тест можно запустить с помощью любого средства выполнения тестов JUnit. Здесь он запускается средством, работающим в командной строке:

```
$ scala -cp junit-4.3.1.jar:. org.junit.runner.JUnitCore SetTest
JUnit version 4.3.1
.
Time: 0.023

OK (1 test)
```

Написание собственных аннотаций

Чтобы создать аннотацию, которая будет видна Java-рефлексии, нужно воспользоваться принятой в Java системой записи и скомпилировать аннотацию с помощью `javac`. В данном случае вряд ли есть смысл записывать аннотации в Scala, поскольку стандартный компилятор их не поддерживает. Дело в том, что поддержка Scala никак не будет соответствовать полным возможностям аннотаций Java, а кроме того, когда-нибудь в Scala появится собственная рефлексия, и тогда захочется получить доступ к аннотациям Scala со Scala-рефлексией.

Пример аннотации:

```
import java.lang.annotation. * ; // Это код Java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Ignore { }
```

После компиляции показанного ранее кода с применением `javac` аннотацией можно будет воспользоваться следующим образом:

```
object Tests {
  @Ignore
  def testData = List(0, 1, -1, 5, -5)

  def test1 = {
    assert(testData == (testData.head :: testData.tail))
  }

  def test2 = {
    assert(testData.contains(testData.head))
  }
}
```

В этом примере предполагается, что `test1` и `test2` являются тестовыми методами, а `testData` должен игнорироваться, несмотря на то что его имя начинается с `test`.

Убедиться в присутствии данных аннотаций можно с помощью API рефлексии Java. Чтобы показать это в работе, можно воспользоваться следующим примером:

```
for {
  method <- Tests.getClass.getMethods
  if method.getName.startsWith("test")
  if method.getAnnotation(classOf[Ignore]) == null
} {
  println("found a test method: " + method)
}
```

Здесь рефлексивные методы `getClass` и `getMethods` служат для инспектирования всех полей класса вводимого объекта. Это обычные рефлексивные методы. В той части, которая относится непосредственно к аннотации, используется метод `getAnnotation`. Он есть у многих рефлексивных объектов и предназначен для

поиска аннотаций конкретного типа. В данном случае код ищет аннотацию нашего нового типа `Ignore`. Поскольку это API, успешное выполнение выявляется по возвращаемому результату, который может быть либо `null`, либо текущим объектом аннотации.

Рассмотрим работу этого кода:

```
$ javac Ignore.java
$ scalac Tests.scala
$ scalac FindTests.scala
$ scala FindTests
found a test method: public void Tests$.test2()
found a test method: public void Tests$.test1()
```

Попутно обратите внимание на то, что эти методы при просмотре в Java-рефлексии относятся к классу `Tests$`, а не `Tests`. В начале главы уже говорилось, что в Java реализация объекта-одиночки Scala помещается в Java-класс с добавлением в конец имени этого класса знака доллара. В данном случае реализация `Tests` попадает в Java-класс `Tests$`.

Следует понимать, что при использовании аннотаций Java придется работать в рамках существующих для них ограничений. Например, в аргументах к аннотациям можно задействовать только константы, а не выражения. Может поддерживаться `@serial(1234)`, но никак не `@serial(x * 2)`, поскольку `x * 2` не является константой.

31.3. Подстановочные типы

У всех типов Java есть Scala-эквивалент. Это нужно, чтобы код Scala мог обращаться к любому допустимому в Java классу. В большинстве случаев преобразование не вызывает никаких затруднений. `Pattern` в Java — это `Pattern` в Scala, а `Iterator<Component>` в Java — это `Iterator[Component]` в Scala. Но в некоторых случаях встречавшихся до сих пор типов Scala оказывается недостаточно. Что, к примеру, можно сделать с такими подстановочными типами Java (wildcard types), как `Iterator<?>` или `Iterator<? extends Component>`? А что можно сделать с такими необработанными типами, как `Iterator`, где параметр типа опущен? Для подстановочных типов Java и необработанных типов в Scala используется дополнительная разновидность типа, которая также называется *подстановочным типом*.

Подстановочные типы записываются с помощью *синтаксиса заместителя*, что очень похоже на краткую форму записи функционального литерала, рассмотренную в разделе 8.5. В сокращении для функциональных литералов вместо выражения можно использовать символ подчеркивания (`_`). Например, `(_ + 1)` — аналог `(x => x + 1)`. В подстановочных типах работает тот же принцип, но только для типов, а не для выражений. Если написать `Iterator[_]`, то знак подчеркивания замещает тип. Такой тип представляет собой `Iterator` с неизвестным типом элементов.

При использовании синтаксиса заместителей можно также вставлять нижние и верхние ограничители. Нужно просто добавить ограничитель после знака под-

черкивания, задействуя тот же синтаксис `<:`, который применялся для параметров типа (см. разделы 19.5 и 19.8). Например, тип `Iterator[_ <: Component]` является итератором с неизвестным типом элементов, но, каким бы ни был этот тип, он должен быть подтипом типа `Component`.

С записью подстановочного типа разобрались, а как насчет его использования? В простых случаях можно игнорировать подстановку и вызывать методы в отношении базового типа. Предположим, к примеру, что у вас есть следующий Java-класс:

```
// Это Java-класс с подстановками
import java.util.*;
public class Wild {
  public Collection<?> contents() {
    Collection<String> stuff = new Vector<String>();
    stuff.add("a");
    stuff.add("b");
    stuff.add("see");
    return stuff;
  }
}
```

Если обращаться к этому коду из кода Scala, то будет видно, что у него есть подстановочный тип:

```
scala> val contents = (new Wild).contents
contents: java.util.Collection[_] = [a, b, see]
```

Если нужно определить количество элементов в коллекции, то можно просто проигнорировать подстановочную часть и вызвать обычным способом метод `size`:

```
scala> contents.size()
res0: Int = 3
```

В более сложных случаях подстановочные типы могут создавать более серьезные проблемы. У подстановочного типа нет имени, поэтому нет и способа использовать его в двух разных местах. Предположим, к примеру, что нужно создать изменяемое Scala-множество и инициализировать его элементами `contents`:

```
import scala.collection.mutable
val iter = (new Wild).contents.iterator
val set = mutable.Set.empty[???]     // Какой тип сюда попадет?
while (iter.hasMore)
  set += iter.next()
```

Проблема возникает в третьей строке. В Java-коллекции назвать тип элементов невозможно, следовательно, нельзя удовлетворительным образом записать тип множества. Обойти данную проблему можно с помощью двух приемов, которые требуют рассмотрения.

1. При передаче подстановочного типа в метод задайте методу параметр для заместителя. Теперь у вас есть название типа, которым можно воспользоваться желаемое количество раз.

2. Вместо возвращения из метода подстановочного типа возвращайте объект, имеющий абстрактные элементы для каждого типа, обозначенного заместителем. (Сведения об абстрактных элементах можно найти в главе 20.)

Используя вместе оба приема, предыдущий код можно превратить в следующий:

```
import scala.collection.mutable
import java.util.Collection

abstract class SetAndType {
  type Elem
  val set: mutable.Set[Elem]
}

def javaSet2ScalaSet[T](jset: Collection[T]): SetAndType = {
  val sset = mutable.Set.empty[T] // Теперь T может быть назван!

  val iter = jset.iterator
  while (iter.hasNext)
    sset += iter.next()

  new SetAndType {
    type Elem = T
    val set = sset
  }
}
```

Можно понять, почему в коде Scala подстановочные типы обычно не используются. Чтобы не возиться с ними, многие склоняются к преобразованию этих типов в абстрактные члены. Поэтому для начала можно также применить абстрактные члены.

31.4. Совместная компиляция Scala и Java

Обычно при проведении компиляции кода Scala, который зависит от кода Java, сначала создается Java-код файлов класса. Затем создается код Scala, а файлы с классами в коде Java указываются в пути к классам. Но такой подход не будет работать, если в коде Java есть обратные ссылки на код Scala. В таком случае порядок, в котором компилируется код, не играет никакой роли — либо та, либо другая сторона будет иметь неудовлетворенные внешние ссылки. Подобная ситуация не редкость, она возникает главным образом в Java-проекте, где какой-нибудь файл исходного кода на Java заменяется файлом исходного кода на Scala.

Для поддержки подобных сборок Scala позволяет компилировать исходный код Java так же, как и файлы классов Java. Нужно лишь указать файлы с исходным кодом на Java в командной строке, словно это файлы Scala. Компилятор Scala не станет их компилировать, но просканирует с целью понять их содержимое.

Чтобы применить это средство, сначала скомпилируйте Scala-код с помощью файлов исходного кода на Java, а затем Java-код с использованием файлов классов Scala.

А вот как выглядит типичная последовательность команд:

```
$ scalac -d bin InventoryAnalysis.scala InventoryItem.java \
      Inventory.java
$ javac -cp bin -d bin Inventory.java InventoryItem.java \
      InventoryManagement.java
$ scala -cp bin InventoryManagement
Most expensive item = sprocket($4.99)
```

31.5. Интеграция Java 8

В версии Java 8 в язык Java и в байт-коды был добавлен ряд усовершенствований, которыми пользуется Scala, начиная с версии 2.12[1]. С помощью новых возможностей Java 8 компилятор Scala может создавать более компактные файлы классов и jar-файлы и улучшить двоичную совместимость трейтов.

Лямбда-выражения и SAM-типы

С точки зрения программистов, работающих на Scala, наиболее ощутимые усовершенствования в Scala 2.12, связанные с применением Java 8, заключаются в том, что функциональные литералы Scala могут использоваться подобно *лямбда-выражениям* в качестве более краткой формы выражения экземпляров анонимных классов. Чтобы передать поведение в метод до выхода Java 8, Java-программисты зачастую определяли анонимные внутренние экземпляры класса:

```
JButton button = new JButton(); // Это код на Java
button.addActionListener(
  new ActionListener() {
    public void actionPerformed(ActionEvent event) {
      System.out.println("pressed!");
    }
  }
);
```

В этом примере создается анонимный экземпляр `ActionListener`, который передается методу `addActionListener`, принадлежащему `JButton` из Swing. Когда пользователь щелкает на кнопке, Swing вызывает в отношении этого экземпляра метод `actionPerformed`, который выводит сообщение `"pressed!"`.

[1] Чтобы в Scala можно было воспользоваться свойствами Java 8, начиная с версии Scala 2.12, требуется Java 8.

В Java 8 лямбда-выражение может использоваться везде, где требуется экземпляр класса или интерфейса, содержащий единственный абстрактный метод (single abstract method, SAM). `ActionListener` — именно такой интерфейс, поскольку содержит единственный абстрактный метод `actionPerformed`. Следовательно, с помощью лямбда-выражения можно зарегистрировать слушателя в отношении Swing-кнопки, например:

```
JButton button = new JButton(); // Это код Java 8
button.addActionListener(
  event -> System.out.println("pressed!")
);
```

В Scala в подобной ситуации также можно использовать безымянный экземпляр внутреннего класса, но можно и отдать предпочтение применению функционального литерала:

```
val button = new JButton
button.addActionListener(
  _ => println("pressed!")
)
```

Как было показано в разделе 21.1, подобный стиль программирования можно поддерживать за счет определения неявного преобразования из функционального типа `ActionEvent => Unit` в `ActionListener`.

В Scala в подобном случае разрешается использовать функциональный литерал даже при отсутствии такого неявного преобразования. Как и в Java 8, в Scala 2.12 позволяется задействовать функциональный тип там, где требуется экземпляр класса или трейта, декларирующий единственный абстрактный метод (SAM). В Scala, начиная с версии 2.12, это работает с любым SAM-типом. Например, можно определить трейт `Increaser` с единственным абстрактным методом `increase`:

```
scala> trait Increaser {
         def increase(i: Int): Int
       }
defined trait Increaser
```

Затем можно определить метод, получающий `Increaser`:

```
scala> def increaseOne(increaser: Increaser): Int =
         increaser.increase(1)
increaseOne: (increaser: Increaser)Int
```

Чтобы вызвать ваш новый метод, можно передать анонимный экземпляр трейта `Increaser`:

```
scala> increaseOne(
         new Increaser {
           def increase(i: Int): Int = i + 7
         }
       )
res0: Int = 8
```

Но в Scala 2.12 в качестве альтернативы можно просто воспользоваться функциональным литералом, поскольку `Increaser` относится к SAM-типу:

```
scala> increaseOne(i => i + 7) // Scala 2.12
res1: Int = 8
```

Использование Stream-объектов Java 8 из Scala

`Stream` в Java является функциональной структурой данных, которая предоставляет метод `map`, принимающий `java.util.function.IntUnaryOperator`. Из Scala `Stream.map` можно вызвать для увеличения значения каждого элемента `Array`-объекта на единицу:

```
scala> import java.util.function.IntUnaryOperator
import java.util.function.IntUnaryOperator

scala> import java.util.Arrays
import java.util.Arrays

scala> val stream = Arrays.stream(Array(1, 2, 3))
stream: java.util.stream.IntStream = ...

scala> stream.map(
         new IntUnaryOperator {
           def applyAsInt(i: Int): Int = i + 1
         }
       ).toArray
res3: Array[Int] = Array(2, 3, 4)
```

Но, поскольку `IntUnaryOperator` относится к SAM-типу, в Scala версии 2.12 и более поздних его можно вызвать в более краткой форме с помощью функционального литерала:

```
scala> val stream = Arrays.stream(Array(1, 2, 3))
stream: java.util.stream.IntStream = ...

scala> stream.map(i => i + 1).toArray // Scala 2.12
res4: Array[Int] = Array(2, 3, 4)
```

Обратите внимание: к SAM-типам можно адаптировать только функциональные *литералы*, а не произвольные выражения, имеющие функциональный тип. Рассмотрим, к примеру, `val`-переменную `f`, у которой есть тип `Int => Int`:

```
scala> val f = (i: Int) => i + 1
f: Int => Int = ...
```

Хотя у `f` тот же тип, что и у функционального литерала, переданного ранее `stream.map`, использовать `f` там, где требуется `IntUnaryOperator`, нельзя:

```
scala> val stream = Arrays.stream(Array(1, 2, 3))
stream: java.util.stream.IntStream = ...
```

```
scala> stream.map(f).toArray
<console>:16: error: type mismatch;
 found   : Int => Int
 required: java.util.function.IntUnaryOperator
       stream.map(f).toArray
                  ^
```

Чтобы воспользоваться `f`, можно явным образом указать вызов, задействуя функциональный литерал:

```
scala> stream.map(i => f(i)).toArray
res5: Array[Int] = Array(2, 3, 4)
```

Или же можно проаннотировать `f` при определении с помощью `IntUnaryOperator`, то есть того типа, который ожидается `Stream.map`:

```
scala> val f: IntUnaryOperator = i => i + 1
f: java.util.function.IntUnaryOperator = ...

scala> val stream = Arrays.stream(Array(1, 2, 3))
stream: java.util.stream.IntStream = ...

scala> stream.map(f).toArray
res6: Array[Int] = Array(2, 3, 4)
```

При работе со Scala 2.12 и Java 8 можно также, используя лямбда-выражения Java, вызывать из Java методы, скомпилированные Scala, передавая функциональные типы Scala. Хотя функциональные типы Scala определяются как трейты, включающие конкретные методы, Scala версии 2.12 и более поздних компилирует трейты в Java-интерфейсы с *методами по умолчанию*, которые являются новой особенностью Java 8. В результате функциональные типы Scala оказываются в Java SAM-типами.

Резюме

В большинстве случаев способ реализации Scala можно игнорировать и просто создавать и запускать на выполнение свой код. Но иногда бывает полезно, что называется, заглянуть под капот, поэтому в данной главе мы рассмотрели три аспекта реализации Scala на платформе Java: на что похоже преобразование кода, как аннотации Scala и Java работают вместе и как подстановочные типы Scala позволяют обращаться к подстановочным типам Java. Кроме того, мы рассмотрели использование из Scala имеющихся в Java примитивов многопоточных вычислений и компиляцию комбинированных Scala- и Java-проектов. Особое значение эти темы приобретают при совместном применении Scala и Java.

32 Фьючерсы и многопоточность

Одним из результатов распространения многоядерных процессоров стал повышенный интерес к многопоточным вычислениям. В Java поддержка таких вычислений выстроена вокруг совместно используемой памяти и блокировок. Этой поддержки вполне достаточно, однако она получилась слишком сложной для практической реализации. В стандартной библиотеке Scala предлагается альтернатива, которая позволяет обойти эти сложности за счет того, что основное внимание уделяется асинхронным преобразованиям неизменяемого состояния: `Future`.

В Java также предлагается `Future`, однако его вариант сильно отличается от того, что используется в Scala. Оба они представляют собой результат асинхронного вычисления, но `Future` в Java требует обращения к результату через блокирующий метод `get`. Несмотря на то что в Java, чтобы узнать о завершении `Future` перед вызовом `get`, можно вызвать метод `isDone`, избавляясь тем самым от любых блокировок, нужно дождаться завершения `Future` перед выполнением любых вычислений, использующих его результат.

В отличие от этого вы можете определять преобразования Scala `Future` независимо от того, завершено ли соответствующее вычисление. Каждое преобразование приводит к созданию нового экземпляра `Future`, который представляет асинхронный результат исходного экземпляра `Future`, преобразованный функцией. Поток, выполняющий вычисление, определяется предоставляемым неявно *контекстом выполнения*. Это позволяет давать описание асинхронных вычислений в виде последовательности преобразований неизменяемых значений, не ощущая при этом потребности в совместном использовании памяти и в блокировках.

32.1. Неприятности в раю

Каждый объект на платформе Java связан с логическим *монитором*, с помощью которого можно управлять многопоточным доступом к данным. Чтобы применить эту модель, следует решить, какие именно данные будут совместно использоваться несколькими потоками, и пометить как синхронизированные (`synchronized`) те разделы кода, которые обращаются к совместно используемым данным или управляют доступом к ним. Система выполнения Java применяет механизм блокировок

с целью гарантировать, что в определенный момент в синхронизированные разделы, охраняемые отдельно взятой блокировкой, войдет лишь один поток, позволяя тем самым дирижировать многопоточным доступом к совместно используемым данным.

Из соображений совместимости в Scala предоставляется доступ к имеющимся в Java примитивам многопоточных вычислений. В Scala могут вызываться методы `wait`, `notify` и `notifyAll`, и их предназначение точно такое же, как и в Java. Технически ключевого слова `synchronized` в Scala нет, но в этот язык включены предопределенные методы синхронизации, вызываемые следующим образом:

```
var counter = 0
synchronized {
  // Здесь может находиться только один поток
  counter = counter + 1
}
```

К сожалению, программистам было очень сложно создавать действительно надежные многопоточные приложения с помощью модели совместно используемых данных и блокировок, особенно с повышением объема и сложности приложений. Дело в том, что в каждой точке программы необходимо представлять, какие данные изменяются или к каким данным происходит обращение и могут ли к ним обращаться или вносить в них изменения другие потоки, удерживаемые от этих действий блокировкой. При каждом вызове метода требовалось понимать, какие именно средства блокировки он испробует для сдерживания, и нужно было убеждаться, что при попытке их получения не возникнет взаимная блокировка. Задачу усложняет еще и то, что блокировки, о которых вы рассуждаете, не фиксируются во время компиляции, поскольку программа во время выполнения может беспрепятственно создавать новые блокировки.

Усугубляет ситуацию то, что тестирование многопоточного кода не дает надежных результатов. Потоки имеют недетерминированный характер, поэтому программа может успешно пройти тестирование тысячу раз и сработать неподобающим образом при первом же запуске на машине клиента. Из-за этого при использовании общих данных и блокировок остается только уповать на корректность программы.

Более того, введением избыточной синхронизации проблема не решается. Уровень проблематичности повсеместного применения синхронизации может быть сопоставим с уровнем проблематичности полного отказа от нее. Хотя новые блокировки могут устранить вероятность появления гонок, одновременно с этим они повышают риск возникновения взаимных блокировок. Корректная программа, рассчитанная на длительный период использования, должна исключать как гонки потоков, так и взаимные блокировки, поэтому в манипуляциях с программой нельзя чересчур усердствовать в обоих направлениях.

Высокоуровневые абстракции для многопоточного программирования предоставляются библиотекой `java.util.concurrent`. Использование утилит многопоточных вычислений делает процесс многопоточного программирования гораздо менее подверженным возникновению ошибок, по сравнению с обкаткой ваших собственных абстракций, создаваемых на основе имеющихся в Java низкоуровневых примитивов синхронизации. И тем не менее утилиты многопоточных вычислений также основаны на применении модели совместно используемых данных

и блокировок, что не позволяет преодолевать основные трудности, возникающие при использовании этой модели.

32.2. Асинхронное выполнение и Try

Хотя универсального решения не существует, один из путей, позволяющих справиться с многопоточностью, в Scala предлагает `Future`, который может уменьшить, а зачастую и устранить необходимость рассматривать вопросы применения совместно используемых данных и блокировок. При вызове метода Scala он выполняет вычисление, пока вы ждете, и возвращает результат. Если этот результат имеет тип `Future`, то экземпляр `Future` заявляет еще одно вычисление, выполняемое в асинхронном режиме, которое зачастую выполняет другой поток. В результате этого многие операции, проводимые над `Future`, требуют неявного *контекста выполнения*, предоставляющего стратегию для асинхронного выполнения функций. Например, попытка создать фьючерс с помощью фабричного метода `Future.apply` без предоставления неявного контекста выполнения, то есть экземпляра `scala.concurrent.ExecutionContext`, породит ошибку компиляции:

```
scala> import scala.concurrent.Future
import scala.concurrent.Future

scala> val fut = Future { Thread.sleep(10000); 21 + 21 }
<console>:11: error: Cannot find an implicit ExecutionContext.
    You might pass an (implicit ec: ExecutionContext)
    parameter to your method or import
    scala.concurrent.ExecutionContext.Implicits.global.
       val fut = Future { Thread.sleep(10000); 21 + 21 }
                          ^
```

Сообщение об ошибке подсказывает один из путей решения проблемы: импорт глобального контекста выполнения, предоставляемого самим языком Scala. На JVM глобальным контекстом выполнения используется пул потоков (thread pool)[1]. Как только неявный контекст выполнения будет введен в область видимости, можно будет создавать фьючерс:

```
scala> import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.ExecutionContext.Implicits.global

scala> val fut = Future { Thread.sleep(10000); 21 + 21 }
fut: scala.concurrent.Future[Int] = ...
```

Фьючерс, созданный в предыдущем примере, выполняет в асинхронном режиме блок кода, используя глобальный контекст выполнения, затем завершается, выдавая значение `42`. Как только он приступит к выполнению, данный поток заснет

[1] Что касается Scala.js, то глобальный контекст выполнения помещает задачи в очередь событий JavaScript.

на 10 секунд. Таким образом, у данного фьючерса на завершение будет по крайней мере 10 секунд.

Выполнять опрос позволяют два метода класса `Future` — `isCompleted` и `value`. Вызов в отношении еще не завершившегося фьючерса метода `isCompleted` приведет к возвращению значения `false`, а метод `value` при таких обстоятельствах возвратит значение `None`:

```
scala> fut.isCompleted
res0: Boolean = false

scala> fut.value
res1: Option[scala.util.Try[Int]] = None
```

Как только фьючерс будет завершен (в данном случае после того, как пройдет по крайней мере 10 секунд), метод `isCompleted` возвратит значение `true`, а метод `value` — значение `Some`:

```
scala> fut.isCompleted
res2: Boolean = true

scala> fut.value
res3: Option[scala.util.Try[Int]] = Some(Success(42))
```

В `Option`-значении, возвращенном `value`, содержится `Try`. Как показано на рис. 32.1, `Try` выражается либо подклассом `Success`, содержащим значение `T`, либо подклассом `Failure`, содержащим исключение (экземпляр класса `java.lang.Throwable`). Предназначение `Try` заключается в предоставлении для асинхронных вычислений той же самой возможности, которую `try`-выражение предоставляло для синхронных вычислений: этот класс позволяет вам справляться с вероятностью внезапного завершения вычисления с генерацией исключения вместо возвращения результата[1].

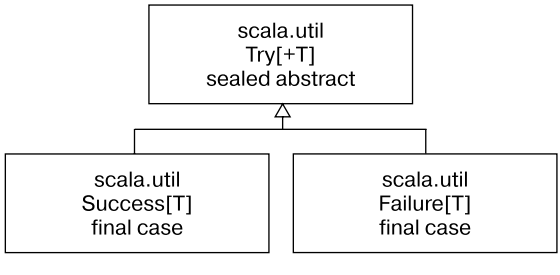


Рис. 32.1. Иерархия классов для Try

Чтобы обеспечить для синхронных вычислений перехват и обработку выданных методом исключений тем самым потоком, который этот метод вызвал, можно

[1] Следует заметить, что у Java Future тоже есть способ обработки потенциальной возможности генерации исключения в ходе асинхронного вычисления: его метод get выдаст это исключение, обернутое в ExecutionException.

воспользоваться инструкциями `try-catch`. Но при асинхронных вычислениях поток, инициировавший вычисление, зачастую переключается на выполнение других задач. Чуть позже, если асинхронное вычисление даст сбой с генерацией исключения, исходный поток уже не сможет обработать исключение в блоке `catch`. Следовательно, чтобы справиться с возможным внезапным сбоем и выдачей исключения взамен итогового значения при работе с экземпляром `Future`, выполняющим асинхронную работу, используется `Try`. Рассмотрим пример, показывающий, что получится при сбое асинхронной работы:

```
scala> val fut = Future { Thread.sleep(10000); 21 / 0 }
fut: scala.concurrent.Future[Int] = ...

scala> fut.value
res4: Option[scala.util.Try[Int]] = None
```

Затем по прошествии 10 секунд:

```
scala> fut.value
res5: Option[scala.util.Try[Int]] =
    Some(Failure(java.lang.ArithmeticException: / by zero))
```

32.3. Работа с фьючерсами

Фьючерсы в Scala позволяют определять преобразования результатов `Future` и получать *новые фьючерсы*, представляющие собой композицию из двух асинхронных вычислений: исходного и преобразования.

Преобразование фьючерсов с помощью map

Основная операция такого преобразования — `map`. Вместо блокировки при продолжении работы с другим вычислением можно просто отобразить следующее вычисление на фьючерс. Результатом станет новый фьючерс, который представляет исходный асинхронно вычисленный результат, асинхронно преобразованный функцией, переданной методу `map`.

Например, следующий фьючерс завершится по прошествии 10 секунд:

```
scala> val fut = Future { Thread.sleep(10000); 21 + 21 }
fut: scala.concurrent.Future[Int] = ...
```

Отображение данного фьючерса с помощью функции, увеличивающей значение на единицу, приведет к выдаче еще одного фьючерса. Этот новый фьючерс будет представлять вычисление, состоящее из исходного сложения, за которым последует инкремент:

```
scala> val result = fut.map(x => x + 1)
result: scala.concurrent.Future[Int] = ...

scala> result.value
res5: Option[scala.util.Try[Int]] = None
```

Как только исходный фьючерс завершится и функция будет применена к его результату, завершится и фьючерс, возвращенный методом `map`:

```
scala> result.value
res6: Option[scala.util.Try[Int]] = Some(Success(43))
```

Следует заметить, что операции, намеченные для выполнения в данном примере, — то есть создание фьючерса, вычисление суммы `21 + 21` и вычисление инкремента `42 + 1` — могут быть выполнены тремя разными потоками.

Преобразование фьючерсов с помощью выражений for

Во фьючерсе Scala также объявлен метод `flatMap`, поэтому преобразование фьючерсов можно выполнять с помощью выражения `for`. Рассмотрим, к примеру, следующие два фьючерса, которые по прошествии 10 секунд выдадут 42 и 46:

```
scala> val fut1 = Future { Thread.sleep(10000); 21 + 21 }
fut1: scala.concurrent.Future[Int] = ...

scala> val fut2 = Future { Thread.sleep(10000); 23 + 23 }
fut2: scala.concurrent.Future[Int] = ...
```

На основе этих двух фьючерсов можно получить новый, представляющий асинхронное суммирование их результатов:

```
scala> for {
         x <- fut1
         y <- fut2
       } yield x + y
res7: scala.concurrent.Future[Int] = ...
```

Как только завершатся исходный фьючерс и последующее суммирование, вы сможете увидеть результат:

```
scala> res7.value
res8: Option[scala.util.Try[Int]] = Some(Success(88))
```

Выражение `for` сериализует свои преобразования[1], если не создать фьючерсы до применения выражения `for`, поэтому они не будут запущены параллельно. Например, хотя предыдущее выражение `for` требует для завершения около 10 секунд, следующее выражение `for` требует как минимум вдвое больше времени:

```
scala> for {
         x <- Future { Thread.sleep(10000); 21 + 21 }
         y <- Future { Thread.sleep(10000); 23 + 23 }
       } yield x + y
res9: scala.concurrent.Future[Int] = ...
```

[1] Выражение for, показанное в этом примере, будет переписано в виде вызова fut1.flatMap, переданного в функцию, которая вызывает fut2.map : fut1.flatMap(x => fut2.map(y => x + y)).

```
scala> res9.value
res27: Option[scala.util.Try[Int]] = None

scala> // Понадобится как минимум 20 секунд на завершение

scala> res9.value
res28: Option[scala.util.Try[Int]] = Some(Success(88))
```

Создание фьючерса: Future.failed, Future.successful, Future.fromTry и Promise

Кроме метода `apply`, использованного в приведенных ранее примерах в целях создания фьючерсов, объект-компаньон `Future` включает также три фабричных метода для создания уже завершенных фьючерсов: `successful`, `failed` и `fromTry`. Эти методы не требуют контекста выполнения `ExecutionContext`.

Фабричный метод `successful` создает фьючерс, который уже успешно завершился:

```
scala> Future.successful { 21 + 21 }
res2: scala.concurrent.Future[Int] = ...
```

Метод `failed` создает фьючерс, который уже дал сбой:

```
scala> Future.failed(new Exception("bummer!"))
res3: scala.concurrent.Future[Nothing] = ...
```

Метод `fromTry` создает уже завершенный фьючерс из `Try`:

```
scala> import scala.util.{Success,Failure}
import scala.util.{Success, Failure}

scala> Future.fromTry(Success { 21 + 21 })
res4: scala.concurrent.Future[Int] = ...

scala> Future.fromTry(Failure(new Exception("bummer!")))
res5: scala.concurrent.Future[Nothing] = ...
```

Наиболее универсальный способ создания фьючерса — использование обещания `Promise`. Давая обещание, можно получить управляемый им фьючерс. Он будет завершен, когда вы завершите обещание. Рассмотрим пример:

```
scala> val pro = Promise[Int]
pro: scala.concurrent.Promise[Int] = ...

scala> val fut = pro.future
fut: scala.concurrent.Future[Int] = ...

scala> fut.value
res8: Option[scala.util.Try[Int]] = None
```

Завершить обещание можно с помощью методов `success`, `failure` и `complete`. Эти методы, применяемые к `Promise`, аналогичны ранее рассмотренным, которые используются для создания уже завершенных фьючерсов. Например, метод `success` успешно завершит фьючерс:

```
scala> pro.success(42)
res9: pro.type = ...

scala> fut.value
res10: Option[scala.util.Try[Int]] = Some(Success(42))
```

Метод `failure` получает исключение, которое заставляет фьючерс выдать сбой с этим исключением. Метод `complete` получает `Try`. Кроме того, есть метод `completeWith`, получающий фьючерс. Принадлежащий обещанию фьючерс в дальнейшем зеркально отобразит состояние завершения фьючерса, переданного методу `completeWith`.

Фильтрация: filter и collect

Фьючерсы Scala предлагают два метода, `filter` и `collect`, позволяющие гарантировать, что свойство истинно (`true`) относительно значения фьючерса. Метод `filter` проверяет результат фьючерса, оставляя его неизменным, если он валиден. Пример, гарантирующий, что `Int`-значение является положительным, выглядит так:

```
scala> val fut = Future { 42 }
fut: scala.concurrent.Future[Int] = ...

scala> val valid = fut.filter(res => res > 0)
valid: scala.concurrent.Future[Int] = ...

scala> valid.value
res0: Option[scala.util.Try[Int]] = Some(Success(42))
```

Если значение фьючерса не валидно, то фьючерс, возвращаемый `filter`, даст сбой с выдачей исключения `NoSuchElementException`:

```
scala> val invalid = fut.filter(res => res < 0)
invalid: scala.concurrent.Future[Int] = ...

scala> invalid.value
res1: Option[scala.util.Try[Int]] =
  Some(Failure(java.util.NoSuchElementException:
  Future.filter predicate is not satisfied))
```

В `Future` предлагается также метод `withFilter`, поэтому ту же операцию можно выполнить с выражениями фильтров:

```
scala> val valid = for (res <- fut if res > 0) yield res
valid: scala.concurrent.Future[Int] = ...
```

```
scala> valid.value
res2: Option[scala.util.Try[Int]] = Some(Success(42))

scala> val invalid = for (res <- fut if res < 0) yield res
invalid: scala.concurrent.Future[Int] = ...

scala> invalid.value
res3: Option[scala.util.Try[Int]] =
  Some(Failure(java.util.NoSuchElementException:
  Future.filter predicate is not satisfied))
```

Имеющийся в `Future` метод `collect` позволяет проверить на допустимость значение фьючерса и превратить все в одну операцию. Если переданная `collect` частично примененная функция определена как результат фьючерса, то фьючерс, возвращаемый `collect`, будет успешно завершен с этим значением, преобразованным функцией:

```
scala> val valid =
         fut collect { case res if res > 0 => res + 46 }
valid: scala.concurrent.Future[Int] = ...

scala> valid.value
res17: Option[scala.util.Try[Int]] = Some(Success(88))
```

В противном случае фьючерс даст сбой с выдачей исключения `NoSuchElementException`:

```
scala> val invalid =
         fut collect { case res if res < 0 => res + 46 }
invalid: scala.concurrent.Future[Int] = ...

scala> invalid.value
res18: Option[scala.util.Try[Int]] =
  Some(Failure(java.util.NoSuchElementException:
  Future.collect partial function is not defined at: 42))
```

Обработка сбоев: failed, fallBackTo, recover и recoverWith

Фьючерсы в Scala позволяют работать с теми фьючерсами, которые дали сбой, с помощью методов `failed`, `fallBackTo`, `recover` и `recoverWith`. Метод `failed` превратит давший сбой фьючерс любого типа в успешно завершенный `Future[Throwable]`, который хранит в себе исключение, ставшее причиной сбоя. Рассмотрим пример:

```
scala> val failure = Future { 42 / 0 }
failure: scala.concurrent.Future[Int] = ...

scala> failure.value
res23: Option[scala.util.Try[Int]] =
  Some(Failure(java.lang.ArithmeticException: / by zero))
```

```
scala> val expectedFailure = failure.failed
expectedFailure: scala.concurrent.Future[Throwable] = ...

scala> expectedFailure.value
res25: Option[scala.util.Try[Throwable]] =
  Some(Success(java.lang.ArithmeticException: / by zero))
```

Если фьючерс, в отношении которого вызван метод `failed`, в конечном счете успешно завершится, фьючерс, возвращенный методом `failed`, сам даст сбой с выдачей исключения `NoSuchElementException`. Поэтому метод `failed` применим только в том случае, когда ожидается сбой фьючерса. Рассмотрим пример:

```
scala> val success = Future { 42 / 1 }
success: scala.concurrent.Future[Int] = ...

scala> success.value
res21: Option[scala.util.Try[Int]] = Some(Success(42))

scala> val unexpectedSuccess = success.failed
unexpectedSuccess: scala.concurrent.Future[Throwable] = ...

scala> unexpectedSuccess.value
res26: Option[scala.util.Try[Throwable]] =
  Some(Failure(java.util.NoSuchElementException:
  Future.failed not completed with a throwable.))
```

Метод `fallbackTo` позволяет предоставить альтернативный фьючерс для использования в том случае, если фьючерс, в отношении которого был вызван `fallbackTo`, даст сбой. Рассмотрим пример, в котором давший сбой фьючерс уступает успешно завершаемому:

```
scala> val fallback = failure.fallbackTo(success)
fallback: scala.concurrent.Future[Int] = ...

scala> fallback.value
res27: Option[scala.util.Try[Int]] = Some(Success(42))
```

Если исходный фьючерс, в отношении которого был вызван метод `fallbackTo`, даст сбой, то этот сбой, переданный `fallbackTo`, по сути, игнорируется. Фьючерс, возвращенный методом `fallbackTo`, даст сбой с выдачей исходного исключения. Рассмотрим пример:

```
scala> val failedFallback = failure.fallbackTo(
         Future { val res = 42; require(res < 0); res }
       )
failedFallback: scala.concurrent.Future[Int] = ...

scala> failedFallback.value
res28: Option[scala.util.Try[Int]] =
  Some(Failure(java.lang.ArithmeticException: / by zero))
```

Метод `recover` позволяет преобразовать давший сбой фьючерс в успешно завершенный, разрешая результату последнего пройти через него в неизмененном виде. Например, в отношении фьючерса, давшего сбой с выдачей исключения `ArithmeticException`, можно воспользоваться методом `recover`, чтобы превратить сбой в успешное завершение:

```
scala> val recovered = failedFallback recover {
         case ex: ArithmeticException => -1
       }
recovered: scala.concurrent.Future[Int] = ...

scala> recovered.value
res32: Option[scala.util.Try[Int]] = Some(Success(-1))
```

Если исходный фьючерс не дал сбой, то фьючерс, возвращенный методом `recover`, завершится с тем же значением:

```
scala> val unrecovered = fallback recover {
         case ex: ArithmeticException => -1
       }
unrecovered: scala.concurrent.Future[Int] = ...

scala> unrecovered.value
res33: Option[scala.util.Try[Int]] = Some(Success(42))
```

Аналогично этому, если частично примененная функция, переданная методу, не определена для исключения, с которым в конечном счете дал сбой исходный фьючерс, то далее пройдет исходный отказ:

```
scala> val alsoUnrecovered = failedFallback recover {
         case ex: IllegalArgumentException => -2
       }
alsoUnrecovered: scala.concurrent.Future[Int] = ...

scala> alsoUnrecovered.value
res34: Option[scala.util.Try[Int]] =
  Some(Failure(java.lang.ArithmeticException: / by zero))
```

Метод `recoverWith` похож на метод `recover`, за исключением того, что вместо восстановления в виде значения, как в `recover`, метод `recoverWith` позволяет выполнять восстановление в значение `Future`. Рассмотрим пример:

```
scala> val alsoRecovered = failedFallback recoverWith {
         case ex: ArithmeticException => Future { 42 + 46 }
       }
alsoRecovered: scala.concurrent.Future[Int] = ...

scala> alsoRecovered.value
res35: Option[scala.util.Try[Int]] = Some(Success(88))
```

Как и в случае применения `recover`, если либо исходный фьючерс не даст сбой, либо частично примененная функция, переданная методу `recoverWith`, не будет определена для исключения, с которым в конечном счете дал сбой исходный фьючерс, через фьючерс, возвращенный методом `recoverWith`, то будет передан исходный успешный (или сбойный) результат.

Отображение обеих возможностей: transform

Определенный в `Future` метод `transform` получает две функции, с помощью которых должен преобразовать фьючерс: одну для использования в случае успеха, другую — в случае сбоя:

```
scala> val first = success.transform(
         res => res * -1,
         ex => new Exception("see cause", ex)
       )
first: scala.concurrent.Future[Int] = ...
```

Если фьючерс завершается успешно, то используется первая функция:

```
scala> first.value
res42: Option[scala.util.Try[Int]] = Some(Success(-42))
```

Если дает сбой, то используется вторая:

```
scala> val second = failure.transform(
         res => res * -1,
         ex => new Exception("see cause", ex)
       )
second: scala.concurrent.Future[Int] = ...

scala> second.value
res43: Option[scala.util.Try[Int]] =
  Some(Failure(java.lang.Exception: see cause))
```

Обратите внимание: с помощью показанного в предыдущих примерах метода `transform` превратить успешно завершенный фьючерс в сбойный и, наоборот, сбойный — в успешно завершенный невозможно. Чтобы упростить подобные преобразования, в Scala 2.12 была введена альтернативная перегруженная форма `transform`, которая использует функцию из `Try` в `Try`. Рассмотрим несколько примеров:

```
scala> val firstCase = success.transform { // Scala 2.12
         case Success(res) => Success(res * -1)
         case Failure(ex) =>
           Failure(new Exception("see cause", ex))
       }
first: scala.concurrent.Future[Int] = ...
```

```
scala> firstCase.value
res6: Option[scala.util.Try[Int]] = Some(Success(-42))

scala> val secondCase = failure.transform {
         case Success(res) => Success(res * -1)
         case Failure(ex) =>
           Failure(new Exception("see cause", ex))
       }
secondCase: scala.concurrent.Future[Int] = ...

scala> secondCase.value
res8: Option[scala.util.Try[Int]] =
   Some(Failure(java.lang.Exception: see cause))
```

А вот как выглядит пример использования нового метода `transform` для превращения сбоя в успешное завершение:

```
scala> val nonNegative = failure.transform { // Scala 2.12
         case Success(res) => Success(res.abs + 1)
         case Failure(_) => Success(0)
       }
nonNegative: scala.concurrent.Future[Int] = ...

scala> nonNegative.value
res11: Option[scala.util.Try[Int]] = Some(Success(0))
```

Объединение фьючерсов: zip, Future.foldLeft, Future.reduceLeft, Future.sequence и Future.traverse

Класс `Future` и его объект-компаньон предлагают методы объединения нескольких фьючерсов. Метод `zip` превратит два успешных фьючерса во фьючерс-кортеж из обоих значений. Рассмотрим пример:

```
scala> val zippedSuccess = success zip recovered
zippedSuccess: scala.concurrent.Future[(Int, Int)] = ...

scala> zippedSuccess.value
res46: Option[scala.util.Try[(Int, Int)]] =
     Some(Success((42,-1)))
```

Но если какой-либо из фьючерсов даст сбой, то фьючерс, возвращенный методом `zip`, также даст сбой с выдачей того же самого исключения:

```
scala> val zippedFailure = success zip failure
zippedFailure: scala.concurrent.Future[(Int, Int)] = ...

scala> zippedFailure.value
res48: Option[scala.util.Try[(Int, Int)]] =
  Some(Failure(java.lang.ArithmeticException: / by zero))
```

Если сбойными будут оба фьючерса, то получающийся сбойный фьючерс будет содержать исключение, сохраненное в исходном фьючерсе — в том, в отношении которого был вызван метод `zip`.

Объект-компаньон класса `Future` предлагает метод `foldLeft`, позволяющий аккумулировать результат, получаемый из коллекции фьючерсов, имеющей тип `Iterable`, выдавая результат фьючерса. Если успешными будут все фьючерсы коллекции, то получающийся фьючерс будет успешным с аккумулированным результатом. Если любой из фьючерсов коллекции даст сбой, то получающийся фьючерс будет сбойным. Если же дадут сбой несколько фьючерсов, то результат будет сбойным с тем самым исключением, с которым потерпел сбой первый фьючерс (самый первый в `Iterable`-коллекции). Рассмотрим пример:

```
scala> val fortyTwo = Future { 21 + 21 }
fortyTwo: scala.concurrent.Future[Int] = ...

scala> val fortySix = Future { 23 + 23 }
fortySix: scala.concurrent.Future[Int] = ...

scala> val futureNums = List(fortyTwo, fortySix)
futureNums: List[scala.concurrent.Future[Int]] = ...

scala> val folded =
         Future.foldLeft(futureNums)(0) { (acc, num) =>
           acc + num
         }
folded: scala.concurrent.Future[Int] = ...

scala> folded.value
res53: Option[scala.util.Try[Int]] = Some(Success(88))
```

Метод `Future.reduceLeft` выполняет свертку без нуля, используя в качестве стартового значения результат исходного фьючерса. Рассмотрим пример:

```
scala> val reduced =
         Future.reduceLeft(futureNums) { (acc, num) =>
           acc + num
         }
reduced: scala.concurrent.Future[Int] = ...

scala> reduced.value
res54: Option[scala.util.Try[Int]] = Some(Success(88))
```

Если передать `reduce` пустую коллекцию, то получившийся фьючерс даст сбой с выдачей исключения `NoSuchElementException`.

Метод `Future.sequence` выполняет преобразование `TraversableOnce`-коллекции фьючерсов во фьючерс, содержащий значения типа `TraversableOnce`. Например, в следующем примере `sequence` используется для преобразования `List[Future[Int]]` в `Future[List[Int]]`:

```
scala> val futureList = Future.sequence(futureNums)
futureList: scala.concurrent.Future[List[Int]] = ...
```

```
scala> futureList.value
res55: Option[scala.util.Try[List[Int]]] =
  Some(Success(List(42, 46)))
```

Метод `Future.traverse` превратит `TraversableOnce`-коллекцию из элементов любого типа в `TraversableOnce`-коллекцию фьючерсов и сведет ее во фьючерс значений `TraversableOnce`. Вот как, к примеру, `Future.traverse` превратит `List[Int]` в `Future[List[Int]]`:

```
scala> val traversed =
        Future.traverse(List(1, 2, 3)) { i => Future(i) }
traversed: scala.concurrent.Future[List[Int]] = ...

scala> traversed.value
res58: Option[scala.util.Try[List[Int]]] =
  Some(Success(List(1, 2, 3)))
```

Получение побочных эффектов: foreach, onComplete и andThen

Иногда после завершения фьючерса требуется получить некий побочный эффект. Для этого в классе `Future` есть несколько методов. Наиболее основательный из них — метод `foreach`, выдающий побочный эффект при успешном завершении фьючерса. Так, в следующем примере `println` выполняется только при успешном завершении фьючерса, но не в случае его сбоя:

```
scala> failure.foreach(ex => println(ex))

scala> success.foreach(res => println(res))
42
```

Выражение `for` без элемента `yield` будет перезаписано в вызов `foreach`, поэтому позволяет добиться того же эффекта:

```
scala> for (res <- failure) println(res)

scala> for (res <- success) println(res)
42
```

Класс `Future` также предоставляет два метода для регистрации функций обратного вызова. Метод `onComplete` будет выполнен независимо от успешного или сбойного завершения фьючерса. Функции будет передано значение типа `Try` либо в виде значения подтипа `Success`, содержащего результат успешно завершенного фьючерса, либо в виде значения подтипа `Failure`, содержащего исключение, ставшее причиной сбоя фьючерса. Рассмотрим пример:

```
scala> import scala.util.{Success, Failure}
import scala.util.{Success, Failure}

scala> success onComplete {
        case Success(res) => println(res)
```

```
        case Failure(ex) => println(ex)
      }
42

scala> failure onComplete {
        case Success(res) => println(res)
        case Failure(ex) => println(ex)
      }
java.lang.ArithmeticException: / by zero
```

Класс `Future` не дает гарантий того, что будет соблюден некий порядок выполнения функций обратного вызова, зарегистрированных с помощью `onComplete`. Если нужно обязательно придерживаться какого-то порядка выполнения функций обратного вызова, то следует вместо этого метода воспользоваться методом `andThen`. Он возвращает новый фьючерс, отражающий исходный фьючерс (также завершающийся успешно или со сбоем), в отношении которого вызывался `andThen`, но он не завершится, пока не будет полностью выполнена функция обратного вызова:

```
scala> val newFuture = success andThen {
        case Success(res) => println(res)
        case Failure(ex) => println(ex)
      }
42
newFuture: scala.concurrent.Future[Int] = ...

scala> newFuture.value
res76: Option[scala.util.Try[Int]] = Some(Success(42))
```

Обратите внимание: если функция обратного вызова, переданная `andThen`, выдает при выполнении исключение, то данное исключение не будет распространено на следующие функции обратного вызова или выдано получающимся в результате фьючерсом.

Другие методы, добавленные в версии 2.12: flatten, zipWith и transformWith

Метод `flatten`, добавленный в версии 2.12, преобразует `Future`-объект, вложенный в другой `Future`-объект, в `Future`-объект вложенного типа. Например, `flatten` может преобразовать `Future[Future[Int]]` в `Future[Int]`:

```
scala> val nestedFuture = Future { Future { 42 } }
nestedFuture: Future[Future[Int]] = ...

scala> val flattened = nestedFuture.flatten // Scala 2.12
flattened: scala.concurrent.Future[Int] = Future(Success(42))
```

Метод `zipWith`, добавленный в версии 2.12, по сути, пакует два `Future`-объекта вместе, затем применяет к получившемуся кортежу метод `map`. Рассмотрим двушаговый процесс, в котором за `zip` следует `map`:

```
scala> val futNum = Future { 21 + 21 }
futNum: scala.concurrent.Future[Int] = ...

scala> val futStr = Future { "ans" + "wer" }
futStr: scala.concurrent.Future[String] = ...

scala> val zipped = futNum zip futStr
zipped: scala.concurrent.Future[(Int, String)] = ...

scala> val mapped = zipped map {
         case (num, str) => s"$num is the $str"
       }
mapped: scala.concurrent.Future[String] = ...

scala> mapped.value
res2: Option[scala.util.Try[String]] =
    Some(Success(42 is the answer))
```

Метод `zipWith` позволяет выполнять ту же операцию за один шаг:

```
scala> val fut = futNum.zipWith(futStr) { // Scala 2.12
         case (num, str) => s"$num is the $str"
       }
zipWithed: scala.concurrent.Future[String] = ...

scala> fut.value
res3: Option[scala.util.Try[String]] =
    Some(Success(42 is the answer))
```

Класс `Future` в Scala 2.12 также получил метод `transformWith`, который позволяет выполнить преобразование с помощью функции из `Try` в `Future`. Рассмотрим пример:

```
scala> val flipped = success.transformWith { // Scala 2.12
         case Success(res) =>
           Future { throw new Exception(res.toString) }
         case Failure(ex) => Future { 21 + 21 }
       }
flipped: scala.concurrent.Future[Int] = ...

scala> flipped.value
res5: Option[scala.util.Try[Int]] =
     Some(Failure(java.lang.Exception: 42))
```

Метод `transformWith` похож на новый, добавленный в Scala 2.12 перегруженный метод `transform`, за исключением того, что вместо выдачи `Try` в переданную вами функцию, как в `transform`, метод `transformWith` позволяет возвращать фьючерс.

32.4. Тестирование с помощью фьючерсов

Одним из преимуществ фьючерсов Scala является то, что они помогают избегать блокировки. В большинстве реализаций JVM-машин после создания всего лишь нескольких тысяч потоков затратность переключения контекста между потоками снизит производительность до неприемлемого уровня. Избегая блокировок, можно сохранить в рабочем состоянии намеченное конечное количество потоков. Тем не менее Scala позволяет при необходимости осуществлять блокировку результата фьючерса. Блокировку в ожидании результата фьючерса обеспечивает имеющийся в Scala объект `Await`. Рассмотрим пример:

```
scala> import scala.concurrent.Await
import scala.concurrent.Await

scala> import scala.concurrent.duration._
import scala.concurrent.duration._

scala> val fut = Future { Thread.sleep(10000); 21 + 21 }
fut: scala.concurrent.Future[Int] = ...

scala> val x = Await.result(fut, 15.seconds) // blocks
x: Int = 42
```

Метод `Await.result` получает объект типа `Future` и объект типа `Duration`. Последний из них показывает продолжительность заданного `Await.result` ожидания завершения `Future`-объекта. В данном примере `Duration`-объект получил указание ждать 15 секунд. Следовательно, метод `Await.result` не достигнет превышения времени ожидания до завершения фьючерса с конечным результатом 42.

Одной из задач, где блокировка, несомненно, приветствуется, является тестирование асинхронного кода. После возвращения управления из `Await.result` появится возможность выполнить на основе результата вычисление, подобное утверждению при проведении теста:

```
scala> import org.scalatest.matchers.should.Matchers._
import org.scalatest.matchers.should.Matchers._

scala> x should be (42)
res0: org.scalatest.Assertion = Succeeded
```

В качестве альтернативного варианта можно воспользоваться конструкциями блокировки, предоставляемыми имеющимся в Scala трейтом `ScalaFutures`. К примеру, метод `futureValue`, неявно добавляемый `ScalaFutures` к `Future`-объекту, установит блокировку до завершения фьючерса. Если он завершится сбоем, то метод `futureValue` выдаст исключение `TestFailedException` с описанием проблемы. В случае успешного завершения фьючерса метод `futureValue` воз-

вратит успешный результат и в отношении него будет позволено использовать утверждение:

```
scala> import org.scalatest.concurrent.ScalaFutures._
import org.scalatest.concurrent.ScalaFutures._

scala> val fut = Future { Thread.sleep(10000); 21 + 21 }
fut: scala.concurrent.Future[Int] = ...

scala> fut.futureValue should be (42) // futureValue blocks
res1: org.scalatest.Assertion = Succeeded
```

Хотя применение блокировки при тестировании зачастую приносит пользу, в ScalaTest 3.0 добавлены асинхронные стили тестирования, позволяющие тестировать фьючерсы без блокировок. Располагая фьючерсом, вместо блокировок и проверки утверждений в отношении результата можно отобразить утверждения напрямую на этот фьючерс и возвратить получившийся `Future[Assertion]` среде ScalaTest. Соответствующий пример показан в листинге 32.1. Когда фьючерс-утверждение завершится, ScalaTest инициирует события (успешного завершения теста, его сбойного завершения и т. п.), чтобы отрапортовать о результате теста в асинхронном режиме.

Листинг 32.1. Возвращение в ScalaTest фьючерса-утверждения

```
import org.scalatest.funspec.AsyncFunSpec
import scala.concurrent.Future

class AddSpec extends AsyncFunSpec {

  def addSoon(addends: Int * ): Future[Int] =
        Future { addends.sum }

  describe("addSoon") {
    it("will eventually compute a sum of passed Ints") {
      val futureSum: Future[Int] = addSoon(1, 2)
      // Утверждение можно отобразить на Future-объект, затем вернуть
      // получившийся фьючерс Future[Assertion] в адрес ScalaTest:
      futureSum map { sum => assert(sum == 3) }
    }
  }
}
```

Использование асинхронного тестирования иллюстрирует общий принцип работы с фьючерсами: находясь в их пространстве, старайтесь его не покидать. Не ставьте блокировку на фьючерсе с последующим вычислением результата. Продолжайте работу в асинхронном режиме, выполняя последовательности преобразований, каждое из которых возвращает для преобразования новый фьючерс. Для получения результатов в пространстве фьючерсов регистрируйте по завершении фьючерсов побочные эффекты, выполняемые асинхронно. Такой подход поможет вам максимально задействовать все имеющиеся потоки.

Резюме

Многопоточное программирование наделяет вас большими возможностями. Оно позволяет упростить код и получить преимущества от наличия в системе нескольких процессоров. К великому сожалению, большинство широко используемых многопоточных примитивов, потоков, блокировок и мониторов похоже на минное поле, усыпанное взаимными блокировками и гонками. Фьючерсы обеспечивают проход в этом минном поле, позволяющий создавать многопоточные программы без особого риска столкновения со взаимными блокировками и гонками. В этой главе среди всего прочего мы представили некоторые основные конструкции для работы с фьючерсами в Scala, включая приемы создания фьючерсов и их преобразования, а также способы тестирования. Затем мы показали, как можно воспользоваться этими конструкциями в качестве части общего стиля фьючерсов.

33 Синтаксический разбор с помощью комбинаторов

Когда-нибудь вам понадобится обработать данные на небольшом языке специального назначения. Например, потребуется прочитать конфигурационные файлы вашей программы и сделать их более подходящими для внесения изменений вручную, чем файлы формата XML. Или же в вашей программе понадобится поддержка языка ввода данных, например поисковой запрос с булевыми операторами (компьютер, найди мне фильм «с *космическими кораблями* и без *любовных историй*»). Независимо от причин вам понадобится *парсер*. Вам нужен способ превращения входного языка в некую структуру данных, которую сможет обработать ваша программа.

Выбор, по сути, невелик. Есть вариант создать собственный парсер (и лексический анализатор). Если вы не узкий специалист в этом деле, то справиться с подобной задачей будет нелегко. И даже если квалификация позволяет, на это все равно уйдет уйма времени.

Альтернативным вариантом будет использование генераторов парсеров. Таких генераторов немного. Наиболее известные — Yacc и Bison, которые применяются для парсеров на языке C, и ANTLR для парсеров на языке Java. Скорее всего, для работы вам понадобится также генератор сканера, такой как Lex, Flex или JFlex. Если не принимать во внимание некоторые неудобства, то такое решение может стать наилучшим. Для этого понадобится изучить новые программные средства, включая их иногда малопонятные сообщения об ошибках. Вдобавок предстоит понять, как связать выход этих утилит со своей программой. Все это может ограничить выбор языка программирования и усложнить цепочку применяемых инструментов.

В этой главе представлен третий вариант: вместо того чтобы использовать внешний предметно-ориентированный язык генаратора парсера, задействовать *внутренний предметно-ориентированный язык* (domain specific language), или, для краткости, внутренний DSL. Данный язык будет состоять из библиотеки *комбинаторов парсеров*, то есть функций и операторов, определенных в Scala и применяемых в качестве строительных блоков парсеров. Эти блоки будут поэлементно отображаться на конструкции контекстно-свободной грамматики, облегчая понимание последних.

В этой главе, в разделе 33.6, вводится только одна ранее не встречавшаяся особенность языка — использование псевдонимов `this`. Однако весьма интенсивно задействуется ряд других особенностей языка, рассмотренных в предыдущих главах. Среди них немаловажную роль играют параметризованные типы, абстрактные типы, применение функций в качестве объектов, перегрузка операторов, использование параметров до востребования и неявных преобразований. В этой главе мы покажем, как эти элементы языка можно скомбинировать при создании библиотеки очень высокого уровня.

Рассматриваемые в главе понятия будут, вероятно, более высокого порядка, чем те, которые мы изучали прежде. Эта глава будет полезна тем, у кого есть неплохие базовые знания об устройстве компиляторов, поскольку она поможет их углубить. Но для усвоения изложенного здесь материала нужно неплохо разбираться в регулярных и контекстно-свободных грамматиках. Если же эта область вызывает затруднения, то материал данной главы можно свободно пропустить.

33.1. Пример: арифметические выражения

Начнем с примера. Предположим, что нужно сконструировать парсер для арифметических выражений, состоящих из чисел с плавающей точкой, круглых скобок и бинарных операторов `+`, `-`, `*` и `/`. В качестве первого шага всегда нужно записать грамматику анализируемого парсером языка. Грамматика для арифметических выражений выглядит так:

```
   expr   ::=   term { "+" term | "-" term}.
   term   ::=   factor { " * " factor | "/" factor}.
 factor   ::=   floatingPointNumber | "(" expr ")" .
```

Здесь знак `|` обозначает альтернативный выбор правил, а знаки `{ ... }` — повторение (нуль и более раз). И хотя в данном примере знаки `[ ... ]` не используются, они обозначают необязательное наличие.

Этой контекстно-свободной грамматикой формально определяется язык арифметических выражений. Каждое выражение, представленное в виде *`expr`*, является синтаксическим *термом* (*`term`*), за которым могут располагаться последовательность операторов `+` или `-` и дополнительные *термы*. *Терм* — *фактор* (*`factor`*), за которым, возможно, идут последовательность операторов `*` или `/` и дополнительные *факторы*. *Фактор* — либо числовой литерал, либо выражение в круглых скобках. Обратите внимание: в грамматике уже закодирована относительная степень приоритетности операторов. Например, у оператора `*` привязка более крепкая, чем у `+`, поскольку операция `*` дает *терм*, а операция `+` дает *выражение* (*`expr`*), а оно может содержать *термы*, но *терм* не может содержать *выражение*, если только оно не заключено в круглые скобки.

Итак, грамматика определена. Что делать дальше? Если используются имеющиеся в Scala комбинаторы парсеров, то основное уже сделано! Нужно лишь

выполнить некоторые систематизированные замены текста и, как показано в листинге 33.1, заключить парсер в класс.

Листинг 33.1. Парсер арифметического выражения

```
import scala.util.parsing.combinator._

class Arith extends JavaTokenParsers {
  def expr: Parser[Any] = term~rep("+"~term | "-"~term)
  def term: Parser[Any] = factor~rep(" * "~factor | "/"~factor)
  def factor: Parser[Any] = floatingPointNumber | "("~expr~")"
}
```

Парсеры для арифметических выражений содержатся в классе, являющемся наследником трейта `JavaTokenParsers`. Этот трейт предоставляет основной механизм для написания парсера, а также некоторые примитивные парсеры, распознающие отдельные классы слов: идентификаторы, строковые литералы и числа. В примере, показанном в листинге 33.1, нужен всего лишь примитивный парсер `floatingPointNumber`, наследуемый из данного трейта.

Набор правил для арифметических выражений представлен тремя определениями в классе `Arith`. Вы увидите, что они весьма точно соответствуют набору правил контекстно-свободной грамматики. Фактически эту часть можно создать из контекстно-свободной грамматики автоматически, выполняя ряд простых замен текста.

1. Каждое правило становится методом, поэтому перед ним нужно ставить префикс `def`.
2. Результирующим типом любого метода является `Parser[Any]`, следовательно, нужно заменить обозначение `::=` кодом `: Parser[Any] =`. Значение типа `Parser[Any]` и порядок его уточнения мы разъясним в этой главе чуть позже.
3. В грамматике последовательная композиция была неявной, но в программе выражена явным оператором `~`. Поэтому нужно вставить оператор `~` между любыми двумя последовательными обозначениями правила. В примере, показанном в листинге 33.1, мы решили не ставить пробелы с обеих сторон от оператора `~`. Так код парсера будет близок внешнему виду грамматики — он просто заменяет пробелы символами `~`.
4. Повторения выражены кодом `rep( ... )` вместо `{ ... }`. Аналогично этому (хотя в примере это не показано) опциональные элементы выражаются кодом `opt( ... )` вместо `[ ... ]`.
5. Точка (`.`) в конце всех правил опущена, но при желании вместо нее можно поставить точку с запятой (`;`).

Вот, собственно, и все. В получившемся в итоге классе `Arith` определяются три парсера: `expr`, `term` и `factor`, которые могут использоваться для синтаксического разбора арифметических выражений и их частей.

33.2. Запуск парсера

Парсер можно использовать со следующей небольшой программой:

```
object ParseExpr extends Arith {
  def main(args: Array[String]) = {
    println("input : " + args(0))
    println(parseAll(expr, args(0)))
  }
}
```

Объект `ParseExpr` определяет метод `main`, выполняющий разбор первого аргумента переданной ему командной строки. С его помощью выводятся исходный аргумент ввода, а затем его проанализированная парсером версия. Разбор выполняется следующим выражением:

```
parseAll(expr, input)
```

В нем парсер `expr` применяется к введенным данным. Ожидается соответствие всем введенным данным, то есть предполагается, что, кроме разобранного выражения, символов нет. Есть также метод `parse`, позволяющий анализировать префикс введенных данных, оставляя все остальное непрочитанным.

Арифметический парсер можно запустить с помощью следующей команды:

```
$ scala ParseExpr "2 * (3 + 7)"
input: 2 * (3 + 7)
[1.12] parsed: ((2~List((*~(((~((3~List())~List((+
~(7~List())))))~)))))~List())
```

Если судить по выходным данным, то парсер успешно проанализировал строку ввода до позиции `[1.12]`. Это означает первую строку и двенадцатый столбец, иными словами, парсер обработал всю введенную строку. Результат, показанный после `parsed:`, мы пока проигнорируем. Пользы от него немного, а как получить от парсера более конкретные результаты, вы поймете чуть позже.

Можно также попробовать ввести строку с недопустимым выражением. Например, поставить одну лишнюю закрывающую скобку:

```
$ scala ParseExpr "2 * (3 + 7))"
input: 2 * (3 + 7))
[1.12] failure: '-' expected but ')' found

2 * (3 + 7))
           ^
```

В данном случае парсер `expr` обработает все вплоть до последней закрывающей скобки, которая не формирует часть арифметического выражения. Затем метод `parseAll` выдаст сообщение об ошибке, в котором будет сказано, что на месте закрывающей скобки ожидался оператор `-`. Чуть позже в этой главе мы объясним, почему было выдано именно такое сообщение об ошибке и как можно улучшить ситуацию.

33.3. Основные парсеры — регулярные выражения

В парсере арифметических выражений используется еще один парсер по имени `floatingPointNumber`. Этот парсер, унаследованный от супертрейта `Arith-JavaTokenParsers` распознает число с плавающей точкой в формате Java. А как поступить, когда нужно выполнить разбор чисел в формате, несколько отличающемся от формата Java? В такой ситуации можно применить *парсер — регулярное выражение*.

Замысел заключается в том, что в качестве парсера можно использовать любое регулярное выражение. Оно разбирает все строки, которым может соответствовать. В результате получается разобранная строка. Например, парсер — регулярное выражение, показанный в листинге 33.2, дает описание идентификаторов Java.

Листинг 33.2. Парсер — регулярное выражение для идентификаторов Java

```
object MyParsers extends RegexParsers {
  val ident: Parser[String] = """[a-zA-Z_]\w * """.r
}
```

Объект `MyParsers` в этом листинге — наследник трейта `RegexParsers`, в то время как `Arith` — наследник `JavaTokenParsers`. Имеющиеся в Scala комбинаторы парсеров располагаются в иерархии трейтов, целиком содержащейся в пакете `scala.util.parsing.combinator`. Трейтом на самом верхнем уровне является `Parsers`, и в нем определяется самый общий фреймворк синтаксического разбора для любого вида входных данных. На один уровень ниже располагается трейт `RegexParsers`, требующий, чтобы входные данные представляли собой последовательность символов, и предоставляемый для разбора с применением регулярных выражений. Еще более специализирован трейт `JavaTokenParsers`, реализующий парсеры для основных классов слов (или *токенов*) в соответствии с определением, используемым в Java.

33.4. Еще один пример: JSON

JSON (JavaScript Object Notation) — популярный формат обмена данными. В этом разделе мы покажем процесс создания парсера для этого формата. Грамматика, описывающая синтаксис JSON, выглядит так:

```
      value ::= obj | arr | stringLiteral |
                floatingPointNumber |
                "null" | "true" | "false" .
        obj ::= "{" [members] "}" .
        arr ::= "[" [values] "]" .
    members ::= member { "," member}.
     member ::= stringLiteral ":" value.
     values ::= value { "," value}.
```

JSON-значение является объектом, массивом, строкой, числом или одним из трех зарезервированных слов: `null`, `true` или `false`. JSON-объект является последовательностью (возможно, пустой) элементов, отделенных друг от друга запятыми и заключенных в круглые скобки. Каждый элемент является парой «строка — значение», где строка и значение отделены друг от друга двоеточием. И наконец, JSON-массив является последовательностью значений, отделенных друг от друга запятыми и заключенных в квадратные скобки. В качестве примера в листинге 33.3 содержится адресная книга, отформатированная в виде JSON-объекта.

Листинг 33.3. Данные в JSON-формате

```
{
  "address book": {
    "name": "John Smith",
    "address": {
       "street": "10 Market Street",
       "city" : "San Francisco, CA",
       "zip" : 94111
    },
    "phone numbers": [
      "408 338-4238",
      "408 111-6892"
    ]
  }
}
```

При использовании имеющихся в Scala комбинаторов парсеров выполнить разбор подобных данных не составляет особого труда. Полнофункциональный парсер показан в листинге 33.4. Он имеет ту же структуру, что и парсер арифметических выражений. Здесь снова применяется прямое отображение правил на грамматику JSON. В наборе правил применяется одно сокращение, упрощающее грамматику: комбинатор `repsep` выполняет разбор последовательности термов (возможно, пустой), отделенных друг от друга заданной строкой-разделителем. Например, в коде, показанном в листинге 33.4, `repsep(member, ",")` выполняет разбор последовательности, элементы которой, представленные термами, отделены друг от друга запятыми. В остальных случаях набор правил в парсере в точности соответствует набору правил в грамматике, как это было в случае с парсерами для арифметических выражений.

Листинг 33.4. Простой парсер формата JSON

```
import scala.util.parsing.combinator._

class JSON extends JavaTokenParsers {

  def value : Parser[Any] = obj | arr |
                            stringLiteral |
                            floatingPointNumber |
                            "null" | "true" | "false"
```

```
  def obj   : Parser[Any] = "{"~repsep(member, ",")~"}"

  def arr   : Parser[Any] = "["~repsep(value, ",")~"]"

  def member: Parser[Any] = stringLiteral~":"~value
}
```

Проверим JSON-парсеры в работе. Для этого немного изменим фреймворк, чтобы парсер работал с файлом, а не с командной строкой:

```
import java.io.FileReader

object ParseJSON extends JSON {
  def main(args: Array[String]) = {
    val reader = new FileReader(args(0))
    println(parseAll(value, reader))
  }
}
```

Метод `main` в этой программе сначала создает объект `FileReader`. Затем анализирует символы, возвращенные им в соответствии со значением правила грамматики JSON. Следует заметить, что `parseAll` и `parse` существуют в перегруженных вариантах: оба метода могут получать последовательность символов или дополнительно, в качестве второго аргумента, считыватель ввода.

Если объект «адресная книга», показанный в листинге 33.3, сохранить в файле с именем `address-book.json` и запустить в отношении этого файла программу `ParseJSON`, то будет получен следующий результат:

```
$ scala ParseJSON address-book.json
[13.4] parsed: (({~List(((("address book"~:)~(({~List(((
"name"~:)~"John Smith"), (("address"~:)~(({~List(((
"street"~:)~"10 Market Street"), (("city"~:)~"San Francisco
,CA"), (("zip"~:)~94111)))~})), (("phone numbers"~:)~(([~
List("408 338-4238", "408 111-6892"))~])))~})))~})
```

33.5. Вывод парсера

Программа `ParseJSON` успешно справляется с разбором адресной книги в формате JSON. Но вывод парсера все же выглядит странно. Он похож на последовательность, составленную из обрывков, склеенных вместе с помощью комбинаций из списков входных данных и символов ~. Пользы от него немного. Читателю с ним труднее разобраться, чем с входными данными, к тому же он слишком слабо организован, чтобы его легко проанализировал компьютер. Пора с этим что-нибудь сделать.

Чтобы понять, как поступить, сначала нужно узнать, что именно во фремворке комбинаторов возвращают в качестве результата отдельно взятые парсеры (в том

случае, если успешно выполнят разбор входных данных). Здесь действуют следующие правила.

1. Каждый парсер, записанный как строка (такая как `"{"`, или `":"`, или `"null"`), возвращает саму разобранную строку.
2. Парсер — регулярное выражение, такой как `"""[a-zA-Z_]\w*""".r`, также возвращает саму разобранную строку. То же самое справедливо и для таких парсеров — регулярных выражений, как `stringLiteral` или `floatingPointNumber`, наследуемых от трейта `JavaTokenParsers`.
3. Последовательная композиция `P~Q` возвращает результаты как `P`, так и `Q`. Эти результаты возвращаются в экземпляре `case`-класса, который также записывается как `~`. Следовательно, если `P` возвращает `"true"`, а `Q` возвращает `"?"`, то последовательная композиция `P~Q` возвращает `~("true", "?")`, что выводится как `(true~?)`.
4. Альтернативная композиция `P | Q` возвращает результат либо `P`, либо `Q`, в зависимости от того, какой из парсеров будет иметь успех.
5. Повторение `rep(P)` или `repsep(P, separator)` возвращает список результатов всех запусков `P`.
6. Опция `opt(P)` возвращает экземпляр типа `Option` Scala. Он возвращает `Some(R)`, если работа `P` завершится успешно с результатом `R`, и `None`, если `P` даст сбой.

Теперь, руководствуясь изложенными выше правилами, можно понять, *почему* вывод парсера появляется именно в том виде, который был показан в предыдущих примерах. Но более удобным его восприятие от этого не стало. Намного лучше было бы отобразить JSON-объект на внутреннее представление Scala, отображающее смысл JSON-значения. Следующее представление было бы более естественным.

- JSON-объект представляется как Scala-отображение типа `Map[String, Any]`. Каждый член представлен связкой «ключ — значение» в отображении.
- JSON-массив представляется как список Scala типа `List[Any]`.
- JSON-строка представляется как Scala `String`.
- Числовой литерал JSON представляется как Scala `Double`.
- Значения `true`, `false` и `null` представляются как одноименные значения Scala.

Чтобы получить такое представление, нужно воспользоваться еще одной формой комбинации парсеров — `^^`.

Оператор `^^` *преобразует* результат парсера. Выражения, использующие этот оператор, имеют вид `P ^^ f`, где `P` обозначает парсер, а `f` — функцию. `P ^^ f` разбирает те же предложения, что и `P`. Когда `P` возвращает некий результат `R`, результатом `P ^^ f` становится `f(R)`.

Рассмотрим в качестве примера парсер, который анализирует число с плавающей точкой и превращает его в значение Scala типа `Double`:

```
floatingPointNumber ^^ (_.toDouble)
```

А так выглядит парсер, анализирующий строку `"true"` и возвращающий булево Scala-значение `true`:

```
"true" ^^ (x => true)
```

Теперь перейдем к более сложным преобразованиям. Вот как выглядит новая версия парсера для JSON-объектов, возвращающих Scala `Map`:

```
def obj: Parser[Map[String, Any]] = // Этот код можно улучшить
  "{"~repsep(member, ",")~"}" ^^
    { case "{"~ms~"}" => Map() ++ ms }
```

Вспомним, что оператор ~ выдает в качестве результата экземпляр класса с таким же именем — ~. А вот как выглядит определение этого класса, являющегося внутренним классом трейта `Parsers`:

```
case class ~ [+A, +B](x: A, y: B) {
  override def toString = "(" + x + "~" + y + ")"
}
```

Одинаковые имена для класса и метода комбинатора последовательностей ~ выбраны не случайно. Это позволяет сопоставлять результаты парсера с паттернами, имеющими такую же структуру, что и сами парсеры. Например, паттерн `"{"~ms~"}"` соответствует строке результата `"{"`, за которой следует переменная результата `ms`, за которой, в свою очередь, следует строка результата `"}"`. Данный паттерн в точности соответствует тому, что возвращает парсер слева от ^^.

В его более открытых версиях, где сначала идет оператор ~, тот же паттерн считывает `~(~("{", ms), "}")`, но выглядит это менее понятно.

Паттерн `"{"~ms~"}"` требуется для избавления от фигурных скобок, чтобы можно было получить список элементов, являющийся результатом работы парсера `repsep(member, ",")`. В подобных случаях имеется альтернатива, позволяющая избегать выдачи ненужных результатов работы парсеров, которые немедленно отбрасываются сопоставлением с образцом. В этом альтернативном варианте используются комбинаторы парсеров ~> и <~. Оба они обозначают последовательную композицию, подобную ~, но ~> сохраняет только результаты своего правого операнда, а <~ — лишь результаты левого операнда. При использовании этих комбинаторов парсер JSON-объекта может быть выражен более кратко:

```
def obj: Parser[Map[String, Any]] =
"{"~> repsep(member, ",") <~"}" ^^ (Map() ++ _)
```

Полноценный парсер JSON-данных, возвращающий вполне выразительные результаты, показан в листинге 33.5.

Листинг 33.5. Полноценный JSON-парсер, возвращающий значимые результаты

```
import scala.util.parsing.combinator._

class JSON1 extends JavaTokenParsers {

  def obj: Parser[Map[String, Any]] =
    "{"~> repsep(member, ",") <~"}" ^^ (Map() ++ _)
```

```
  def arr: Parser[List[Any]] =
    "["~> repsep(value, ",") <~"]"

  def member: Parser[(String, Any)] =
    stringLiteral~":"~value ^^
      { case name~":"~value => (name, value) }

  def value: Parser[Any] = (
    obj
  | arr
  | stringLiteral
  | floatingPointNumber ^^ (_.toDouble)
  | "null"  ^^ (x => null)
  | "true"  ^^ (x => true)
  | "false" ^^ (x => false)
  )
}
```

Если запустить этот парсер в отношении файла `address-book.json`, то будет (после добавления ряда символов новой строки и отступов) получен следующий результат:

```
$ scala JSON1Test address-book.json
[14.1] parsed: Map(
  address book -> Map(
    name -> John Smith,
    address -> Map(
      street -> 10 Market Street,
      city -> San Francisco, CA,
      zip -> 94111),
    phone numbers -> List(408 338-4238, 408 111-6892)
  )
)
```

Вот это, в общем-то, и нужно знать, чтобы приступить к написанию собственных парсеров. В качестве шпаргалки в табл. 33.1 перечислены все рассмотренные до сих пор комбинаторы парсеров.

Таблица 33.1. Сводка комбинаторов парсеров

Комбинатор парсеров	Определение
"..."	Литерал
"...".r	Регулярное выражение
P~Q	Последовательная композиция
P <~ Q, P ~> Q	Последовательная композиция; сохраняет только левую или правую часть
P | Q	Альтернатива
opt(P)	Опция
rep(P)	Повторение

Комбинатор парсеров	Определение
repsep(P, Q)	Чередующееся повторение
P ^^ f	Преобразование результата

Сравнение символьных и буквенно-цифровых имен

Для многих парсер-комбинаторов, указанных в табл. 33.1, используются символьные имена. У такого подхода есть как преимущества, так и недостатки. К последним относится то, что символьные имена нужно запоминать. Пользователи, не знакомые со Scala-библиотеками синтаксического разбора с применением комбинаторов, скорее всего, не поймут значения ~, ~> или ^^. К положительной стороне можно отнести краткость символьных имен и возможность правильно выбрать их уровень приоритета и ассоциативность. Например, комбинаторы парсеров ~, ^^ и | выбраны преднамеренно в порядке убывания уровня приоритета. Обычное грамматическое правило состоит из альтернатив, имеющих часть парсера и часть преобразования. Часть парсера обычно содержит несколько последовательных элементов, разделенных операторами ~. С выбранными уровнями приоритета ~, ^^ и | можно записать такое грамматическое правило, не нуждаясь в расстановке круглых скобок.

Отключение неявных точек с запятыми

Заметьте, что тело парсера `value` в листинге 33.5 заключено в круглые скобки. Этот прием позволяет отключить неявные точки с запятыми в выражениях парсеров. В разделе 4.2 мы показали, что Scala подразумевает наличие между двумя строками кода точки с запятой, которая синтаксически может выступать в качестве разделителя инструкций, если только первая строка не заканчивается инфиксным оператором или две строки не заключены в круглые или квадратные скобки. Теперь можно ставить оператор | в конце каждого альтернативного варианта, а не начинать вариант с него:

```
def value: Parser[Any] =
  obj |
  arr |
  stringLiteral |
  ...
```

В таком случае не понадобятся круглые скобки вокруг тела парсера `value`. Однако некоторые предпочитают видеть оператор | в начале второго альтернативного варианта, а не в конце первого. Обычно это приводит к нежелательной вставке между двумя строками точки с запятой:

```
  obj; // Здесь подразумевается точка с запятой
| arr
```

Точка с запятой изменяет структуру кода, вызывая сбой компиляции. Заключение всего выражения в круглые скобки позволяет убрать точку с запятой и добиться правильной компиляции кода.

Более того, символьные операторы визуально занимают меньше места, чем буквенно-цифровые. Для парсера это важно, поскольку позволяет сконцентрироваться на исходной грамматике, а не на самих комбинаторах. Чтобы увидеть разницу, представьте на минуту, что последовательная композиция (~) была названа `andThen`, а альтернативный вариант (|) получил имя `orElse`. Тогда парсер арифметического выражения, показанный в листинге 33.1, приобрел бы следующий вид:

```
class ArithHypothetical extends JavaTokenParsers {
  def expr: Parser[Any] =
    term andThen rep(("+" andThen term) orElse
                     ("-" andThen term))
  def term: Parser[Any] =
    factor andThen rep((" * " andThen factor) orElse
                       ("/" andThen factor))
  def factor: Parser[Any] =
    floatingPointNumber orElse
    ("(" andThen expr andThen ")")
}
```

Как видите, код стал намного длиннее и в нем труднее «заметить» грамматику среди всех этих операторов и круглых скобок. С другой стороны, новичкам разбора с применением комбинаторов, вероятно, будет проще разобраться в том, что именно делается с помощью данного кода.

33.6. Реализация комбинаторов парсеров

В предыдущем разделе было показано, что имеющиеся в Scala комбинаторы парсеров предоставляют весьма удобные средства для создания ваших собственных парсеров. Представляя собой не что иное, как библиотеку Scala, они хорошо вписываются в ваши программы на Scala. Таким образом, очень легко объединить парсер с кодом, обрабатывающим поставляемые им результаты, или же выстроить парсер так, чтобы он получал свои данные из какого-то определенного источника (скажем, из файла, строки или массива символов).

Каким образом это достигается? Далее в главе вы сможете «заглянуть под капот» библиотеки комбинаторов парсеров. Вы увидите, что представляет собой парсер и как реализуются уже встречавшиеся элементарные парсеры и комбинаторы парсеров. Если ваши желания ограничиваются написанием простых комбинаторов парсеров, то эту часть главы можете спокойно пропустить. Однако, дочитав главу до конца, вы можете получить более глубокое представление о комбинаторах парсеров в частности и о принципах разработки использующего комбинаторы предметно-ориентированного языка в целом.

Выбор между символьными и буквенно-цифровыми именами

В качестве руководства по выбору между символьными и буквенно-цифровыми именами мы даем следующие рекомендации.

- Используйте символьные имена в тех случаях, когда у них уже есть устоявшееся универсальное значение. Например, никто не станет рекомендовать для сложения чисел запись вида `add` вместо `+`.
- В противном случае отдавайте предпочтение алфавитно-цифровым именам, если хотите, чтобы ваш код был понятен случайным читателям.
- Для предметно-ориентированных библиотек можно выбирать символьные имена, если они дают явное преимущество в удобочитаемости и не ожидается, что случайный читатель без основательной подготовки в предметной области сможет с ходу разобраться в коде.

Если искомые комбинаторы парсеров находятся в предметно-ориентированном языке более высокого уровня, то у случайных читателей могут возникнуть проблемы с пониманием даже при использовании буквенно-цифровых имен. Кроме того, специалистам символьные имена предоставляют явные преимущества, заключающиеся в удобочитаемости кода. Поэтому их применение в данном приложении мы считаем вполне оправданным.

Ядро фреймворка комбинаторов парсеров Scala содержится в трейте `scala.util.parsing.combinator.Parsers`. Он определяет тип `Parser`, как и все фундаментальные комбинаторы. Кроме тех мест, где это специально оговаривается, определения, рассматриваемые в следующих двух подразделах, находятся в данном трейте. То есть подразумевается, что они содержатся в определении трейта, которое начинается со следующего кода:

```
package scala.util.parsing.combinator
trait Parsers {
  ... // Код находится здесь, если не указано иное
}
```

По сути, `Parser` является функцией от некоторого типа входных данных к результату синтаксического разбора. В первом приближении тип можно записать так:

```
type Parser[T] = Input => ParseResult[T]
```

Входные данные парсера

Иногда парсер считывает поток токенов, а не простую последовательность символов. Тогда для преобразования потока обычных символов в поток токенов

используется отдельный лексический анализатор. Тип входных данных парсера определяется следующим образом:

```
type Input = Reader[Elem]
```

Класс `Reader` берется из пакета `scala.util.parsing.input`. Он похож на класс `Stream`, но вдобавок ко всему отслеживает позиции всех считываемых элементов. Отдельные входные элементы представлены типом `Elem`. Это член абстрактного типа трейта `Parsers`:

```
type Elem
```

Это значит, подклассам и подтрейтам `Parsers` необходимо создавать экземпляр класса `Elem` для типа входных анализируемых парсером элементов. Например, `RegexParsers` и `JavaTokenParsers` определяют `Elem` эквивалентом `Char`. Но можно было бы также установить для `Elem` какой-либо другой тип, а именно тип токенов, возвращаемых отдельным лексическим анализатором (лексером).

Результаты парсера

Парсер может завершить работу с какими-либо входными данными либо успешно, либо со сбоем. Следовательно, для представления успеха и сбоя у класса `ParseResult` есть два подкласса:

```
sealed abstract class ParseResult[+T]
case class Success[T](result: T, in: Input)
  extends ParseResult[T]
case class Failure(msg: String, in: Input)
  extends ParseResult[Nothing]
```

`Case`-класс `Success` переносит результат, возвращенный парсером, в свой параметр `result`. Тип у результатов парсера произвольный, именно поэтому и `ParseResult`, и `Success`, и `Parser` параметризованы с параметром типа `T`. Параметр типа представляет разновидность результата, возвращенного данным парсером. `Success` получает также второй параметр, `in`, ссылающийся на ту часть входных данных, которая следует сразу же за частью, поглощенной парсером. Это поле необходимо для парсеров, выстроенных в цепочку, чтобы один из них мог работать после другого. Обратите внимание: это чисто функциональный подход к разбору. Входные данные не считываются в виде побочного эффекта, но при этом сохраняются в потоке. Парсер анализирует некоторую часть потока входных данных, после чего возвращает оставшуюся часть в своем результате.

Еще одним подклассом `ParseResult` является `Failure`. Он получает в качестве параметра сообщение, в котором указана причина сбоя. Как и `Success`, класс `Failure` получает в качестве второго параметра оставшуюся часть потока входных данных. Это нужно не для цепочки парсеров (парсер не станет продолжать работу после сбоя), а чтобы поместить сведения о месте сбоя в потоке входных данных в сообщение об ошибке. Заметьте, что результаты парсера определены в пара-

метре `T` как ковариантные. То есть парсер, возвращающий, к примеру, в качестве результата значения типа `String`, совместим с парсером, который возвращает значения типа `AnyRef`.

Класс Parser

Предыдущая характеристика парсеров как функций от входных данных к результатам парсера давала слишком упрощенное представление о них. Приведенные выше примеры показывают, что в парсерах реализуются также *методы* `~` для последовательной композиции двух парсеров и `|` — для их альтернативной композиции. Следовательно, на самом деле `Parser` — класс, наследуемый от функционального типа `Input => ParseResult[T]` и дополнительно определяющий эти методы:

```
abstract class Parser[+T] extends (Input => ParseResult[T])
{ p =>
  // Произвольный метод, определяющий
  // поведение парсера
  def apply(in: Input): ParseResult[T]

  def ~ ...
  def | ...
  ...
}
```

Парсеры являются функциями (то есть наследуются от них), поэтому нуждаются в определении метода `apply`. Абстрактный метод `apply` можно увидеть в классе `Parser`, однако он предназначен только для документирования, как и любой такой метод, в любом случае наследуемый от типа `Input => ParseResult[T]` (вспомним, что данный тип есть сокращенная форма записи от `scala.Function1[Input, ParseResult[T]]`). Метод `apply` все же нуждается в реализации в отдельно взятых парсерах, являющихся наследниками абстрактного класса `Parser`. Эти парсеры мы рассмотрим после следующего раздела, который посвящен использованию псевдонимов `this`.

Псевдонимы this

Тело класса `Parser` начинается с весьма странного выражения:

```
abstract class Parser[+T] extends ... { p =>
```

Таким спецификатором, как `id =>`, который стоит сразу же после открывающей фигурной скобки шаблона класса, идентификатор `id` определяется в качестве псевдонима для `this` в данном классе. Это равнозначно записи:

```
val id = this
```

в теле класса, за исключением того, что компилятор Scala знает, что `id` является псевдонимом для `this`. Например, можно обратиться к приватному элементу

объекта m данного класса, воспользовавшись либо id.m, либо this.m — эти обращения абсолютно эквивалентны. Первое выражение не скомпилируется, если id был просто определен как val-переменная с ключевым словом this, указанным в правой части, поскольку в этом случае компилятор Scala будет рассматривать id в качестве обычного идентификатора.

Подобный синтаксис уже встречался в разделе 29.4, где с его помощью трейту предоставлялся собственный тип. Использование псевдонимов также может послужить неплохой сокращенной записью, когда понадобится обратиться с помощью this к охватывающему классу. Рассмотрим пример:

```
class Outer { outer =>
  class Inner {
    println(Outer.this eq outer) // Выводит true
  }
}
```

В нем определяются два вложенных класса: Outer и Inner. Внутри Inner на значение this класса Outer имеется две ссылки с использованием разных выражений. В первом выражении показан способ, применяемый в Java: перед зарезервированным словом this можно поставить имя внешнего класса и точку, и такое выражение затем сошлется на this внешнего класса. Во втором выражении показана альтернатива, предоставляемая языком Scala. За счет введения псевдонима, названного outer для this в классе Outer, появляется возможность ссылаться на псевдоним this непосредственно во внутренних классах. Способ, предлагаемый в Scala, гораздо лаконичнее и может повысить удобочитаемость кода при удачном выборе псевдонима. Примеры применения таких псевдонимов будут показаны чуть позже.

Парсеры отдельных токенов

В трейте Parsers определен универсальный парсер elem, который может использоваться для анализа любого отдельно взятого токена:

```
def elem(kind: String, p: Elem => Boolean) =
  new Parser[Elem] {
    def apply(in: Input) =
      if (p(in.first)) Success(in.first, in.rest)
      else Failure(kind + " expected", in)
  }
```

Этот парсер получает два параметра: строку kind, являющуюся описанием разновидности анализируемого токена, и применяемый к Elem-объектам предикат p, который указывает, соответствует ли элемент классу токенов, подлежащих разбору.

Когда парсер elem(kind, p) применяется к какому-либо вводу in, первый элемент потока входных данных тестируется предикатом p. Если p возвращает true,

то работа парсера завершается успешно. Его результатом являются сам элемент и оставшийся поток входных данных, который начинается сразу же после обработанного парсером элемента. Если же p возвращает false, то парсер дает сбой с сообщением об ошибке, указывающей, какого рода токен ожидался.

Последовательная композиция

Парсер elem использует только отдельно взятый элемент. Чтобы разобрать более содержательные фразы, можно связать парсеры с помощью оператора последовательной композиции ~. Ранее мы уже показали, что P~Q — парсер, который сначала применяет к заданной входной строке парсер P. Затем, если работа P завершится успешно, к входным данным, оставшимся после того, как свою работу выполнил парсер P, применяется парсер Q.

Комбинатор ~ реализован в виде метода класса Parser. Его определение показано в листинге 33.6. Метод является членом класса Parser. Внутри этого класса p указан частью p => в качестве псевдонима this, следовательно, p обозначает левый операнд (или получатель) оператора ~. Его правый операнд представлен параметром q. Теперь, если последовательность парсеров p~q будет запущена в отношении каких-либо входных данных in, то сначала в отношении in будет запущен p и результатом станет анализ в сопоставлении с образцом. При успешном завершении работы p в отношении оставшихся входных данных in1 будет запущен парсер q. Если и его работа завершится успешно, то успех будет сопутствовать всему парсеру. Его результатом станет ~-объект, содержащий как результат парсера p (то есть x), так и результат парсера q (то есть y). Если же или p, или q даст сбой, то результатом p~q станет объект Failure, возвращенный либо p, либо q.

Листинг 33.6. Метод-комбинатор ~

```
abstract class Parser[+T] ... { p =>
  ...
  def ~ [U](q: => Parser[U]) = new Parser[T~U] {
    def apply(in: Input) = p(in) match {
      case Success(x, in1) =>
        q(in1) match {
          case Success(y, in2) => Success(new ~(x, y), in2)
          case failure => failure
        }
    case failure => failure
  }
}
```

Результирующим типом ~ является парсер, возвращающий экземпляр case-класса ~ с элементами типов T и U. Выражение типа T~U — просто более удобочитаемая сокращенная форма записи для параметризованного типа ~[T, U]. Обычно Scala интерпретирует операции бинарного типа наподобие A op B как

параметризованный тип `op[A, B]`. Это аналогично ситуации для паттернов, где бинарный паттерн `P op Q` также интерпретируется как применение, то есть `op(P, Q)`.

Остальные два оператора последовательных композиций, `<~` и `~>`, могут быть определены точно так же, как и `~`, только с небольшим уточнением порядка вычисления результата. Но более изящно будет определить их в понятиях `~` следующим образом:

```
def <~ [U](q: => Parser[U]): Parser[T] =
  (p~q) ^^ { case x~y => x }
def ~> [U](q: => Parser[U]): Parser[U] =
  (p~q) ^^ { case x~y => y }
```

Альтернативная композиция

В альтернативной композиции `P | Q` к заданным входным данным применяется либо `P`, либо `Q`. Сначала предпринимается попытка использования `P`. При успешном завершении работы `P` завершается и работа всего парсера с выдачей результата `P`. Но если `P` даст сбой, то предпринимается попытка применить в отношении *тех же входных данных* парсер `Q`. Затем результат выполнения `Q` становится результатом выполнения всего парсера.

Определение метода `|` класса `Parser` выглядит следующим образом:

```
def | (q: => Parser[T]) = new Parser[T] {
  def apply(in: Input) = p(in) match {
    case s1 @ Success(_, _) => s1
    case failure => q(in)
  }
}
```

Следует заметить, что при сбое обоих парсеров, и `P`, и `Q`, сообщение об ошибке определяется парсером `Q`. Этот неочевидный выбор мы рассмотрим чуть позже, в разделе 33.9.

Работа с рекурсией

Обратите внимание: параметр `q` в методах `~` и `|` является передаваемым по имени параметром — перед его типом стоит стрелка `=>`. То есть фактический аргумент парсера будет вычислен лишь в том случае, если понадобится `q`, а это случится только после запуска `p`. Благодаря этому появляется возможность создать рекурсивные парсеры, похожие на следующий, который выполняет разбор числа, заключенного в произвольное количество скобок:

```
def parens = floatingPointNumber | "("~parens~")"
```

Если `|` и `~` получили *параметры, передаваемые по значению*, то это определение тут же вызовет переполнение стека, еще ничего не считывая, поскольку значение `parens` оказалось в середине его правой части.

Преобразование результата

Последний метод класса `Parser` преобразует результат парсера. Парсер `P ^^ f` завершает свою работу успешно при успешном завершении работы парсера `P`. В таком случае данный парсер возвращает результат парсера `P`, преобразованный с помощью функции `f`. Реализация этого метода выглядит следующим образом:

```
  def ^^ [U](f: T => U): Parser[U] = new Parser[U] {
    def apply(in: Input) = p(in) match {
      case Success(x, in1) => Success(f(x), in1)
      case failure => failure
    }
  }
} // Завершение Parser
```

Парсеры, не считывающие данных

Существует еще два парсера, которые не используют никаких входных данных: `success` и `failure`. Парсер `success(result)` всегда завершает работу успешно с заданным результатом `result`. Парсер `failure(msg)` всегда дает сбой с выдачей сообщения об ошибке `msg`. Оба парсера реализованы в виде методов в трейте `Parsers`, во внешнем трейте, который также содержит класс `Parser`:

```
def success[T](v: T) = new Parser[T] {
  def apply(in: Input) = Success(v, in)
}
def failure(msg: String) = new Parser[Nothing] {
  def apply(in: Input) = Failure(msg, in)
}
```

Option и повторение

В трейте `Parsers` также реализованы комбинаторы `Option` и повторения `opt`, `rep` и `repsep`. Они реализованы через последовательные и альтернативные композиции и преобразования результата:

```
  def opt[T](p: => Parser[T]): Parser[Option[T]] = (
    p ^^ Some(_)
  | success(None)
  )

  def rep[T](p: => Parser[T]): Parser[List[T]] = (
    p~rep(p) ^^ { case x~xs => x :: xs }
  | success(List())
  )

  def repsep[T](p: => Parser[T],
      q: => Parser[Any]): Parser[List[T]] = (
```

```
    p~rep(q ~> p) ^^ { case r~rs => r :: rs }
  | success(List())
  )
} // Завершение Parsers
```

33.7. Строковые литералы и регулярные выражения

Показанные до сих пор парсеры использовали для разбора отдельных слов строковые литералы и регулярные выражения. Поддержка этих возможностей исходит из `RegexParsers`, подтрейта `Parsers`:

```
trait RegexParsers extends Parsers {
```

Этот трейт имеет более специализированный характер, чем трейт `Parsers`, в том смысле, что работает только с входными данными, представляющими собой последовательность символов:

```
type Elem = Char
```

В нем определены два метода, `literal` и `regex`, имеющие такие сигнатуры:

```
implicit def literal(s: String): Parser[String] = ...
implicit def regex(r: Regex): Parser[String] = ...
```

Заметьте, что у обоих методов есть модификатор `implicit`, следовательно, они автоматически применяются там, где задаются значения типа `String` или `Regex`, но ожидается значение `Parser`. Именно поэтому вы можете непосредственно в грамматике записывать строковые литералы и регулярные выражения, не прибегая к необходимости заключать их в один из этих методов. Например, парсер `"("~expr~")"` будет автоматически развернут в `literal("(")~expr~literal(")")`.

В трейте `RegexParsers` также предусмотрена обработка пробельных символов. Для этого перед запуском парсера `literal` или `regex` в трейте вызывается метод по имени `handleWhiteSpace`. Он пропускает самую длинную последовательность входных данных, соответствующую регулярному выражению `whiteSpace`, которое по умолчанию определено так:

```
  protected val whiteSpace = """\s+""".r
} // Завершение RegexParsers
```

Если предпочтение отдается иной трактовке пробельных символов, то значение `val`-переменной `whiteSpace` можно переписать. Например, если нужно, чтобы пробельные символы вообще не пропускались, `whiteSpace` можно переписать в виде пустого регулярного выражения:

```
object MyParsers extends RegexParsers {
  override val whiteSpace = "".r
  ...
}
```

33.8. Лексинг и парсинг

Решение задачи выполнения синтаксического анализа зачастую разбивается на две фазы. В фазе *лексера* во входных данных распознаются отдельные слова и определяется их принадлежность к некоторым классам *токенов*. Эта фаза называется также *лексическим анализом*. За ней следует фаза *синтаксического анализа*, в которой анализируется последовательность токенов. Синтаксический анализ иногда называют просто разбором, хотя это не совсем верно, поскольку лексический анализ тоже может рассматриваться как задача разбора.

В соответствии с описанием, рассмотренным в предыдущем разделе, трейт `Parsers` может использоваться в любой из фаз, поскольку его входные элементы принадлежат абстрактному типу `Elem`. В целях лексического анализа из данных типа `Elem` могут создаваться экземпляры типа `Char`, означающие отдельные символы, которые составляют разбираемое слово. В свою очередь, для синтаксического анализа из данных типа `Elem` могут создаваться экземпляры типа токенов, возвращаемых лексером.

Комбинаторы парсеров Scala предоставляют для лексического и синтаксического анализа несколько полезных классов. Они содержатся в двух подпакетах отдельно для каждой разновидности анализа:

```
scala.util.parsing.combinator.lexical
scala.util.parsing.combinator.syntactical
```

Если нужно разбить парсер отдельно на лексер и синтаксический анализатор, то можно обратиться к документации по этим пакетам в Scaladoc. Но что касается простых парсеров, то обычно бывает достаточно воспользоваться ранее показанным в данной главе подходом на основе регулярных выражений.

33.9. Сообщения об ошибках

Есть еще одна последняя и пока не рассмотренная тема: как парсер выдает сообщение об ошибке? Выдача сообщений об ошибках для парсеров сродни черной магии. Одна из проблем заключается в том, что парсер, отвергая входные данные, обычно обнаруживает множество различных недочетов. Каждый альтернативный синтаксический анализ должен иметь неудачный исход, и рекурсивно это случается в любой точке выбора. Тогда какой именно из многочисленных сбоев должен выдаваться пользователю в сообщении об ошибке?

В имеющейся в Scala библиотеке парсинга реализовано простое эвристическое правило: среди всех сбоев выбирается тот, который произошел на последней позиции входных данных. Иными словами, парсер выбирает самый длинный приемлемый префикс и выдает сообщение об ошибке с описанием того, почему синтаксический анализ префикса не может быть продолжен. Если в последней позиции есть сразу несколько точек сбоя, то выбирается та из них, к которой было последнее обращение.

Рассмотрим, к примеру, работу JSON-парсера над дефектной адресной книгой, которая начинается со следующей строки:

```
{ "name": John,
```

Самый длинный приемлемый префикс в этой фразе имеет вид `{ "name":`. Следовательно, JSON-парсер поставит флажок, отмечая ошибку, на имени `John`. В данном месте JSON-парсер ожидал значение, но `John` — идентификатор, не содержащий значения (по-видимому, автор документа забыл заключить имя в кавычки). Сообщение об ошибке, выданное парсером для этого документа, будет выглядеть так:

```
[1.13] failure: "false" expected but identifier John found
  { "name": John,
            ^
```

Предполагаемая сбойная часть берется на основе того факта, что `"false"` в грамматике JSON — последний альтернативный вариант правила для `value`. Следовательно, данный сбой и выступил в качестве последнего в этом месте. Пользователи, разбирающиеся в тонкостях грамматики JSON, могут реконструировать сообщение об ошибке, но у тех, кто в них не разбирается, данное сообщение может, вероятно, вызвать удивление, и вдобавок ввести их в заблуждение.

Более полезное сообщение об ошибке может быть создано с помощью добавления в последнюю альтернативу правила `value`, обобщающей точки сбоя:

```
def value: Parser[Any] =
  obj | arr | stringLit | floatingPointNumber | "null" |
  "true" | "false" | failure("illegal start of value")
```

Это добавление не изменяет множество входных данных, приемлемых в качестве допустимых документов. Оно просто повышает полезность сообщений об ошибках, поскольку теперь явным образом добавляется сбой, получающийся в последнем альтернативном варианте и потому отображаемый в отчете об ошибке:

```
[1.13] failure: illegal start of value
  { "name": John,
            ^
```

Чтобы пометить сбой, произошедший в последней позиции входных данных, в реализации последней возможной схемы отчета об ошибках используется поле `lastFailure` в трейте `Parsers`:

```
var lastFailure: Option[Failure] = None
```

Поле инициализируется значением `None`. Оно обновляется в конструкторе класса `Failure`:

```
case class Failure(msg: String, in: Input)
    extends ParseResult[Nothing] {

  if (lastFailure.isDefined &&
      lastFailure.get.in.pos <= in.pos)
    lastFailure = Some(this)
}
```

Значение поля считывается методом `phrase`, который при сбое парсера выдает конечное сообщение об ошибке. Реализация метода `phrase` в трейте `Parsers` выглядит следующим образом:

```
def phrase[T](p: Parser[T]) = new Parser[T] {
  lastFailure = None
  def apply(in: Input) = p(in) match {
    case s @ Success(out, in1) =>
      if (in1.atEnd) s
      else Failure("end of input expected", in1)
    case f : Failure =>
      lastFailure
  }
}
```

Метод `phrase` запускает свой аргумент — парсер `p`. При успешном завершении работы `p` и полном использовании входных данных возвращается успешный результат работы парсера `p`. Если работа `p` завершается успешно, но входные данные считаны не полностью, то выдается сообщение о сбое с формулировкой end of input expected («ожидалось окончание входных данных»). Если работа `p` завершается сбоем, то возвращаются сведения о сбое или ошибке, сохраненные в поле `lastFailure`. Следует заметить, что подход, примененный при реализации поля `lastFailure`, нефункционален — его значение обновляется конструктором класса `Failure` и самим методом `phrase` благодаря побочному эффекту. Можно было бы создать и функциональную версию той же самой схемы, но для нее потребовалось бы распараллелить значение `lastFailure` на каждый результат парсера независимо от того, был бы результат `Success` или `Failure`.

33.10. Сравнение отката с LL(1)

Выбор между альтернативными парсерами комбинаторы парсеров осуществляется с помощью *отката*. Если фигурирующий в выражении `P | Q` парсер `P` дает сбой, то запускается парсер `Q`, который использует те же входные данные, что и `P`. Это происходит, даже если до сбоя `P` уже выполнил разбор некоторых токенов. В таком случае те же самые токены пройдут разбор еще раз, но уже с использованием парсера `Q`.

Чтобы сохранить возможности разбора, откат накладывает на способ формулировки грамматики некоторые ограничения. По сути, нужно всего лишь избегать леворекурсивных правил. Такой набор правил, как:

```
expr ::= expr "+" term | term.
```

всегда будет приводить к сбою, поскольку выражение `expr` тут же вызывает само себя и поэтому никогда не продвигается дальше[1]. В то же время откат потенциально

[1] Существуют способы, позволяющие избегать переполнения стека даже при наличии леворекурсивного правила, но они требуют более сложного фреймворка комбинаторов парсеров, который на данный момент еще не реализован.

затратен, так как одни и те же входные данные можно проанализировать несколько раз. Рассмотрим, к примеру, следующий набор правил:

```
expr ::= term "+" expr | term.
```

Что произойдет, если парсер `expr` применяется к таким входным данным, как `(1 + 2) * 3`, представляющим собой вполне легальный терм? Сначала будет применен первый вариант альтернативы, который даст сбой при поиске соответствия знаку `+`. Затем в отношении того же самого терма будет применен второй альтернативный вариант, работа которого завершится успешно. В итоге терм окажется разобранным дважды.

Зачастую грамматику можно изменить таким образом, чтобы получить возможность избавиться от отката. Например, в случае арифметических выражений сработает одно из двух правил:

```
expr ::= term [ "+" expr].
expr ::= term { "+" term}.
```

Многие языки допускают использование так называемых LL(1)-грамматик[1]. Когда комбинатор парсеров сформирован на основе такой грамматики, откаты исключены, то есть позиция входа никогда не будет заново установлена на более раннее значение. Например, к LL(1) относятся показанные ранее в этой главе грамматики и для арифметических выражений, и для термов JSON, и потому возможность отката для входных данных из этих языков в фреймворке комбинаторов парсеров не используется никогда.

Работа фреймворка комбинаторов парсеров позволяет вам выразить предположение о принадлежности грамматики к LL(1) явным образом, используя новый оператор `~!`. Этот оператор похож на оператор последовательной композиции `~`, но никогда не приводит к откату к непрочитанным элементам входных данных, которые уже были разобраны. При использовании данного оператора правила в парсере арифметических выражений могут быть переписаны следующим образом:

```
def expr : Parser[Any] =
  term ~! rep("+" ~! term | "-" ~! term)
def term : Parser[Any] =
  factor ~! rep(" * " ~! factor | "/" ~! factor)
def factor: Parser[Any] =
  "(" ~! expr ~! ")" | floatingPointNumber
```

Одно из преимуществ LL(1)-парсера заключается в том, что он может использовать более простую технику обработки входных данных. Эти данные могут быть считаны последовательно, и входные элементы после считывания могут быть отброшены. В этом заключается еще одна причина, по которой LL(1)-парсеры обычно более эффективны по сравнению с парсерами, использующими откаты.

---

[1] *Aho A. V., Sethi R., Ullman J.* Compilers: Principles, Techniques and Tools. — Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 1986.

Резюме

Теперь вы узнали обо всех наиболее важных элементах фреймворка комбинаторов парсеров в Scala. Для такого по-настоящему полезного средства в них содержится на удивление мало кода. Используя фреймворк, можно создавать парсеры для большого класса контекстно-свободных грамматик. Фреймворком легко начать пользоваться, а также его легко настроить под новые разновидности грамматик и методы ввода данных. Будучи библиотекой Scala, этот фреймворк хорошо взаимодействует с языком в целом. Таким образом, комбинаторы парсеров легко интегрируются в более объемные программы.

Один из недостатков комбинаторов парсеров состоит в их невысокой эффективности, по крайней мере по сравнению с парсерами, создаваемыми с применением таких средств специального назначения, как Yacc или Bison. Причин здесь две. Первая — невысокая эффективность метода отката, используемого комбинаторами парсеров. Из-за повторяющихся откатов в зависимости от структуры грамматики и входных данных парсера скорость работы может экспоненциально снижаться. Проблему можно устранить путем приведения грамматики к виду LL(1) и использования преимущественно оператора последовательной композиции `~!`.

Вторая проблема, отрицательно влияющая на производительность комбинаторов парсеров, — присущее им смешивание конструкции парсера и анализа входных данных в одном наборе операций. По сути, для каждого анализируемого входа парсер создается заново.

С данной проблемой можно справиться, но для этого потребуется иная реализация фреймворка комбинаторов парсеров. В оптимизирующем фреймворке парсер не был бы представлен функцией от входных данных к результатам разбора. Вместо этого он был бы представлен деревом, где каждый шаг разбора был бы представлен `case`-классом. Например, последовательная композиция может быть представлена `case`-классом `Seq`, альтернативная — классом `Alt` и т. д. Внешний метод разбора, `phrase`, тогда мог бы получить символьное представление парсера и превратить его с помощью стандартных алгоритмов генерации парсеров в высокоэффективные таблицы разбора.

Самое приятное здесь то, что, с точки зрения пользователя, по сравнению с применением обычных комбинаторов парсеров ничего не изменится. Пользователи попрежнему будут создавать парсеры в терминах `ident`, `floatingPointNumber`, `~`, `|` и т. д. Им не понадобится вникать в то, что эти методы создают символьное представление парсера, а не функцию парсера. Поскольку `phrase`-комбинатор превращает эти представления в реальные парсеры, то все будет работать как и прежде.

Что касается производительности, при такой схеме будет получено двойное преимущество. Во-первых, теперь любую конструкцию парсера можно будет вынести за скобки анализа входных данных. Если бы нужно было воспользоваться таким кодом:

```
val jsonParser = phrase(value)
```

а затем применить `jsonParser` к нескольким различным входным данным, то `jsonParser` создавался бы только один раз, а не при каждом считывании входных данных.

Во-вторых генерация парсера могла бы использовать эффективные алгоритмы разбора, такие как LALR(1)[1]. Эти алгоритмы обычно позволяют существенно ускорить разбор, по сравнению с разбором с откатом.

На данный момент такой оптимизированный генератор парсеров для Scala еще не создан. Но возможность его создания все же существует. Если кто-нибудь сделает такой генератор, то его можно будет легко интегрировать в стандартную библиотеку Scala. Если даже предположить, что в будущем такой генератор появится, то повод для сохранения текущего фреймворка комбинаторов парсеров все же останется. Его намного проще понять и настроить, чем генератор парсеров, а разница в скорости работы на практике зачастую не играет существенной роли, если только вы не собираетесь выполнять синтаксический разбор очень большого объема входных данных.

[1] *Aho A. V., Sethi R., Ullman J.* Compilers: Principles, Techniques and Tools. — Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 1986.

34 Программирование GUI

В этой главе вы научитесь разрабатывать приложения, применяющие графический пользовательский интерфейс (graphical user interface, GUI) на Scala. Приложения, которые мы разработаем, основаны на библиотеке Scala, предоставляющей доступ к классам GUI фреймворка Java Swing. Концептуально библиотека Scala похожа на положенные в ее основу Swing-классы, но при этом она скрывает большинство их сложных моментов. Благодаря этому вы поймете, что разрабатывать GUI-приложения с помощью данного фреймворка на самом деле легко.

Даже при характерных для Scala упрощениях фреймворк, подобный Swing, перенасыщен всевозможными классами, в каждом из которых определено множество методов. Найти свой путь в столь богатой библиотеке поможет использование IDE-среды, такой как Eclipse с расширением для Scala. Очевидное ее преимущество заключается в том, что среда IDE может в интерактивном режиме выполнения своих команд показать, какие классы доступны в пакете и какие методы доступны объектам, на которые вы ссылаетесь. Это существенно ускорит освоение ранее неизвестного библиотечного пространства.

34.1. Первое Swing-приложение

Первым Swing-приложением, с которого мы начнем работу, станет окно, содержащее одну кнопку. Для программирования с использованием Swing нужно будет импортировать из пакета Swing API Scala различные классы:

```
import scala.swing._
```

Код вашего первого Swing-приложения на Scala показан в листинге 34.1. Если скомпилировать этот файл и запустить его на выполнение, то вы увидите окно, показанное в левой части рис. 34.1. Как показано в правой части данного рисунка, размеры окна можно увеличить.

Листинг 34.1. Простое Swing-приложение на Scala

```
import scala.swing._

object FirstSwingApp extends SimpleSwingApplication {
  def top = new MainFrame {
    title = "First Swing App"
    contents = new Button {
      text = "Click me"
    }
  }
}
```

Рис. 34.1. Простое Swing-приложение: исходный вид *(слева)* и увеличенный вид *(справа)*

Если построчно проанализировать код данного листинга, то можно заметить следующие элементы:

```
object FirstSwingApp extends SimpleSwingApplication {
```

Объект `FirstSwingApp` в первой строке после инструкции `import` — наследник класса `scala.swing.SimpleSwingApplication`. Это приложение отличается от традиционного приложения командной строки, которое может быть наследником класса `scala.App`. Класс `SimpleSwingApplication` уже определяет метод `main`, содержащий настроечный код для фреймворка Java Swing. Затем метод `main` продолжается вызовом предоставляемого вами метода `top`:

```
def top = new MainFrame {
```

В следующей строке реализуется метод `top`. Он содержит код, определяющий ваш GUI-компонент верхнего уровня. Обычно это некая разновидность фрейма `Frame`, то есть окна, которое может содержать произвольные данные. В показанном выше листинге 34.1 в качестве компонента верхнего уровня был выбран `MainFrame`, который похож на обычный Swing `Frame`, за исключением того, что с его закрытием закрывается и все GUI-приложение:

```
title = "First Swing App"
```

У `Frame`-объектов есть ряд атрибутов. Два наиболее важных из них — заголовок фрейма `title`, который будет записан в заголовке открывающегося окна, и его содержимое `content`, которое будет показано в самом окне. В Swing API Scala такие атрибуты смоделированы в виде свойств. Из материалов раздела 18.2 известно, что свойства кодируются в Scala в виде пар геттеров (`get`-методов) и сет-

теров (set-методов). Например, свойство title объекта типа Frame моделируется в виде геттера:

```
def title: String
```

и сеттера:

```
def title_=(s: String)
```

Это тот самый сеттер, который вызывается показанным ранее присваиванием значения переменной title. Эффект присваивания состоит в том, что выбранный заголовок отображается в заголовке окна. Если данный код убрать, то у окна будет пустой заголовок.

```
contents = new Button {
```

Корневой компонент Swing-приложения — фрейм top. Это контейнер (Container), следовательно, все остальные компоненты должны быть определены именно в нем. Каждый Swing-контейнер обладает свойством contents, позволяющим получать и устанавливать содержащиеся в нем компоненты. Метод получения значения свойства contents относится к типу Seq[Component], который показывает, что содержимым компонента в целом могут быть сразу несколько объектов. Но в качестве содержимого Frame-объектов всегда фигурирует только один компонент. Значение ему присваивается с помощью set-метода contents_=, и он же потенциально может изменять это значение. Например, в приведенном выше листинге 34.1 содержимое фрейма top составляет единственная кнопка Button:

```
text = "Click me"
```

У кнопки есть надпись, в данном случае это "Click me".

34.2. Панели и разметки

В качестве следующего шага добавим к фрейму top нашего приложения текст, который станет вторым элементом его содержимого. Внешний вид, который при этом приобретает приложение, показан в левой части рис. 34.2.

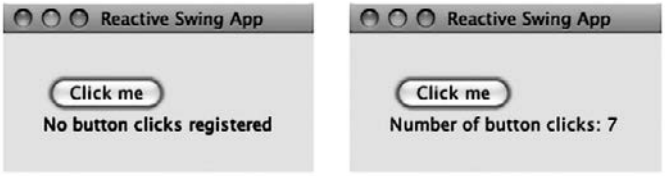


Рис. 34.2. Реактивное Swing-приложение в исходном состоянии *(слева)* и после щелчков *(справа)*

В предыдущем разделе было показано, что фрейм содержит всего один дочерний компонент. Следовательно, для создания фрейма не только с кнопкой,

но и с надписью нужно создать другой контейнерный компонент, содержащий и то и другое. Для этой цели используется панель. `Panel`-объект — контейнер, который показывает все содержащиеся в нем компоненты в соответствии с некими жестко заданными правилами разметки. Существует множество возможных разметок, реализуемых разными подклассами класса `Panel`, от самых простых до весьма замысловатых. По сути, одной из самых сложных частей комплексного GUI-приложения может стать получение правильной разметки: не так уж просто добиться, чтобы некий элемент довольно рационально отображался на всех разновидностях устройств и при всех размерах окон.

Полноценная реализация компоновки компонентов показана в листинге 34.2. Имеющиеся в этом классе два подкомпонента фрейма `top` называются `button` и `label`. Подкомпонент `button` определен так же, как в ранее показанном коде. Подкомпонент `label` показывает текстовое поле, не предназначенное для редактирования:

```
val label = new Label {
  text = "No button clicks registered"
}
```

Листинг 34.2. Компоновка компонентов на панели

```
import scala.swing._

object SecondSwingApp extends SimpleSwingApplication {
  def top = new MainFrame {
    title = "Second Swing App"
    val button = new Button {
      text = "Click me"
    }
    val label = new Label {
      text = "No button clicks registered"
    }
    contents = new BoxPanel(Orientation.Vertical) {
      contents += button
      contents += label
      border = Swing.EmptyBorder(30, 30, 10, 30)
    }
  }
}
```

Для кода в этом листинге выбрана простая вертикальная разметка, где компоненты устанавливаются друг на друга в `BoxPanel`:

```
contents = new BoxPanel(Orientation.Vertical) {
```

Свойство `contents`, принадлежащее `BoxPanel`, представляет собой изначально пустой буфер, в который с помощью оператора `+=` добавляются элементы `button` и `label`:

```
contents += button
contents += label
```

С помощью присваивания значения свойству панели `border` была добавлена рамка, которая окружает оба объекта:

```
border = Swing.EmptyBorder(30, 30, 10, 30)
```

Как и другие GUI-компоненты, рамки представлены в виде объектов. `EmptyBorder` является в `Swing`-объекте фабричным методом, получающим четыре параметра, которые задают ширину рамок рисуемых объектов сверху, справа снизу и слева.

Каким бы простым ни был этот пример, он уже продемонстрировал основной способ структурирования GUI-приложения. Он построен из компонентов, являющихся экземплярами таких классов `scala.swing`, как `Frame`, `Panel`, `Label` или `Button`. У компонентов есть свойства, которые могут настраиваться приложением. Компоненты объекта `Panel` могут содержать в своих свойствах `contents` несколько других компонентов. Таким образом, созданное GUI-приложение состоит из дерева компонентов.

34.3. Обработка событий

От созданного нами приложения, по сути, нет никакого проку. Если запустить на выполнение код, показанный выше в листинге 34.2, и щелкнуть на показанной в окне кнопке, ничего не произойдет. Фактически приложение имеет статический характер — оно никак не реагирует на события, инициируемые пользователем, за исключением щелчка на кнопке закрытия окна во фрейме `top`, который приводит к завершению работы приложения. Поэтому следующим шагом мы усовершенствуем приложение, чтобы видимая рядом с кнопкой надпись служила индикатором количества щелчков на кнопке. В правой части рис. 34.2 (см. выше) содержится копия экрана, показывающая, как должно выглядеть приложение после нескольких щелчков на кнопке.

Чтобы добиться от приложения соответствующего поведения, нужно подключить событие от пользователя (щелчок на кнопке) к действию (обновлению показываемой надписи). К обработке событий в Java и Scala используется одинаковый основной подход публикации-подписки: компоненты могут выступать в роли издателей и/или подписчиков. Издатель публикует события. Подписчик подписывается на публикацию, чтобы быть в курсе любых публикуемых событий. Издателей также называют источниками событий, а подписчиков — слушателями событий. Например, `Button`-объект является источником события, занимающимся публикацией события щелчка на кнопке — `ButtonClicked`.

В Scala подписка на источник события выполняется вызовом метода `listenTo(source)`. Кроме того, можно отписаться от источника события, вызвав метод `deafTo(source)`. В текущем примере приложения первым делом нужно заставить фрейм `top` отслеживать события вокруг его кнопки, чтобы получать уведомления о любых выдаваемых ею событиях. Для этого следует добавить к телу фрейма `top` следующий вызов:

```
listenTo(button)
```

Получение уведомления о событии — лишь первая половина дела, а вторая заключается в его обработке. Вот здесь фреймворк Scala Swing наиболее существенно отличается от Java Swing API и куда проще справляется с задачей. В Java появление сигнала о событии означает вызов уведомляющего метода в отношении объекта, который должен реализовать какие-либо `Listener`-интерфейсы. Обычно все это предполагает большой объем косвенного и шаблонного кода, который затрудняет написание и чтение приложений, обрабатывающих события. В отличие от этого в Scala событие является настоящим объектом, отправляемым в адрес подписавшихся компонентов, что во многом похоже на сообщения, отправляемые акторам. Например, щелчок на кнопке приведет к созданию события, являющегося экземпляром следующего `case`-класса:

```
case class ButtonClicked(source: Button)
```

Параметр `case`-класса ссылается на кнопку, на которой был сделан щелчок. Как и в случае со всеми другими событиями Scala Swing, этот класс события содержится в пакете `scala.swing.event`.

Чтобы заставить компоненты реагировать на поступающие события, нужно к свойству по имени `reactions` добавить обработчик. Код, выполняющий эту задачу, выглядит так:

```
var nClicks = 0
reactions += {
  case ButtonClicked(b) =>
    nClicks += 1
    label.text = "Number of button clicks: " + nClicks
}
```

В первой строке кода определяется переменная `nClicks`, в которой хранятся данные о количестве щелчков на кнопке. В остальных строках между фигурными скобками добавляется код, выступающий в качестве *обработчика* свойства `reactions` фрейма `top`. Обработчики — это функции, определяемые сопоставлением с образцами событий, что во многом похоже на то, как актор Akka получает методы, определенные сопоставлением с образцами сообщений. Показанный ранее обработчик соответствует событиям, которые имеют форму `ButtonClicked(b)`, то есть любому событию, являющемуся экземпляром класса `ButtonClicked`. Переменная паттерна `b` ссылается на кнопку, на которой щелкнули. Действие, соответствующее этому событию в показанном коде, увеличивает значение `nClicks` на единицу и обновляет текст надписи.

В общем, обработчик относится к функции `PartialFunction`, которая соответствует событиям и выполняет некоторые действия. Вдобавок в одном обработчике можно определить соответствия нескольким разновидностям событий, для чего используется несколько `case`-вариантов.

Свойство `reactions` реализует коллекцию, точно так же, как и свойство `contents`. Некоторые компоненты берутся из предопределенных реакций. Например, в классе `Frame` есть предопределенная реакция, закрывающая окно при щелчке пользовате-

ля на кнопке закрытия в верхнем правом углу. Если установить собственные реакции, добавляя их к свойству `reactions` с помощью оператора `+=`, то определяемые вами реакции будут рассматриваться наряду со стандартными. Концептуально обработчики, установленные в `reactions`, формируют стек. В текущем примере при получении события фреймом `top` первым будет испробован обработчик, соответствующий `ButtonClicked`, поскольку он был последним обработчиком, установленным для фрейма. Если полученное событие относится к типу `ButtonClicked`, то вызывается код, ассоциированный с паттерном. После выполнения кода система станет искать в стеке обработчиков следующие обработчики, которые также можно применить к событию. Если полученное событие не относится к типу `ButtonClicked`, то тут же распространяется на все остальное, что установлено в стеке обработчиков. Кроме того, обработчики из свойства `reactions` можно удалять, воспользовавшись для этого оператором `-=`.

В листинге 34.3 показано полноценное приложение, включающее реакции. В коде проиллюстрированы основные элементы GUI-приложения фреймворка Scala Swing: приложение состоит из трех компонентов, начинающихся с фрейма `top`. В коде показаны компоненты `Frame`, `BoxPanel`, `Button` и `Label`, но существует и множество других разновидностей компонентов, определенных в библиотеках Swing.

Листинг 34.3. Реализация Swing-приложения, реагирующего на события

```
import scala.swing._
import scala.swing.event._

object ReactiveSwingApp extends SimpleSwingApplication {
  def top = new MainFrame {
    title = "Reactive Swing App"
    val button = new Button {
      text = "Click me"
    }
    val label = new Label {
      text = "No button clicks registered"
    }
    contents = new BoxPanel(Orientation.Vertical) {
      contents += button
      contents += label
      border = Swing.EmptyBorder(30, 30, 10, 30)
    }
    listenTo(button)
    var nClicks = 0
    reactions += {
      case ButtonClicked(b) =>
        nClicks += 1
        label.text = "Number of button clicks: " + nClicks
    }
  }
}
```

Каждый компонент настраивается установкой атрибутов. Два важных атрибута — это `contents`, в котором дочерние элементы компонента фиксируются в дереве, и `reactions`, в котором определяется, как компонент реагирует на события.

34.4. Пример: программа перевода градусов между шкалами Цельсия и Фаренгейта

В качестве примера создадим GUI-программу перевода градусов между шкалами Цельсия и Фаренгейта. Пользовательский интерфейс программы показан на рис. 34.3. Он состоит из двух текстовых полей, показанных на белом фоне, за каждым из которых следует надпись. В одном текстовом поле показана температура в градусах Цельсия, а в другом — в градусах Фаренгейта. Пользователь приложения может редактировать каждое из полей. Как только он меняет температуру в любом из полей, автоматически обновляется температура в другом поле.

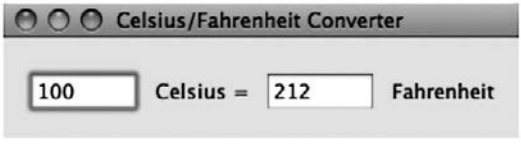


Рис. 34.3. Программа перевода градусов между шкалами Цельсия и Фаренгейта

Полный код, реализующий это приложение, показан в листинге 34.4. Импортирование в верхней части кода использует короткую форму записи:

```
import swing._
import event._
```

Фактически это эквивалент импортирования, которое использовалось ранее:

```
import scala.swing._
import scala.swing.event._
```

Листинг 34.4. Реализация конвертера температур

```
import swing._
import event._

object TempConverter extends SimpleSwingApplication {
  def top = new MainFrame {
    title = "Celsius/Fahrenheit Converter"
    object celsius extends TextField { columns = 5 }
    object fahrenheit extends TextField { columns = 5 }
    contents = new FlowPanel {
      contents += celsius
      contents += new Label(" Celsius = ")
      contents += fahrenheit
```

```
      contents += new Label(" Fahrenheit")
      border = Swing.EmptyBorder(15, 10, 10, 10)
    }
    listenTo(celsius, fahrenheit)
    reactions += {
      case EditDone(`fahrenheit`) =>
        val f = fahrenheit.text.toInt
        val c = (f - 32) * 5 / 9
        celsius.text = c.toString
      case EditDone(`celsius`) =>
        val c = celsius.text.toInt
        val f = c * 9 / 5 + 32
        fahrenheit.text = f.toString
    }
  }
}
```

Возможность использования короткой записи обусловливается тем, что эти пакеты вложены в Scala. Пакет `scala.swing` содержится в пакете `scala` и все содержимое пакета импортируется автоматически, поэтому для ссылки на пакет можно просто записать `swing`. Аналогично этому пакет `scala.swing.event` содержится в качестве подпакета `event` в пакете `scala.swing`. Поскольку благодаря первой инструкции `import` вы уже импортировали все, что имеется в `scala.swing`, то после этого на пакет `event` можно просто сослаться как на `event`.

Два компонента, `celsius` и `fahrenheit`, в `TempConverter` являются объектами класса `TextField`. Класс `TextField` в Swing выступает компонентом, позволяющим редактировать одну строку текста. Его исходная ширина задается свойством `columns`, значение которого измеряется в символах (для обоих полей установлено значение `5`).

Содержимое `TempConverter` собрано на панели, куда включены два текстовых поля и две надписи, поясняющие, что это за поля. Панель относится к классу `FlowPanel`, а это значит, она отображает все свои элементы один за другим в одной или несколько строках в зависимости от ширины фрейма.

Свойство `reactions`, принадлежащее `TempConverter`, определено обработчиком, содержащим два `case`-варианта. Каждый `case`-вариант соответствует событию `EditDone` для одного из двух текстовых полей. Это событие выдается после редактирования текстового поля пользователем. Обратите внимание на форму паттернов, включающую обратные кавычки вокруг имен элементов:

```
case EditDone(`celsius`)
```

В соответствии с разъяснениями, которые были даны в разделе 15.2, обратные кавычки вокруг `celsius` гарантируют, что паттерн будет соответствовать только в том случае, если источником события будет объект `celsius`. Если обратные кавычки не поставить, а просто написать `case EditDone(celsius)`, то паттерн будет соответствовать каждому событию класса `EditDone`. Затем измененное поле будет сохранено в переменной паттерна `celsius`. Очевидно, это совсем не то, что вам было нужно. В качестве альтернативного варианта можно определить два объекта

`TextField` с именами, начинающимися с символов в верхнем регистре, то есть `Celsius` и `Fahrenheit`. В таком случае соответствие им можно определять напрямую, без обратных кавычек, как в коде `case EditDone(Celsius)`.

Два действия событий `EditDone` преобразуют одну величину в другую. Каждое из них начинается со считывания содержимого измененного поля и его преобразования в `Int`-значение. Затем один температурный показатель в градусах преобразуется в другой с помощью формулы, а результат сохраняется как строка в другом текстовом поле.

Резюме

В этой главе мы предоставили вам возможность получить первые впечатления о программировании GUI с помощью имеющейся в Scala оболочки для фреймворка Swing. В ней мы показали, как собираются GUI-компоненты, как можно настроить их свойства и как обрабатываются события. В целях экономии места в книге мы смогли рассмотреть лишь небольшое количество простых компонентов. Но существует множество компонентов других разновидностей. Узнать о них можно из документации по языку Scala, касающейся пакета `scala.swing`. В следующей главе мы представим пример разработки более сложного Swing-приложения.

Кроме того, есть множество руководств по исходному фреймворку Java Swing, на котором основана имеющаяся в Scala оболочка[1]. Оболочки Scala похожи на исходные Swing-классы, но в них предпринята попытка упростить концепции и придать им более универсальный характер. В упрощениях широко используются свойства языка Scala. Например, имеющаяся в Scala эмуляция свойств и перегрузка операторов позволяют составлять удобные определения свойств, применяя присваивания и операции `+=`. Используемая в языке философия «абсолютно все является объектом» позволяет наследовать метод `main` GUI-приложения. Таким образом, этот метод может быть скрыт от пользовательских приложений, включая весь попутно следующий рутинный настроечный код. И наконец, самое главное: имеющиеся в Scala функции первого класса и технология сопоставления с образцом позволяют формулировать обработку событий в виде свойства компонента `reactions`, тем самым значительно упрощая жизнь разработчику приложений.

[1] Обратите, к примеру, внимание на издание The Java Tutorials: Creating a GUI with JFC/Swing. Доступно на сайте http://java.sun.com/docs/books/tutorial/uiswing.

35 Электронная таблица SCells

В предыдущих главах мы рассмотрели множество различных конструкций языка программирования Scala. Здесь же покажем совместную работу этих конструкций при реализации серьезного приложения. Задача заключается в создании приложения в виде электронной таблицы, для которого выбрано имя `SCells`.

Данное приложение представляет интерес по нескольким причинам. Во-первых, все знают, что такое электронные таблицы, следовательно, будет нетрудно разобраться в том, чем именно должно заниматься приложение. Во-вторых, электронные таблицы — это программы, в которых выполняется широкий диапазон различных вычислительных задач. Присутствует визуальный аспект: электронная таблица представляется как весьма развитое GUI-приложение. Есть символьный аспект, касающийся формул, их синтаксического разбора и интерпретации. Имеется также вычислительный аспект, предусматривающий наращиваемое обновление, возможно, весьма крупных таблиц. Не обходится и без реактивного аспекта, выражающегося в подходе к электронным таблицам как к программам, реагирующим на события довольно сложным образом. И наконец, существует аспект компонентов, при котором приложение конструируется в виде набора компонентов, допускающих повторное использование. И все эти аспекты мы довольно пристально рассмотрим в текущей главе.

35.1. Визуальная среда

Начнем с создания основной визуальной среды приложения. В первом приближении внешний вид пользовательского интерфейса показан на рис. 35.1. Как видите, электронная таблица может прокручиваться. В ней имеются строки от 0 до 99, и столбцы от A до Z. В Swing это выражается путем определения электронной таблицы в виде объекта `ScrollPane`, содержащего объект `Table`. Код показан в листинге 35.1.

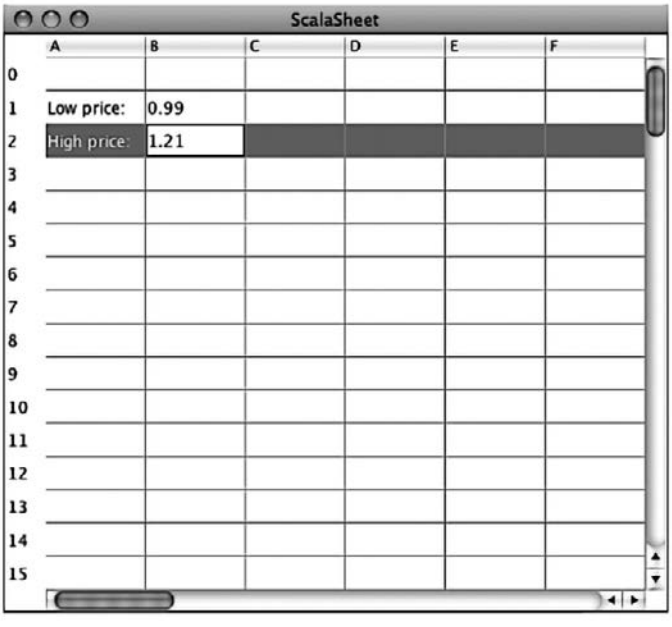


Рис. 35.1. Простая электронная таблица

Листинг 35.1. Код для электронной таблицы, показанной на рис. 35.1

```
package org.stairwaybook.scells
import swing._

class Spreadsheet(val height: Int, val width: Int)
    extends ScrollPane {

  val table = new Table(height, width) {
    rowHeight = 25
    autoResizeMode = Table.AutoResizeMode.Off
    showGrid = true
    gridColor = new java.awt.Color(150, 150, 150)
  }

  val rowHeader =
    new ListView((0 until height) map (_.toString)) {
      fixedCellWidth = 30
      fixedCellHeight = table.rowHeight
    }

  viewportView = table
  rowHeaderView = rowHeader
}
```

Компонент электронной таблицы, показанный в данном листинге, определен в пакете `org.stairwaybook.scells`, который будет содержать все необходимые приложению классы, трейты и объекты. Основные элементы оболочки Scala Swing импортируются в него из пакета `scala.swing`. Сама электронная таблица — класс, получающий в качестве параметров высоту и ширину (`height` и `width`), которые выражаются в количестве ячеек. Он расширяет класс `ScrollPane`, который дает ему полосы прокрутки, расположенные с правой стороны и внизу таблицы, показанной на рис. 35.1. Он содержит два компонента с именами `table` и `rowHeader`.

Компонент `table` — экземпляр анонимного подкласса, суперклассом которого является `scala.swing.Table`. В четырех строках его тела устанавливаются значения его атрибутов: `rowHeight` для высоты строки таблицы в точках, `autoResizeMode` для отключения автоматической установки размеров таблицы, `showGrid` для отображения сетки между ячейками и `gridColor` для задания сетке темно-серого цвета.

Компонент `rowHeader`, содержащий заголовки с номерами строк в левой части электронной таблицы на рис. 35.1, относится к типу `ListView`, отображающему в своих элементах строковые значения от 0 до 99. Две строки в его теле фиксируют ширину ячейки, задавая ее значение 30 точек, и высоту, определяя ее равной значению атрибута `rowHeight`, принадлежащего компоненту `table`.

Вся электронная таблица собирается установкой значений для двух полей в `ScrollPane`. Для поля `viewportView` устанавливается значение `table`, а для поля `rowHeaderView` — значение списка `rowHeader`. Разница между этими двумя представлениями состоит в том, что окно просмотра области прокрутки является областью, прокручиваемой с помощью двух полос, а заголовки строк в левой части при перемещении полосы горизонтальной прокрутки остаются зафиксированными. В соответствии с особенностями Swing заголовки столбцов задаются в исходном виде в верхней части таблицы, что избавляет от необходимости определять их явно.

Чтобы получить простую электронную таблицу, показанную выше, на рис. 35.1, нужно всего лишь определить основную программу, создающую компонент `Spreadsheet`. Такая программа показана в листинге 35.2.

Листинг 35.2. Основная программа для приложения электронной таблицы

```
package org.stairwaybook.scells
import swing._

object Main extends SimpleSwingApplication {
  def top = new MainFrame {
    title = "ScalaSheet"
    contents = new Spreadsheet(100, 26)
  }
}
```

Программа `Main` — наследник класса `SimpleSwingApplication`, который берет на себя решение всех низкоуровневых настроек, предваряющих запуск Swing-приложения. Остается только определить высокоуровневое окно приложения

в методе top. В нашем примере top относится к объекту класса MainFrame, у которого определяются два элемента: заголовок title, для которого установлено значение ScalaSheet, и содержимое contents, для которого установлено значение в виде экземпляра класса Spreadsheet, имеющего 100 строк и 26 столбцов. Вот и все. Если запустить это приложение командой scala org.stairwaybook.scells.Main, то появится такая же электронная таблица, что и на рис. 35.1 (см. выше).

35.2. Разделение введенных данных и отображения

Если поработать с созданной на данный момент электронной таблицей, то сразу станет заметно: вывод, отображаемый в ячейке, всегда в точности соответствует тому, что введено в эту ячейку. Настоящая электронная таблица так себя не ведет. В ячейку вводится формула, а отображается вычисленное с ее помощью значение. То есть то, что введено в ячейку, отличается от отображаемого в ней.

В качестве первого шага к созданию настоящего приложения электронной таблицы нужно сконцентрироваться на разделении введенных данных и отображения. Основной механизм отображения содержится в методе rendererComponent класса Table. Изначально rendererComponent отображает то, что введено. Если требуется внести изменения в этот порядок, то нужно переопределить rendererComponent делать что-нибудь другое. Новая версия Spreadsheet с методом rendererComponent показана в листинге 35.3.

Листинг 35.3. Электронная таблица с методом rendererComponent

```
package org.stairwaybook.scells
import swing._

class Spreadsheet(val height: Int, val width: Int)
    extends ScrollPane {

  val cellModel = new Model(height, width)
  import cellModel._

  val table = new Table(height, width) {

    // Те же настройки, что и прежде...

    override def rendererComponent(isSelected: Boolean,
        hasFocus: Boolean, row: Int, column: Int): Component =

      if (hasFocus) new TextField(userData(row, column))
      else
         new Label(cells(row)(column).toString) {
           xAlignment = Alignment.Right
         }
```

```
    def userData(row: Int, column: Int): String = {
      val v = this(row, column)
      if (v == null) "" else v.toString
    }
  }
  // Все остальное, как и прежде...
}
```

Метод `rendererComponent` переопределяет исходный метод в классе `Table`. Он получает четыре параметра. Параметры `isSelected` и `hasFocus` типа `Boolean` показывают, была ли выбрана ячейка и имеет ли она фокус, означающий, что события клавиатуры будут отражаться на содержимом ячейки. Остальные два параметра, `row` и `column`, задают координаты ячейки.

В новом методе `rendererComponent` проверяется наличие у ячейки фокуса ввода. Если значение `hasFocus` равно `true`, то это говорит об использовании ячейки для редактирования. В таком случае нужно, чтобы отображалось редактируемое поле `TextField`, содержащее данные, введенные пользователем на текущий момент. Эти данные возвращаются благодаря применению вспомогательного метода `userData`, который отображает содержимое данной строки и данного столбца таблицы. Содержимое извлекается путем вызова `this(row, column)`[1]. Метод `userData` также берет на себя вывод вместо `null`-элемента не `null`, а пустой строки.

Пока все идет хорошо. Но что будет отображено, если у ячейки нет фокуса? В настоящей электронной таблице станет отображаться значение ячейки. Таким образом, в действительности в работе находятся сразу две таблицы. Первая из них, по имени `table`, содержит то, что ввел пользователь. Вторая, перекрытая, таблица содержит внутреннее представление ячеек и что должно быть отображено. В примере электронной таблицы эта таблица представлена двумерным массивом `cells`. Если ячейка на пересечении данных строки и столбца не имеет фокуса редактирования, то метод `rendererComponent` отобразит элемент `cells(row)(column)`. Он не может быть редактируемым, следовательно, должен быть отображен в виде значения типа `Label`, а не в виде редактируемого значения типа `TextField`.

Осталось определить внутренний массив ячеек. Сделать это можно непосредственно в классе `Spreadsheet`, но обычно более предпочтительно отделить представление GUI-компонента от его внутренней модели. Поэтому в показанном ранее примере массив `cells` определен в отдельном классе по имени `Model`. Модель интегрирована в `Spreadsheet` путем определения значения `cellModel` типа `Model`. Инструкция `import`, которая следует за этим определением `val`-переменной, делает элементы `cellModel` доступными внутри `Spreadsheet`, не указывая для них какой-либо префикс. Первая, упрощенная версия класса `Model` показана в листинге 35.4. В классе определяются внутренний класс по имени `Cell` и двумерный массив по имени `cells`, содержащий элементы типа `Cell`. Каждый элемент инициализирован как совершенно новый `Cell`-объект.

---

[1] Хотя код this(row, column) может быть похож на вызов конструктора, в данном случае он является вызовом метода apply текущего экземпляра класса Table.

Листинг 35.4. Первая версия класса Model

```
package org.stairwaybook.scells

class Model(val height: Int, val width: Int) {

  case class Cell(row: Int, column: Int)

  val cells = Array.ofDim[Cell](height, width)

  for (i <- 0 until height; j <- 0 until width)
    cells(i)(j) = new Cell(i, j)
}
```

Пока все правильно. Если скомпилировать измененный класс `Spreadsheet` с классом `Model` и запустить приложение `Main`, то появится такое же окно, как на рис. 35.2.

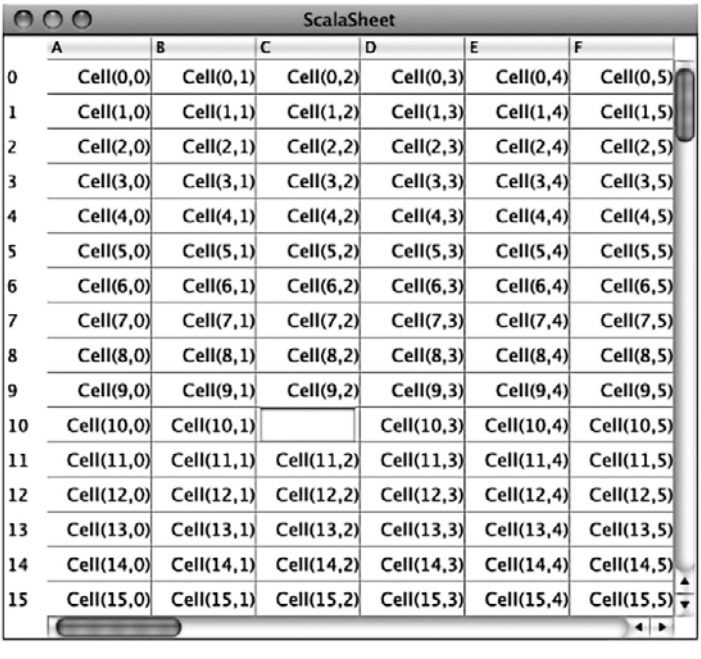


Рис. 35.2. Ячейки, показывающие собственные координаты

Целью данного раздела было создать конструкцию, в которой отображаемое в ячейке значение отличается от введенной в нее строки. Вполне очевидно, что цель достигнута, хотя и весьма примитивным способом. В новой электронной таблице можно вводить в ячейку все что угодно, но при потере фокуса в ней всегда будут отображаться только ее координаты. Понятно, что работа еще не завершена.

35.3. Формулы

В действительности в ячейке электронной таблицы хранятся две вещи: текущее значение и формула его вычисления. В электронной таблице есть три типа формул.

1. Числовые значения, такие как `1.22`, `-3` или `0`.
2. Текстовые надписи, такие как `Annual sales`, `Deprecation` или `total`.
3. Формулы, вычисляющие новое значение из содержимого ячеек, например `=add(A1,B2)` или `=sum(mul(2, A2), C1:D16)`.

Формула, вычисляющая значение, всегда начинается со знака равенства и продолжается арифметическим выражением. Электронная таблица `SCells` придерживается в отношении арифметических выражений весьма простого и постоянного соглашения: каждое выражение есть применение какой-либо функции к списку аргументов. Именем функции выступает идентификатор `add` — для бинарного сложения или `sum` — для суммирования произвольного количества операндов. Аргументом функции может быть число, ссылка на ячейку, ссылка на диапазон ячеек, такая как `C1:D16`, или еще одно применение функции. Чуть позже вы увидите, что у `SCells` открытая архитектура, которая облегчает установку собственных функций с помощью составления композиции примесей.

Первым шагом в решении задачи обработки формул станет запись представляющих их типов. В соответствии с возможными ожиданиями различные виды формул представлены `case`-классами. Содержимое файла `Formulas.scala`, в котором определены эти `case`-классы, показано в листинге 35.5.

Листинг 35.5. Классы, представляющие формулы

```
package org.stairwaybook.scells

trait Formula

case class Coord(row: Int, column: Int) extends Formula {
  override def toString = ('A' + column).toChar.toString + row
}
case class Range(c1: Coord, c2: Coord) extends Formula {
  override def toString = c1.toString + ":" + c2.toString
}
case class Number(value: Double) extends Formula {
  override def toString = value.toString
}
case class Textual(value: String) extends Formula {
  override def toString = value
}
case class Application(function: String,
    arguments: List[Formula]) extends Formula {

  override def toString =
    function + arguments.mkString("(", ",", ")")
}
object Empty extends Textual("")
```

Трейт `Formula`, показанный в данном листинге, содержит в качестве дочерних пять `case`-классов:

- `Coord` — для координат ячейки, например `A3`;
- `Range` — для диапазона ячеек, например `A3:B17`;
- `Number` — для чисел с плавающей точкой, например `3.1415`;
- `Textual` — для текстовых надписей, например `Deprecation`;
- `Application` — для применения функций, например `sum(A1,A2)`.

В каждом `case`-классе метод `toString` переопределяется с тем, чтобы с его помощью показанным ранее стандартным способом отобразить вид его формул. Для удобства есть также объект `Empty`, представляющий содержимое пустой ячейки. Объект `Empty` — экземпляр класса `Textual` с пустым строковым аргументом.

35.4. Синтаксический разбор формул

В предыдущем разделе мы показали различные виды формул и то, как они отображаются в строковом виде. В этом поговорим о том, как запустить данный процесс в обратную сторону: преобразовать пользовательский строковый ввод в дерево типа `Formula`. Далее в разделе поочередно рассмотрим различные элементы класса `FormulaParsers`, содержащие парсеры, выполняющие преобразования. Класс построен на базе фреймворка комбинаторов, рассмотренного в главе 33. Выражаясь конкретнее, парсеры формул — экземпляры класса `RegexParsers`, рассмотренного в данной главе:

```
package org.stairwaybook.scells
import scala.util.parsing.combinator._

object FormulaParsers extends RegexParsers {
```

Первые два элемента объекта `FormulaParsers` — дополнительные парсеры для идентификаторов и десятичных чисел:

```
def ident: Parser[String] = """[a-zA-Z_]\w * """.r
def decimal: Parser[String] = """-?\d+(\.\d * )?""".r
```

Если исходить из первого показанного ранее регулярного выражения, то идентификатор начинается с буквы или знака подчеркивания. Затем следует произвольное количество символов «слова», представленных кодом регулярного выражения `\w`, который распознает буквы, цифры или знаки подчеркивания. Второе регулярное выражение дает описание десятичных чисел, в которых могут быть необязательный знак минус, одна или несколько цифр, представленных кодом регулярного выражения `\d`, и необязательная дробная часть, состоящая из точки, за которой стоят от нуля до нескольких цифр.

Следующий элемент объекта `FormulaParsers` — парсер ячейки, который распознает координаты ячейки, такие как `C11` или `B2`. Сначала он вызывает парсер на основе регулярного выражения, определяющий форму координат: одна буква, за которой следует одна или несколько цифр. Затем строка, возвращенная этим парсером, преобразуется в координаты ячейки отделением буквы от числовой части и преобразованием двух частей в индексы для столбца и строки ячейки:

```
def cell: Parser[Coord] =
  """[A-Za-z]\d+""".r ^^ { s =>
    val column = s.charAt(0).toUpper - 'A'
    val row = s.substring(1).toInt
    Coord(row, column)
}
```

Обратите внимание на некоторую ограниченность парсера ячейки, позволяющего использовать только те координаты столбцов, которые состоят из одной буквы. Таким образом, количество столбцов электронной таблицы фактически ограничивается числом не более 26, поскольку последующие столбцы не могут пройти синтаксический разбор. Было бы неплохо расширить возможности парсера, чтобы он воспринимал ячейки с несколькими начальными буквами. Пусть это станет упражнением для самостоятельного выполнения.

Диапазон, распознаваемый парсером `range`, — диапазон ячеек. Такой диапазон состоит из координат двух ячеек с двоеточием между ними:

```
def range: Parser[Range] =
  cell~":"~cell ^^ {
    case c1~":"~c2 => Range(c1, c2)
  }
```

Парсер числа `number` распознает десятичное число, которое конвертируется в значение типа `Double` и заключается в экземпляр класса `Number`:

```
def number: Parser[Number] =
  decimal ^^ (d => Number(d.toDouble))
```

Парсер применения `application` распознает применение функции. Оно состоит из идентификатора, за которым стоит список аргументов, заключенный в круглые скобки:

```
def application: Parser[Application] =
  ident~"("~repsep(expr, ",")~")" ^^ {
    case f~"("~ps~")" => Application(f, ps)
  }
```

Парсер выражения `expr` распознает выражение формулы: либо формулы высшего уровня, следующей за знаком `=`, либо аргумента для функции. Такое выражение формулы определяется в виде ячейки, диапазона ячеек, числа или применения:

```
def expr: Parser[Formula] =
  range | cell | number | application
```

Это определение парсера `expr` представлено в несколько упрощенном виде, поскольку диапазоны ячеек должны появляться только в аргументах функций — они недопустимы в формулах верхнего уровня. Грамматику формулы можно изменить, чтобы разделить два случая использования выражений, а диапазон синтаксически исключить из формул верхнего уровня. В представленной здесь электронной таблице подобная ошибка обнаруживается при вычислении выражения.

Парсер текста `textual` распознает произвольную строку ввода, если только она не начинается со знака равенства (вспомним, что строки, начинающиеся с `=`, считаются формулами):

```
def textual: Parser[Textual] =
  """[^=]. * """.r ^^ Textual
```

Парсер формул `formula` распознает все виды допустимого для ячейки ввода. Формулой является либо число, либо текстовый ввод, либо формула, начинающаяся со знака равенства:

```
def formula: Parser[Formula] =
  number | textual | "=" ~ >expr
```

На этом грамматика для ячеек электронных таблиц завершается. Заключительный метод `parse` использует данную грамматику в методе, преобразующем строку ввода в дерево типа `Formula`:

```
  def parse(input: String): Formula =
    parseAll(formula, input) match {
      case Success(e, _) => e
      case f: NoSuccess => Textual("[" + f.msg + "]")
    }
} // Окончание FormulaParsers
```

Метод `parse` выполняет синтаксический разбор всего ввода с помощью парсера `formula`. В случае успеха возвращается получившаяся формула. В случае сбоя вместо этого возвращается объект типа `Textual` с сообщением об ошибке.

Вот, собственно, и все, что касается разбора формул. Осталось только встроить парсеры в электронную таблицу. Для этого можно обогатить класс `Cell` в классе `Model` полем `formula`:

```
case class Cell(row: Int, column: Int) {
  var formula: Formula = Empty
  override def toString = formula.toString
}
```

В новой версии класса `Cell` метод `toString` определен для отображения формулы ячейки. Таким образом, можно будет проконтролировать, был ли синтаксический разбор формулы выполнен правильно.

Последним шагом в этом разделе станет встраивание парсера в электронную таблицу. Синтаксический разбор формулы — ответная реакция на пользовательский ввод в ячейку. Полноценный ввод в ячейку смоделирован в Swing событием

`TableUpdated`. Класс `TableUpdated` содержится в пакете `scala.swing.event`. Событие имеет следующую форму:

```
TableUpdated(table, rows, column)
```

Оно включает измененную таблицу, а также набор координат задействованных ячеек, заданный `rows` и `column`. Параметр `rows` — значение диапазона типа `Range[Int]`[1]. Параметр `column` — целое число. Таким образом, в общем событие `TableUpdated` может ссылаться на несколько задействованных ячеек, но они должны содержаться в последовательном диапазоне строк и относиться к одному и тому же столбцу.

Как только в таблицу внесены изменения, задействованные ячейки должны быть подвергнуты синтаксическому разбору заново. Чтобы реагировать на событие `TableUpdated`, нужно, как показано в листинге 35.6, добавить вариант к значению `reactions` компонента `table`.

Листинг 35.6. Электронная таблица, выполняющая синтаксический разбор формул

```
package org.stairwaybook.scells
import swing._
import event._

class Spreadsheet(val height: Int, val width: Int) ... {
  val table = new Table(height, width) {
    ...
    reactions += {
      case TableUpdated(table, rows, column) =>
        for (row <- rows)
          cells(row)(column).formula =
            FormulaParsers.parse(userData(row, column))
    }
  }
}
```

Теперь при редактировании таблицы формулы всех задействованных ячеек будут обновляться за счет разбора соответствующих пользовательских данных. После компиляции рассмотренных на данный момент классов и запуска приложения `scells.Main` вы увидите такую же электронную таблицу, как та, что показана на рис. 35.3. Редактировать ячейки можно, набирая в них текст. После завершения редактирования в ячейке будет показана содержащаяся в ней формула. Можно также попробовать ввести в поле, имеющее фокус редактирования (см. рис. 35.3), какой-нибудь недопустимый текст, например `=add(1, X)`.

О недопустимом вводе будет свидетельствовать сообщение об ошибке. К примеру, как только вы покинете редактируемое поле, показанное на рис. 35.3, в ячейке появится сообщение об ошибке `['(' expected]` (чтобы просмотреть сообщение об ошибке целиком, может понадобиться расширить столбец, перетаскивая вправо разделитель столбцов в заголовке).

---

[1] Range[Int] также является типом такого выражения Scala, как 1 to N.

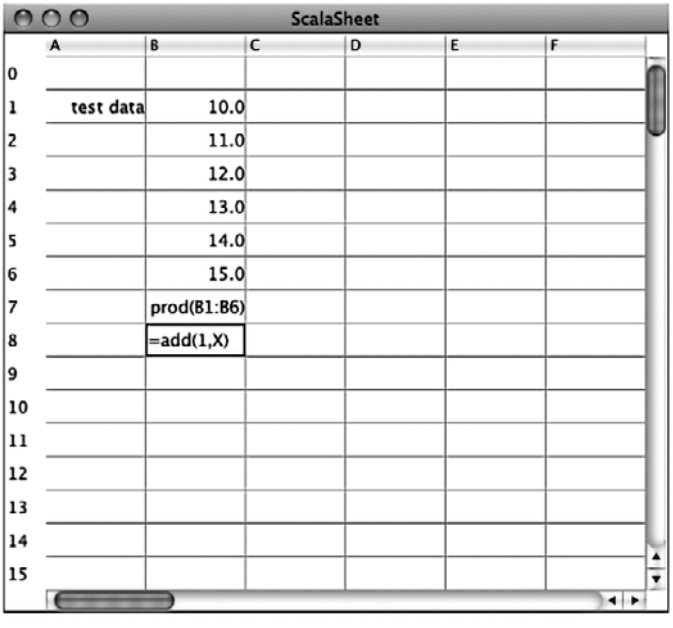


Рис. 35.3. Ячейки, показывающие свои формулы

35.5. Вычисление

Разумеется, в итоге электронная таблица должна не только показывать формулы, но и выполнять по ним вычисления. В этом разделе мы добавим все необходимые для этого компоненты.

Нам необходим метод `evaluate`, получающий формулу и возвращающий значение, представленное типом `Double`, которое вычисляется по этой формуле в текущей электронной таблице. Данный метод будет помещен в новый трейт по имени `Evaluator`. Методу требуется доступ к полям ячеек в классе `Model`, чтобы определить текущие значения ячеек, на которые имеются ссылки в формуле. В то же время классу `Model` нужно вызывать метод `evaluate`. Следовательно, между `Model` и `Evaluator` возникает взаимозависимость. Вполне подходящий способ выражения подобной взаимозависимости между классами был показан в главе 29: в одном направлении следует использовать наследование, а в другом — собственные типы.

В примере электронной таблицы класс `Model` является наследником класса `Evaluator`, получая таким образом доступ к его методу `evaluate`. В обратном направлении в классе `Evaluator` в качестве его собственного типа определяется класс `Model`:

```
package org.stairwaybook.scells
trait Evaluator { this: Model => ...
```

Таким образом, предполагается, что значением `this` внутри класса `Evaluator` будет `Model` и массив ячеек будет доступен при написании кода в виде либо `cells`, либо `this.cells`.

После создания подобной связи сконцентрируемся на определении содержимого класса `Evaluator`. В листинге 35.7 показана реализация метода `evaluate`. Как вы и ожидали, в методе содержится сопоставление с образцом в отношении различных типов формул. Для координат `Coord(row, column)` оно возвращает значение массива ячеек по заданным координатам. Для числа `Number(v)` — значение `v`. Для текстовой надписи `Textual(s)` возвращает нуль. И наконец, для применения функции `Application(function, arguments)` оно вычисляет значения всех аргументов, извлекает объект функции, соответствующий ее имени, из таблицы `operations` и применяет эту функцию к значениям всех аргументов.

Листинг 35.7. Метод evaluate трейта Evaluator

```
def evaluate(e: Formula): Double = try {
  e match {
    case Coord(row, column) =>
      cells(row)(column).value
    case Number(v) =>
      v
    case Textual(_) =>
      0
    case Application(function, arguments) =>
      val argvals = arguments flatMap evalList
      operations(function)(argvals)
  }
} catch {
  case ex: Exception => Double.NaN
}
```

В таблице `operations` имена функций отображаются на объекты функций. Определение выглядит следующим образом:

```
type Op = List[Double] => Double
val operations = new collection.mutable.HashMap[String, Op]
```

Как видно из данного определения, операции моделируются как функции от списков значений к значениям. Тип `Op` вводит весьма удобный псевдоним для типа операции.

Для защиты от ошибок ввода вычисление в `evaluate` заключено в блок `try-catch`. При вычислении формулы ячейки может сложиться множество неблагоприятных обстоятельств: координаты могут выйти за пределы диапазона, имена функций могут быть не определены, у функций может оказаться неверное количество аргументов, арифметические операции могут быть неприемлемыми или вызывающими переполнение буфера. Реакция на любую из этих ошибок одна и та же — возвращение значения «не число». Возвращаемое значение, `Double.NaN`, в соответствии со спецификацией IEEE — представление для вычисления, которое не имеет представляемого значения в виде числа с плавающей точкой. Так может

случиться, к примеру, по причине переполнения буфера или деления на нуль. Метод `evaluate`, показанный в листинге 35.7 (см. выше), выбирает возвращение такого же значения и для всех прочих видов ошибок. Преимущество этой схемы заключается в том, что в ней проще разобраться и для ее реализации не требуется слишком большой объем кода. Недостатком является то, что все виды ошибок оказываются объединены, что не позволяет пользователю электронной таблицы получить подробный ответ на вопрос о причине сбоя. При желании вы можете поэкспериментировать с созданием более конкретных способов представления ошибок в приложении `SCells`.

Вычисление аргументов отличается от вычисления формул верхнего уровня. Аргументы могут быть списком, а функции верхнего уровня — не могут. Например, выражение аргумента `A1:A3` в `sum(A1:A3)` возвращает значения ячеек `A1`, `A2`, `A3` в списке. Этот список затем передается операции `sum`. В выражениях аргументов можно также смешивать списки и отдельные значения, как, например, в операции `sum(A1:A3, 1.0, C7)`, которой будет возвращена сумма пяти значений. Обрабатывать аргументы, которые могут вычисляться в списки, можно с помощью еще одной функции, `evalList`, получающей формулу и возвращающей список значений:

```
private def evalList(e: Formula): List[Double] = e match {
  case Range(_, _) => references(e) map (_.value)
  case _ => List(evaluate(e))
}
```

Если аргумент-формула передан методу `evalList` в виде диапазона, имеющего тип `Range`, возвращаемое значение будет представлять собой список, состоящий из значений всех ячеек, ссылки на которые попадают в данный диапазон. Для любой другой формулы результатом будет список, состоящий из одного значения, получившегося в результате вычисления по этой формуле. Ячейки, на которые ссылается формула, вычисляются третьей функцией, `references`. Ее определение выглядит следующим образом:

```
  def references(e: Formula): List[Cell] = e match {
    case Coord(row, column) =>
      List(cells(row)(column))
    case Range(Coord(r1, c1), Coord(r2, c2)) =>
      for (row <- (r1 to r2).toList; column <- c1 to c2)
      yield cells(row)(column)
    case Application(function, arguments) =>
      arguments flatMap references
    case _ =>
      List()
  }
} // Окончание трейта Evaluator
```

Метод `references` пока имеет более общий характер, чем необходимо. Это выражается в том, что он вычисляет список ячеек, на которые ссылается любая формула, а не только формула типа `Range`. Чуть позже обнаружится, что эта дополнительная функциональная возможность требуется для вычисления набо-

ров ячеек, нуждающихся в обновлении. Тело метода представляет собой простое сопоставление с образцом в отношении разновидностей формул. Для координат `Coord(row, column)` оно возвращает одноэлементный список, содержащий ячейку, имеющую эти координаты. Для выражения диапазона `Range(coord1, coord2)` это тело возвращает все ячейки между двумя координатами, вычисляемые с помощью выражения `for`. Для применения функции `Application(function, arguments)` оно возвращает ячейки, на которые имеются ссылки в каждом аргументе выражения, объединенные с помощью функции `flatMap` в единый список. Для остальных двух типов формул, `Textual` и `Number`, оно возвращает пустой список.

35.6. Библиотеки операций

В самом классе `Evaluator` не определены операции, которые могут выполняться в отношении ячеек, — его таблица операций изначально пуста. Замысел заключается в определении таких операций в других трейтах, которые затем примешиваются в класс `Model`. Пример трейта, реализующего обычные арифметические операции, показан в листинге 35.8.

Листинг 35.8. Библиотека арифметических операций

```
package org.stairwaybook.scells
trait Arithmetic { this: Evaluator =>
  operations ++= List(
    "add"  -> { case List(x, y) => x + y },
    "sub"  -> { case List(x, y) => x - y },
    "div"  -> { case List(x, y) => x / y },
    "mul"  -> { case List(x, y) => x * y },
    "mod"  -> { case List(x, y) => x % y },
    "sum"  -> { xs => xs.foldLeft(0.0)(_ + _) },
    "prod" -> { xs => xs.foldLeft(1.0)(_ * _) }
  )
}
```

Примечательно, что у этого трейта нет экспортированных членов. Он всего лишь наполняет таблицу `operations` в ходе инициализации. Он получает доступ к таблице за счет использования собственного типа, принадлежащего классу `Evaluator`, то есть задействует ту же технологию доступа к модели, которая применяется в классе `Arithmetic`.

Из семи операций, определенных трейтом `Arithmetic`, пять являются бинарными, а две получают произвольное количество аргументов. Все бинарные операции действуют по одной и той же схеме. Например, операция сложения `add` определена с помощью следующего выражения:

```
{ case List(x, y) => x + y }
```

То есть она ожидает список аргументов, содержащий два элемента, `x` и `y`, и возвращает сумму `x` и `y`. Если в списке аргументов содержатся не два аргумента,

то выдается исключение MatchError. Это соответствует общей философии «пусть будет сбой» (let it crush), которой придерживается модель вычисления, принятая в приложении SCell. Согласно ей, неверный ввод в ходе выполнения программы ожидаемо приводит к генерации исключения, которое затем отлавливается конструкцией try-catch, имеющейся внутри метода вычисления.

Последние две операции, sum и prod, получают список аргументов произвольной длины и вставляют бинарные операции между идущими друг за другом элементами. То есть они являются экземплярами схемы правой свертки, которая выражена в классе List операцией /:. Например, чтобы выполнить сложение списка чисел List(x, y, z), операция вычисляет 0 + x + y + z . Первый операнд 0 становится результатом в том случае, если список пуст.

Эту библиотеку операций можно встроить в приложение электронной таблицы путем примешивания трейта Arithmetic в класс Model:

```
package org.stairwaybook.scells

class Model(val height: Int, val width: Int)
    extends Evaluator with Arithmetic {

  case class Cell(row: Int, column: Int) {
    var formula: Formula = Empty
    def value = evaluate(formula)

    override def toString = formula match {
      case Textual(s) => s
      case _ => value.toString
    }
  }

   ... // Все остальное, как и прежде
}
```

Еще одно изменение в классе Model касается способа отображения самих ячеек. В новой версии отображаемое значение ячейки зависит от ее формулы. Если формула представлена полем типа Textual, то содержимое поля отображается буквально. Во всех остальных случаях формула вычисляется и отображается результат этого вычисления.

Если скомпилировать измененные трейты и классы и заново запустить программу Main, то получится уже что-то напоминающее настоящую электронную таблицу. Пример показан на рис. 35.4. В ячейки можно вводить формулы и заставлять выполнять по ним вычисления. Например, как только будет закрыт фокус редактирования ячейки C5, в ней будет показано число 86.0, являющееся результатом вычисления формулы sum(C1:C4).

Но в электронной таблице по-прежнему отсутствует весьма важный элемент. Если изменить значение ячейки C1 с 20 на 100, то сумма в ячейке C5 не будет автоматически обновлена до 166. Чтобы увидеть изменения в C5, на этой ячейке придется щелкать вручную. У нас пока нет способа заставить ячейки после изменения вычислять свои значения в автоматическом режиме.

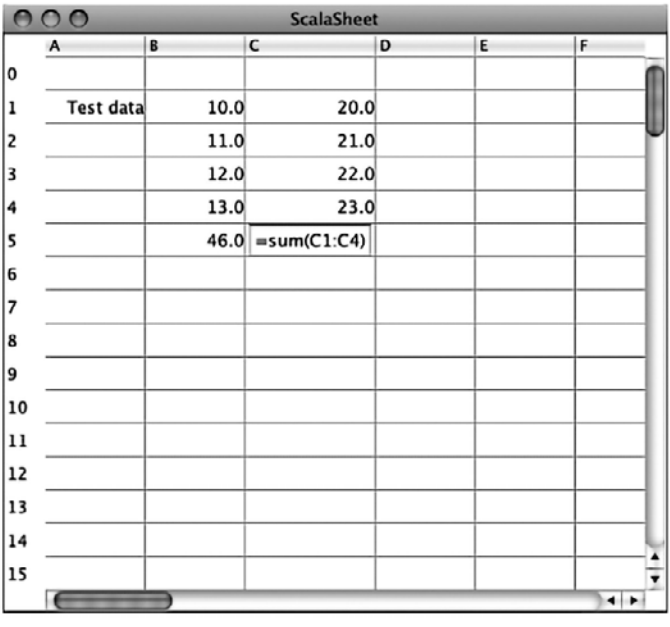


Рис. 35.4. Вычисляемые ячейки

35.7. Распространение изменений

Если значение ячейки изменилось, то все ячейки, значения которых от него зависят, должны заново вычислить и показать свои результаты. Проще всего добиться этого повторным вычислением каждой ячейки электронной таблицы после каждого изменения. Но такой подход плохо поддается масштабированию в случае разрастания электронной таблицы.

Более удачным будет подход, при котором заново вычисляются значения только тех ячеек, в формулах которых имеются ссылки на измененную ячейку. Он заключается в том, чтобы для распространения изменения воспользоваться фреймворком публикаций-подписок на основе событий: как только ячейке присваивается формула, она будет подписываться на уведомления обо всех изменениях в тех ячейках, на которые ссылается эта формула. Значение, измененное в одной из таких ячеек, запустит перевычисление подписавшейся ячейки. Если такое повторное вычисление вызовет изменение в значении ячейки, то она, в свою очередь, уведомит все ячейки, которые зависят от этого изменения. Процесс продолжится до тех пор, пока все значения ячеек не стабилизируются, то есть значения любых ячеек не прекратят изменяться[1].

[1] Здесь предполагается, что между ячейками нет циклических зависимостей. Мы обсудим отказ от этого предположения в конце главы.

Фреймворк публикаций-подписок реализован в классе `Model` с помощью стандартного механизма событий, имеющегося в Scala Swing. Новая (окончательная) версия этого класса выглядит так:

```
package org.stairwaybook.scells
import swing._

class Model(val height: Int, val width: Int)
extends Evaluator with Arithmetic {
```

По сравнению с предыдущей версией `Model` в эту добавлен импорт `swing._`, позволяющий обращаться к абстракциям событий фреймворка Swing напрямую. Основное изменение класса `Model` коснулось вложенного класса `Cell`. Теперь класс `Cell` является наследником класса `Publisher`, что позволяет ему публиковать события. Логика обработки событий целиком находится в `set`-методах двух свойств: `value` и `formul`. Новая версия `Cell` выглядит так:

```
case class Cell(row: Int, column: Int) extends Publisher {
  override def toString = formula match {
    case Textual(s) => s
    case _ => value.toString
}
```

Похоже, `value` и `formul` — две переменные в классе `Cell`. Они реализуются в двух приватных полях, оснащенных публичными `get`-методами `value` и `formula` и `set`-методами `value_=` и `formula_=`. Определение реализации свойства `value` имеет следующий вид:

```
private var v: Double = 0
def value: Double = v
def value_=(w: Double) = {
  if (!(v == w || v.isNaN && w.isNaN)) {
    v = w
    publish(ValueChanged(this))
  }
}
```

`Set`-метод `value_=` присваивает новое значение `w` приватному полю `v`. Если новое значение отличается от старого, то он также публикует событие `ValueChanged` с самой ячейкой в качестве аргумента. Следует заметить, что тест на изменение значения немного усложнен, поскольку в нем используется значение `NaN`. В спецификации языка Java говорится, что `NaN` отличается от любого другого значения, включая само себя! Поэтому равенство двух значений `NaN` рассматривается особым образом: значения `v` и `w` одинаковы, если равны в отношении применения оператора `==` или оба являются значениями `NaN`, то есть оба выражения, `v.isNaN` и `w.isNaN`, выдают значение `true`.

`Set`-функция `value_=` выполняет во фреймворке публикаций-подписок публикацию, в то время как `set`-функция `formula_=` осуществляет подписку:

```
private var f: Formula = Empty
def formula: Formula = f
```

```
def formula_=(f: Formula) = {
  for (c <- references(formula)) deafTo(c)
  this.f = f
  for (c <- references(formula)) listenTo(c)
  value = evaluate(f)
}
```

Если ячейке присваивается новая формула, то она сначала отписывается с помощью метода `deafTo` от событий во всех ячейках, на которые ссылалась прежняя формула. Затем в ней в приватной переменной `f` сохраняется новая формула, а с помощью метода `listenTo` выполняется подписка на события во всех ячейках, на которые имеются ссылки в этой формуле. И наконец, значение данной ячейки вычисляется с использованием новой формулы.

В последнем фрагменте кода в переделанном классе `Cell` указывается, как следует реагировать на событие `ValueChanged`:

```
  reactions += {
    case ValueChanged(_) => value = evaluate(formula)
  }
} // Окончание класса Cell
```

Класс `ValueChanged` также содержится в классе `Model`:

```
case class ValueChanged(cell: Cell) extends event.Event
```

Остальная часть класса `Model` остается неизменной:

```
  val cells = Array.ofDim[Cell](height, width)

  for (i <- 0 until height; j <- 0 until width)
    cells(i)(j) = new Cell(i, j)
} // Окончание класса Model
```

Теперь разработка кода электронной таблицы почти завершена. Заключительный отсутствующий фрагмент относится к повторному отображению измененных ячеек. До сих пор все распространение значений касалось только внутренних `Cell`-значений, видимую таблицу они не затрагивали. Одним из способов изменения видимого представления может стать добавление к `set`-функции `value_=` команды перерисовки `redraw`. Но тогда нарушится наблюдавшееся до сих пор четкое разделение между моделью и представлением. Более модульное решение — уведомить таблицу обо всех событиях `ValueChanged` и позволить ей выполнять перерисовку самостоятельно. Последний компонент электронной таблицы, реализующий эту схему, показан в листинге 35.9.

Листинг 35.9. Законченный компонент электронной таблицы

```
package org.stairwaybook.scells
import swing._, event._

class Spreadsheet(val height: Int, val width: Int)
    extends ScrollPane {
```

```
  val cellModel = new Model(height, width)
  import cellModel._

  val table = new Table(height, width) {
    ... // Все установки как в листинге 35.1

    override def rendererComponent(
         isSelected: Boolean, hasFocus: Boolean,
         row: Int, column: Int) =
      ... // Как в листинге 35.3

    def userData(row: Int, column: Int): String =
      ... // Как в листинге 35.3

    reactions += {
      case TableUpdated(table, rows, column) =>
        for (row <- rows)
          cells(row)(column).formula =
             FormulaParsers.parse(userData(row, column))
      case ValueChanged(cell) =>
        updateCell(cell.row, cell.column)
    }

    for (row <- cells; cell <- row) listenTo(cell)
  }

  val rowHeader = new ListView(0 until height) {
    fixedCellWidth = 30
    fixedCellHeight = table.rowHeight
  }

  viewportView = table
  rowHeaderView = rowHeader
}
```

Класс `Spreadsheet`, показанный в этом листинге, содержит только два пересмотренных фрагмента. Во-первых, теперь компонент `table` с помощью метода `listenTo` подписан на события, происходящие во всех ячейках модели. Во-вторых, в реакциях таблицы появился новый `case`-вариант: если поступает уведомление о событии `ValueChanged(cell)`, то выдается запрос на перерисовку соответствующей ячейки вызовом метода `updateCell(cell.row, cell.column)`.

Резюме

Разработанная в данной главе электронная таблица полностью функциональна, хотя в некоторых местах применяются не самые удобные для пользователя, но самые простые варианты реализации. Это позволило уместить код менее чем в 200 строках. Несмотря на это, архитектура электронной таблицы позволяет

легко выполнять ее модификацию и расширение. Если захочется продолжить эксперименты с кодом, то в качестве возможных изменений и добавлений предлагаем выполнить следующие действия.

1. Сделать электронную таблицу изменяемой в размерах, чтобы количество строк и столбцов корректировалось в интерактивном режиме.
2. Добавить новые виды формул, например бинарные операции или другие функции.
3. Подумать над тем, что делать, когда ячейки рекурсивно ссылаются друг на друга. Например, если ячейка `A1` содержит формулу `add(B1, 1)`, а ячейка `B1` — формулу `mul(A1, 2)`, повторное вычисление любой ячейки приведет к переполнению стека. Понятно, что ничего хорошего в таком решении нет. В качестве альтернативных решений можно либо не допустить подобной ситуации, либо просто выполнять только одну итерацию при каждом применении одной из ячеек.
4. Усовершенствовать систему обработки ошибок, выдавая более подробные сообщения, описывающие причины сбоев.
5. Добавить в верхнюю часть таблицы поле ввода формул, чтобы упростить ввод длинных формул.

В начале книги подчеркивался аспект масштабируемости программ на Scala. Утверждалось, что сочетание допускаемых в Scala объектно-ориентированных и функциональных конструкций делает язык подходящим для программ в диапазоне от небольших скриптов до очень крупных систем. Очевидно, что представленная здесь электронная таблица относится к небольшим системам, даже если учесть, что при ее создании с использованием большинства других языков программирования количество строк кода наверняка намного превышало бы 200. Несмотря на это, при работе над данным приложением вам представилась возможность ознакомиться со многими подробностями языка Scala, позволяющими расширять масштабы приложений.

В электронной таблице использовались классы и трейты Scala с их композицией примесей для объединения компонентов весьма гибкими способами. Рекурсивные зависимости между компонентами выражались с помощью собственных типов. Потребность в статическом состоянии была полностью исключена, единственными компонентами верхнего уровня, не являющимися классами, были деревья формул и парсеры формул, и все они имели чисто функциональный характер. Кроме того, в приложении широко использовались функции высшего порядка и сопоставление с образцом, которые применялись как при обращении к формулам, так и при обработке событий. В итоге у нас неплохо получилось продемонстрировать то, как легко можно объединить функциональное и объектно-ориентированное программирование.

Одной из причин такой лаконичности приложения электронной таблицы является то, что оно базируется на весьма мощных библиотеках. Библиотека комбинаторов парсеров, по сути, предоставила внутренний предметно-ориентированный

язык написания парсеров. Без него выполнить синтаксический разбор формул было бы гораздо сложнее. Обработка событий в имеющейся в Scala библиотеке Swing стала хорошим примером эффективности управляющих абстракций. Если вам знакомы библиотеки Swing, входящие в Java, то вы, вероятно, сможете по достоинству оценить краткость концепции реакций в Scala, особенно по сравнению с рутиной создания методов оповещения и реализации интерфейсов слушателей в классическом шаблоне проектирования публикаций-подписок. Таким образом, электронная таблица позволила продемонстрировать преимущества расширяемости, при которой библиотеки верхнего уровня могут создаваться очень похожими на расширения самого языка.

Приложение. Скрипты Scala на Unix и Windows

В Unix можно запустить скрипт на Scala в качестве скрипта оболочки, предваряя этот запуск шебанг-инструкцией в самом начале файла. Наберите, к примеру, для файла `helloarg` следующий код:

```
#!/bin/sh
exec scala "$0" "$@"
!#
// Поприветствуйте первый аргумент
println("Hello, " + args(0) + "!")
```

Начальная команда `#!/bin/sh` должна быть самой первой строкой файла. Как только будет установлено разрешение на выполнение:

```
$ chmod +x helloarg
```

можно будет запустить скрипт на Scala в качестве скрипта оболочки простой командой:

```
$ ./helloarg globe
```

В Windows такого же эффекта можно достичь, назвав файл `helloarg.bat` и поместив в верхней части своего скрипта следующий код:

```
::#!
@echo off
call scala %0 % *
goto :eof
::!#
```

Глоссарий

- *Алгебраический тип данных* (algebraic data type). Тип, определяемый путем предоставления нескольких альтернатив, у каждой из которых есть собственный конструктор. Обычно предоставляется возможность его декомпозиции через сопоставление с образцом. Эта концепция встречается в языках спецификаций и функциональных языках программирования. В Scala алгебраические типы данных можно эмулировать с помощью `case`-классов.
- *Альтернатива* (alternative). Ветвь выражения `match`. Имеет вид «`case` *паттерн => выражение*». Альтернативу также могут называть *вариантом* (или *case*).
- *Аннотация* (annotation). Встречается в исходном коде и закрепляется за какой-то частью синтаксиса. Аннотации обрабатываются компьютером, поэтому их фактически можно использовать для расширения Scala.
- *Анонимная функция* (anonymous function). Альтернативное название функционального литерала.
- *Анонимный класс* (anonymous class). Синтетический подкласс, сгенерированный компилятором Scala из выражения `new`, в котором вслед за именем класса или трейта идут фигурные скобки. В них содержится тело анонимного подкласса, которое может быть пустым. Но если имя, указанное после выражения `new`, ссылается на трейт или класс с абстрактными членами, то эти члены необходимо конкретизировать внутри фигурных скобок, определяющих тело анонимного подкласса.
- *Аргумент* (argument). При вызове функции каждому ее параметру соответствует передаваемый *аргумент*. Параметр — это переменная, которая ссылается на аргумент. Аргумент — это объект, который передается во время вызова. Кроме того, приложения могут принимать аргументы (командной строки), содержащиеся в массиве `Array[String]`, который передается методам `main` объектов-одиночек.
- *Блок* (block). Одно или несколько выражений или объявлений, заключенных в фигурные скобки. При вычислении блока все его выражения и объявления обрабатываются по очереди, после чего блок возвращает в качестве результата значение последнего выражения. Блоки обычно используются в качестве тел функции, выражений `for`, циклов `while` и в любых других местах, где нужно сгруппировать какое-то количество инструкций. Если говорить более формально, то блок — это конструкция для инкапсуляции, в которой видны только побочные эффекты и итоговое значение. Таким образом, фигурные скобки,

в которых определяется класс или объект, не формируют блок, поскольку поля и методы (определенные внутри этих скобок) видны снаружи. Такие фигурные скобки составляют *шаблон*.

- *Вариантность* (variance). Параметр типа для класса или трейта можно пометить с помощью аннотации *вариантности*: либо как *ковариантный* (`+`), либо как *контравариантный* (`-`). Такие аннотации определяют, как у обобщенного класса или трейта работает отношение наследования. Например, обобщенный класс `List` ковариантен в своем параметре типа, поэтому `List[String]` является подтипом `List[Any]`. По умолчанию, если не указывать аннотации `+` или `-`, то параметры типов *нонвариантны*.
- *Виртуальная машина Java* (Java Virtual Machine, JVM). *Среда выполнения*, в которой работают программы на языке Scala.
- *Возврат* (return). В программах на языке Scala функция *возвращает* значение. Вы можете называть его *результатом* функции. Кроме того, можно сказать, что функция *приводит к* значению. Результат любой функции в Scala — объект.
- *Вспомогательная функция* (helper function). Предназначена для предоставления каких-либо возможностей для одной или нескольких других функций поблизости. Вспомогательные функции часто реализуются как локальные.
- *Вспомогательный конструктор* (auxiliary constructor). Внутри фигурных скобок с определением класса можно указывать дополнительные конструкторы, которые выглядят как определения методов с именем `this`, но без результирующего типа.
- *Вспомогательный метод* (helper method). Вспомогательная функция, являющаяся членом класса. Вспомогательные методы зачастую являются приватными.
- *Выдача* (yield). Выражение может *выдавать* (yield) результат. Ключевое слово `yield` обозначает результат выражения `for`.
- *Вызов* (invoke). Вы можете *вызвать* метод, функцию или замыкание *с* аргументами — то есть при выполнении их тела будут использованы заданные аргументы.
- *Выражение* (expression). Любой фрагмент кода в Scala дает какой-либо результат. Можно сказать, что результат *вычисляется из* выражения или выражение *возвращает* значение.
- *Выражение генератора* (generator expression). Генерирует последовательность значений в выражении `for`. Например, в `for(i <- 1 to 10)` выражением генератора выступает `1 to 10`.
- *Генератор* (generator). Определяет именованное значение и последовательно присваивает ему результаты выражения `for`. Например, в `for(i <- 1 to 10)` генератором выступает `i <- 1 to 10`. Значение справа от `<-` — это *выражение генератора*.
- *Замыкание* (closure). Функциональный объект, который захватывает свободные переменные и как будто замыкается вокруг переменных, видимых в момент его создания.

- *Значение* (value). Результат любого вычисления или выражения в Scala. При этом все значения в Scala являются объектами. Значение, в сущности, — это образ объекта в памяти (в куче JVM или стеке).
- *Императивный стиль* (imperative style). В этом стиле программирования акцент делается на том, в какой последовательности выполняются операции, чтобы их побочные эффекты проявились в правильном порядке. Этот стиль характеризуется итерацией с циклами, изменением данных без копирования и методами с побочными эффектами. Эта парадигма доминирует в таких языках, как C, C++, C# и Java, контрастируя с *функциональным стилем*.
- *Инвариант* (invariant). Термин имеет два значения. Это может быть свойство, которое всегда остается истинным при условии, что структура данных имеет правильный формат. Например, инвариантом отсортированного двоичного дерева может быть требование, согласно которому каждый узел должен быть упорядочен перед своим правым подузлом, если таковой имеется. Термин *«инвариант»* также иногда служит синонимом нонварианта: «класс `Array` является инвариантным в своем параметре типа».
- *Инициализация* (initialize). При определении переменной в исходном коде Scala вы должны *инициализировать* ее с помощью какого-либо объекта.
- *Инструкция* (statement). Выражение, определение или импорт — то есть действия, которые могут быть заданы в шаблоне или блоке исходного кода Scala.
- *Карринг* (currying). Способ написания функций с несколькими списками параметров. Например, `def f(x: Int)(y: Int)` — это каррированная функция с двумя списками параметров. Чтобы применить каррированную функцию, ей нужно передать несколько списков аргументов, как в `f(3)(4)`. Но мы можем также выполнить *частичное применение* каррированной функции, как в `f(3)`.
- *Класс* (class). Определяется с помощью ключевого слова `class` и может быть либо абстрактным, либо конкретным. Во время создания экземпляра класса его можно параметризовать с указанием типов и значений. В выражении `new Array[String](2)` создается экземпляр класса `Array`, а тип получаемого значения — `Array[String]`. Класс, который принимает параметры типов, называется *конструктором типа*. У типа тоже может быть класс — например, классом типа `Array[String]` является `Array`.
- *Класс-компаньон* (companion class). Класс с тем же именем, что и у объекта-синглтона, определенного в том же исходном файле. Это класс-компаньон объекта-одиночки.
- *Ковариантность* (covariant). Аннотацию *ковариантности* можно применить к параметру типа в классе или трейте, указав перед ним знак плюс (`+`). В результате класс или трейт формирует ковариантные взаимоотношения с аннотированным параметром типа (направленные в ту же сторону). Например, если класс `List` является ковариантным в своем первом параметре типа, то `List[String]` будет подтипом `List[Any]`.

- *Композиция примесей* (mixin composition). Процесс примешивания трейтов в классы или другие трейты. От традиционного множественного наследования *композиция примесей* отличается тем, что тип ссылки `super` неизвестен в момент определения трейта и определяется каждый раз, когда трейт примешивается в класс или другой трейт.
- *Конструктор типа* (type constructor). Класс или трейт, принимающий параметры типов.
- *Контравариантность* (contravariant). Аннотацию *контравариантности* можно применить к любому параметру типа в классе или трейте, указав перед ним знак минус (`-`). В результате класс или трейт формирует контравариантные взаимоотношения с аннотированным параметром типа (направленные в противоположную от него сторону). Например, если класс `Function1` является контравариантным в своем первом параметре типа, то `Function1[Any, Any]` будет подтипом `Function1[String, Any]`.
- *Литерал* (literal). `1`, `"Один"` и `(x: Int) => x + 1` — это все примеры *литералов*. Это краткое описание объекта, которое в точности отражает структуру созданного объекта.
- *Локальная переменная* (local variable). `val` или `var`, определенные внутри блока. Параметры функции, несмотря на свою схожесть с локальными переменными, таковыми не считаются; их называют просто параметрами или переменными.
- *Локальная функция* (local function). Функция, определенная внутри блока. Тогда как функция, определенная как член класса, трейта или объекта-одиночки, называется *методом*.
- *Метапрограммирование* (meta-programming). Вид программирования, в котором программы принимают на вход другие программы. Компиляторы и инструменты наподобие `scaladoc` являются метапрограммами. Средства метапрограммирования необходимы для работы с *аннотациями*.
- *Метод* (method). Функция, которая является членом какого-то класса, трейта или объекта-одиночки.
- *Метод без параметров* (parameterless method). Это функция без параметров, которая является членом класса, трейта или объекта-одиночки.
- *Множественные определения* (multiple definitions). Одно и то же выражение может быть присвоено в *нескольких определениях*, если используется синтаксис вида `val v1, v2, v3 = exp`.
- *Модификатор* (modifier). Ключевое слово, которое каким-то образом ограничивает определение класса, трейта, поля или метода. Например, модификатор `private` говорит о том, что определяемый класс, трейт, поле или метод является приватным.
- *Недоступность* (unreachable). На уровне Scala объекты могут становиться недоступными; в этом случае среда выполнения может освободить память, которую

они занимают. Недоступность вовсе не означает отсутствие ссылок. Ссылочные типы (экземпляры `AnyRef`) реализуются в виде объектов, размещенных в куче JVM. Когда экземпляр ссылочного типа становится недоступным, на него и в самом деле ничего не ссылается, что позволяет сборщику мусора его освободить. Типы значений (экземпляры `AnyVal`) реализуются как с помощью примитивных типов, так и в виде типов-оберток (таких как `java.lang.Integer`), которые находятся в куче. Экземпляры типов значений могут упаковываться (превращаться из примитивного значения в объект-обертку) и распаковываться (превращаться из объекта-обертки в примитивное значение) на протяжении существования переменной, которая на них ссылается. Если экземпляр типа значения, который в настоящий момент представлен объектом-оберткой в куче JVM, становится недоступным, то на него и в самом деле ничего не ссылается, что позволяет сборщику мусора его освободить. Но если в настоящий момент значение имеет примитивный тип и становится недоступным, то это не означает, что на него больше ничего не ссылается, поскольку он больше не существует в виде объекта в куче JVM. Среда выполнения может освободить память, занятую недоступными объектами. Но если тип `Int`, к примеру, реализован во время выполнения как примитивное значение `int` из языка Java и занимает какую-то память в стековом фрейме выполняющегося метода, то эта память освобождается, только когда фрейм достается из стека при завершении метода. Память для ссылочных типов, таких как `String`, может быть освобождена сборщиком мусора JVM после того, как они станут недоступными.

- *Неизменяемость* (immutable). Объект является *неизменяемым*, если после того, как он был создан, его значение нельзя изменить никаким видимым для клиентов образом. Объекты могут быть неизменяемыми, но не обязательно.
- *Нонвариант* (nonvariant). Параметр типа класса или трейта по умолчанию является *нонвариантом*. Когда этот параметр меняется, класс или трейт не производит подтип. Например, класс `Array` является нонвариантом в своем параметре типа, поэтому `Array[String]` нельзя считать ни подтипом, ни супертипом `Array[Any]`.
- *Обобщенный класс* (generic class). Класс, принимающий параметры типа. Например, класс `scala.List` принимает параметр типа, поэтому является обобщенным.
- *Обобщенный трейт* (generic trait). Трейт, принимающий параметры типа. Например, трейт `scala.collection.Set` принимает параметр типа, поэтому является обобщенным.
- *Образец (паттерн)* (pattern). В выражении `match` *образца* указывается после каждого ключевого слова `case`, но перед *ограждением образца* (или символом `=>`).
- *Объект без ссылки* (unreferenced). См. *Недоступность*.
- *Объект-компаньон* (companion object). Объект-одиночка с тем же именем, что и у класса, определенного в том же исходном файле. Объекты-компаньоны и классы-компаньоны имеют доступ к приватным членам друг друга. Кроме

того, любое неявное приведение типов, определенное в объекте-компаньоне, будет находиться в области видимости кода, в котором используется класс.

- *Объект-одиночка* (singleton object). Объект, в определении которого указано ключевое слово `object`. У каждого объекта-одиночки есть один и только один экземпляр. Если он находится в одном исходном файле с одноименным классом, то его называют *объектом-компаньоном* данного класса. При этом класс является его *классом-компаньоном*. Объект-одиночка, у которого нет класса-компаньона, является *самостоятельным объектом*.
- *Объявление* (declare). Вы можете *объявить* абстрактное поле, метод или тип, указав имя сущности, но не ее реализацию. Ключевое отличие между объявлением и определением в том, что последнее, в отличие от первого, создает реализацию именованной сущности.
- *Ограждение образца* (pattern guard). В выражении `match` за образцом может следовать его *ограждение*. Например, в `case x if x % 2 == 0 => x + 1` ограждением *образца* выступает `if x % 2 == 0`. Блок `case` с ограждением *образца* выбирается только в случае соответствия *образцу*, и если его ограждение возвращает `true`.
- *Ограничение типа* (type constraint). Некоторые аннотации выступают *ограничениями типа*; это значит, они накладывают дополнительные ограничения на то, какие значения охватывает тип. Например, `@positive` может ограничивать 32-битный целочисленный тип `Int` положительными значениями. Ограничения типов проверяются не стандартным компилятором Scala, а его подключаемым модулем или дополнительным инструментом.
- *Операция* (operation). В Scala каждая *операция* представляет собой вызов метода. Методы могут вызываться с помощью *синтаксиса оператора*; например, в выражении `b + 2` знак `+` — это *оператор*.
- *Определение* (define). В программе на языке Scala *определение* состоит из имени и реализации. Вы можете определять классы, трейты, объекты-одиночки, поля, методы, локальные функции, локальные переменные и т. д. Определение всегда подразумевает какую-то реализацию, поэтому абстрактные члены не определяются, а *объявляются*.
- *Параметр* (parameter). Функции могут принимать любое количество *параметров* (от нуля и больше). У каждого параметра есть имя и тип. Если сравнивать аргументы и параметры, то первые ссылаются непосредственно на объекты, которые передаются при вызове функции, а вторые являются переменными, которые ссылаются на эти передаваемые аргументы.
- *Параметр типа* (type parameter). Параметр обобщенного класса или метода, в качестве которого должен быть указан тип. Например, класс `List` определен как `class List[T] { ...`, а метод `identity` объекта `Predef` имеет определение `def identity[T](x:T) = x`. В обоих случаях `T` — это параметр типа.
- *Параметр, передаваемый по значению* (by-value parameter). Параметр, перед типом которого *не* указано `a =>`; например, `(x: Int)`. Аргумент, относящийся

к такому параметру, вычисляется перед вызовом метода. Это альтернатива передаче параметров *по имени*.

- *Параметр, передаваемый по имени* (by-name parameter). Параметр, перед типом которого указано `a =>`, например `(x: => Int)`. Аргумент, относящийся к такому параметру, вычисляется не перед вызовом метода, а каждый раз, когда в методе на него ссылаются *по имени*. Если параметр не передается по имени, то передается *по значению*.
- *Параметрическое поле* (parametric field). Поле, определенное как параметр класса.
- *Первичный конструктор* (primary constructor). Главный конструктор класса, который вызывает конструктор суперкласса, при необходимости инициализирует поля с использованием переданных значений и выполняет любой высокоуровневый код, определенный в фигурных скобках класса. Поля инициализируются только для тех параметров, которые не были переданы в суперконструктор; исключение составляют поля, которые не используются в теле класса и, следовательно, могут быть опущены.
- *Перекрытие* (shadow). Новое определение локальной переменной *перекрывает* переменную с тем же именем, находящуюся в наружной области видимости.
- *Переменная* (variable). Именованная сущность, которая ссылается на объект. Переменная может быть определена как `val` или `var`. В обоих случаях при определении требуется инициализация, но только `var` позже можно будет назначить ссылку на другой объект.
- *Переназначаемость* (reassignable). Переменная может быть *переназначаемой*. Значения `var` переназначаемые, а `val` — нет.
- *Подкласс* (subclass). Класс является *подклассом* всех своих суперклассов и трейтов.
- *Подстановочный тип* (wildcard type). Включает ссылки на неизвестные переменные типов. Например, `Array[_]` является подстановочным типом. Это массив, о типе элементов которого ничего не известно.
- *Подтип* (subtype). Компилятор Scala позволяет использовать вместо типа любые его *подтипы*. Если классы и трейты не принимают никаких параметров типов, то отношение подпипа является зеркальным отражением отношения подкласса. Например, если класс `Cat` является подклассом абстрактного класса `Animal` и ни один из них не принимает параметры типов, то тип `Cat` будет подтипом `Animal`. Точно так же если трейт `Apple` является подтрейтом трейта `Fruit` и ни один из них не принимает параметры типов, то тип `Apple` будет подтипом `Fruit`. Но если классы и трейты принимают параметры типов, то в силу вступает вариантность. Например, поскольку абстрактный класс `List` является ковариантным в своем единственном параметре типа (то есть объявлен как `List[+A]`), `List[Cat]` будет подтипом `List[Animal]`, а `List[Apple]` — подтипом `List[Fruit]`. Эти отношения подтипов существуют несмотря на то, что классом каждого из этих типов выступает `List`. Для сравнения, поскольку класс `Set` не является ковариантным

в своем параметре типа (объявлен как `Set[A]`, без знака плюс), `Set[Cat]` *не* будет подтипом `Set[Animal]`. Подтип должен правильно реализовывать контракты своих супертипов с соблюдением принципа подстановки Лисков, но компилятор проверяет это только на уровне типа.

- *Подтрейт* (subtrait). Трейт является *подтрейтом* всех своих супертрейтов.
- *Предикат* (predicate). Функция, возвращающая тип `Boolean`.
- *Применение* (apply). Вы можете *применить* метод, функцию или замыкание *к* аргументам — то есть вызвать их для этих аргументов.
- *Примесь* (mixin). Так называют трейт, который используется в композиции примесей. Иными словами, `Hat` в `trait Hat` является обычным трейтом, а в `new Cat extends AnyRef with Hat` его можно назвать примесью. Этот термин можно использовать как глагол. Например, вы можете *примешать* трейты *в* классы или другие трейты.
- *Принцип единообразного доступа* (uniform access principle). Согласно этому принципу, для доступа к переменным и функциям, у которых нет параметров, должен использоваться один и тот же синтаксис. Scala поддерживает данный принцип, не позволяя указывать скобки при вызове функций без параметров. Благодаря этому вместо определения функции без параметров можно подставить `val` и *наоборот*, не затрагивая клиентский код.
- *Присваивание* (assign). Вы можете *присвоить* объект переменной. После этого переменная будет ссылаться на данный объект.
- *Процедура* (procedure). Функция, возвращающая тип `Unit`, которая выполняется исключительно для получения побочных эффектов.
- *Прямой подкласс* (direct subclass). Класс, являющийся *прямым наследником* своего суперкласса.
- *Прямой суперкласс* (direct superclass). Класс, непосредственной производной которого является другой класс или трейт — то есть ближайший класс, который находится сверху от него в иерархии наследования. Если класс `Parent` указан в необязательной инструкции `extends` класса `Child`, то это значит, что `Parent` является прямым суперклассом `Child`. Если в инструкции `extends` класса `Child` указан трейт, то данный трейт будет прямым суперклассом `Child`. Если У `Child` нет инструкции `extends`, его прямым родителем является `AnyRef`. Прямой суперкласс может принимать параметры типов — например, класс `Child` может расширять `Parent[String]`; в этом случае прямым суперклассом `Child` по-прежнему будет `Parent`, а не `Parent[String]`. С другой стороны, `Parent[String]` будет прямым *супертипом* `Child`. Чтобы узнать больше о различиях между классами и типами, см. *Супертип*.
- *Равенство* (equality). Отношение между значениями, выраженное как `==` (при отсутствии уточняющих параметров). См. также *Равенство ссылок*.
- *Равенство ссылок* (reference equality). Означает, что две ссылки идентифицируют один и тот же объект Java. В случае с ссылочными типами равенство

ссылок можно определить с помощью вызова `eq` в `AnyRef` (в Java-программах для определения равенства ссылок можно использовать `==`, но тоже только для ссылочных типов).

- *Результат* (result). Выражения в программах на языке Scala дают *результат*. Результат любого выражения в Scala — объект.
- *Результирующий (в Scala), возвращаемый (в Java) тип* (result type). Результирующий/возвращаемый тип метода — тип значения, которое возвращается в результате вызова этого метода.
- *Рекурсия* (recursive). Функция, которая вызывает саму себя, называется *рекурсивной*. Если этот вызов происходит только в последнем выражении функции, то она является хвостовой рекурсией.
- *Самостоятельный объект* (standalone object). Объект-одиночка без класса-компаньона.
- *Свободная переменная* (free variable). Переменная называется *свободной*, если используется в выражении, но не объявлена внутри него. Например, в выражении функционального литерала `(x: Int) => (x, y)` используются обе переменные, `x` и `y`, но только `y` является свободной, так как не определена внутри этого выражения.
- *Связанная переменная* (bound variable). Переменная, которая определена и используется внутри выражения, является его *связанной переменной*. Например, в выражении функционального литерала `(x: Int) => (x, y)` используются две переменные, `x` и `y`, но только `x` является связанной, поскольку определена в выражении как `Int` и выступает единственным аргументом функции, описанной этим выражением.
- *Селектор* (selector). Значение, с которым сопоставляются образцы в выражении `match`. Например, в `s match { case _ => }` селектором выступает `s`.
- *Сериализация* (serialization). Объект можно *сериализовать* в поток байтов, который затем может быть сохранен в файлы или передан по сети. Позже поток байтов можно будет *десериализовать*, даже находясь на другом компьютере, и получить объект, который идентичен сериализованному оригиналу.
- *Сигнатура* (signature). Сокращенный вариант понятия *«сигнатура типа»*.
- *Сигнатура типа* (type signature). Сигнатура типа метода определяет его имя, а также количество, порядок и типы его параметров (если таковые имеются) и возвращаемый тип. Сигнатура типа класса, трейта или объекта-синглтона определяет его имя, сигнатуры типов всех его членов и конструкторов, а также отношения наследования и примешивания, которые в нем объявлены.
- *Синтетический класс* (synthetic class). Не пишется вручную программистом, а генерируется автоматически компилятором.
- *Скрипт* (script). Файл с высокоуровневыми определениями и выражениями, который можно запускать непосредственно с помощью команды `scala`, без

предварительной компиляции. Скрипт должен заканчиваться выражением, а не определением.

- *Слабоструктурированные данные* (semi-structured data). Данные XML являются слабоструктурированными. Они структурированы лучше, чем плоский двоичный или текстовый файл, но при этом уступают полноценным структурам данных в языках программирования.
- *Собственный тип* (self type). Собственный тип трейта — это предполагаемый тип `this` получателя, который будет использоваться внутри трейта. Любой конкретный класс, который примешивается в трейт, должен иметь тип, соответствующий собственному типу трейта. Чаще всего собственные типы используются для разбиения крупных классов на несколько трейтов (см. главу 29).
- *Создание экземпляра* (instantiate). Создать экземпляр класса означает создать на основе класса новый объект. Эта операция происходит только во время выполнения.
- *Сообщение* (message). Акторы взаимодействуют между собой, отправляя друг другу *сообщения*. Отправка сообщения не прерывает то, чем занимается получатель. Последний может подождать, пока не завершится текущее действие, и пока не будут восстановлены его инварианты.
- *Среда выполнения* (runtime). Виртуальная машина Java (Java virtual machine, JVM), в которой выполняется программа на языке Scala. *Среда выполнения* включает в себя как виртуальную машину, соответствующую спецификации JVM, так и библиотеки Java API вместе со стандартными библиотеками Scala API. Словосочетание *«во время выполнения»* (run time) означает, что программа выполняется. Существует также время компиляции.
- *Ссылаться* (refers). Переменная в выполняемой программе на языке Scala всегда *ссылается* на какой-то объект. Даже если этой переменной присвоить `null`, на концептуальном уровне она будет ссылаться на объект `Null`. Во время выполнения объект может быть реализован в виде объекта Java или значения примитивного типа, но Scala позволяет программистам рассуждать о выполнении своего кода на более высоком уровне абстракции. См. также *Ссылка*.
- *Ссылка* (reference). Абстракция для указателя в Java, которая однозначно идентифицирует объект, размещенный в куче JVM. Переменные ссылочных типов хранят ссылки на объекты, поскольку ссылочные типы (экземпляры `AnyRef`) реализованы в виде объектов Java, находящихся в куче JVM. Для сравнения, переменные с типом значения могут хранить ссылку (на тип-обертку), а могут и нет (когда объект представлен примитивным значением). В целом переменные в Scala *ссылаются* на объекты. Термин «ссылаться» более абстрактный, чем «хранить ссылку». Если переменная типа `scala.Int` в настоящий момент представлена в виде примитивного значения `int` из Java, то все равно ссылается на объект `Int`, хотя никаких ссылок при этом не используется.
- *Ссылочная прозрачность* (referential transparency). Свойство функций, которые не зависят от временного контекста и не имеют побочных эффектов. Если взять

конкретные входные данные, то вызов ссылочно прозрачной функции можно заменить ее результатом, не меняя семантику программы.

- *Ссылочный тип* (reference type). Подкласс `AnyRef`. Во время выполнения экземпляры ссылочных типов всегда находятся в куче JVM.
- *Статический тип* (static type). См. *Тип*.
- *Суперкласс* (superclass). Суперклассом класса являются его прямой суперкласс, прямой суперкласс прямого суперкласса и далее вплоть до `Any`.
- *Супертип* (supertype). Тип является супертипом по отношению ко всем своим подтипам.
- *Супертрейт* (supertrait). Супертрейты класса или трейта (если таковые имеются) включают все трейты, напрямую примешанные в класс, в трейт, в любые его суперклассы, а также в супертрейты этих трейтов.
- *Тип* (type). У всех переменных и выражений в программе на языке Scala есть *тип*, известный во время компиляции. Он ограничивает значения, на которые может ссылаться переменная и которые может возвращать выражение во время выполнения. Тип переменной или выражения можно также называть *статическим типом*, если необходимо подчеркнуть его отличие от *типа времени выполнения*. Иными словами, понятие «тип» само по себе является статическим. Тип отличается от класса, поскольку параметризованный класс может формировать много разных типов. Например, `List` — это класс, а не тип. `List[T]` — тип со свободным параметром типа. `List[Int]` и `List[String]` — это тоже типы (их называют *образующими типами*, поскольку у них нет свободных параметров типов). У типа может быть класс или трейт. Например, классом типа `List[Int]` является `List`, а трейтом типа `Set[String]` — `Set`.
- *Тип времени выполнения* (runtime type). Тип объекта во время выполнения. Для сравнения, *статическим* называют тип выражения во время компиляции. Большинство типов времени выполнения представляют собой типы классов без параметров типов. Например, тип времени выполнения `"Hi"` является строкой, а `(x: Int) => x + 1` — `Function1`. Для проверки типов времени выполнения можно использовать `isInstanceOf`.
- *Тип значения* (value type). Любой подкласс `AnyVal`, такой как `Int`, `Double` или `Unit`. Этот термин имеет смысл на уровне исходного кода Scala. Во время выполнения экземпляры типов значений, соответствующие примитивным типам Java, могут быть реализованы в виде значений примитивных типов или экземпляров типов-оберток, таких как `java.lang.Integer`. На протяжении существования экземпляра типа значения среда выполнения может превращать его из примитивного типа в тип-обертку и обратно (то есть упаковывать и распаковывать).
- *Тип, зависящий от пути* (path-dependent type). Тип наподобие `swiss.cow.Food`, где `swiss.cow` — это *путь*, составляющий ссылку на объект. Смысл типа зависит от пути, по которому вы к нему обращаетесь. Например, `swiss.cow.Food` и `fish.Food` — это разные типы.

- *Трейт* (trait). Определяется с помощью ключевого слова `trait` и представляет собой нечто похожее на абстрактный класс, который не может принимать никаких значений. Его можно примешивать в классы или другие трейты с помощью процедуры под названием *композиция примесей*. Трейт, примешанный в класс или другой трейт, называют *примесью*. Трейт может быть параметризован с использованием одного или нескольких типов; в этом случае формирует новый тип. Например, `Set` — трейт, который принимает один параметр типа, в то время как `Set[Int]` — это тип. Можно сказать, что `Set` является трейтом типа `Set[Int]`.
- *Уточняющий тип* (refinement type). Тип, который формируется за счет присваивания значений членам базового типа внутри его фигурных скобок. Эти члены уточняют типы, присутствующие в базовом типе. Например, тип «животное, которое ест траву» можно выразить как `Animal { type SuitableFood = Grass }`.
- *Фильтр* (filter). Инструкция `if` в выражении `for`, за которой идет булево выражение. В `for(i <- 1 to 10; if i % 2 == 0)` фильтром выступает `if i % 2 == 0`. Значение справа от `if` — это *выражение фильтра*.
- *Выражение фильтра* (filter expression). Булево выражение, которое идет за инструкцией `if` в выражении `for`. В `for(i <- 1 to 10; if i % 2 == 0)` фильтрующим выражением выступает `i % 2 == 0`.
- *Функциональное значение* (function value). Функциональный объект, который можно вызывать как любую другую функцию. Класс функционального значения расширяет один из трейтов `FunctionN` (например, `Function0`, `Function1`) из пакета `scala` и обычно выражается в исходном коде с помощью синтаксиса *функциональных литералов*. Функциональное значение вызывается, когда срабатывает его метод `apply`. Функциональное значение, захватывающее свободные переменные, является *замыканием*.
- *Функциональный литерал* (function literal). Функция без имени в исходном коде Scala, описанная с помощью синтаксиса функциональных литералов. Например, `(x: Int, y: Int) => x + y`.
- *Функциональный стиль* (functional style). В этом стиле программирования акцент делается на функциях и вычислении результатов, а порядок выполнения операций играет второстепенную роль. Характерные черты этого стиля — передача функциональных значений в методы с циклами, неизменяемые данные и методы без побочных эффектов. Эта парадигма доминирует в таких языках, как Haskell и Erlang, контрастируя с *императивным стилем*.
- *Функция* (function). Функцию можно *вызвать* со списком аргументов для получения какого-либо результата. У функции есть список параметров, тело и возвращаемый тип. Функции, являющиеся членами класса, трейта или объекта-сигнлтона, называются *методами*. Функции, определенные внутри других функций, называются *локальными*. Функции, возвращающие тип `Unit`, называются *процедурами*. Анонимные функции в исходном коде называются *функциональными литералами*. Во время выполнения для функционального литерала создается объект, называемый *функциональным значением*.

- *Функция без параметров* (parameterless function). Функция, которая не принимает параметров и определяется без использования пустых скобок. При вызове таких функций можно не указывать скобки. Это соответствует *принципу единообразного доступа*, что позволяет поменять `def` на `val`, не модифицируя клиентский код.
- *Функция первого класса* (first-class function). Scala поддерживает *функции первого класса*. Это значит, что вы можете выразить функцию в виде *функционального литерала* (как, например, в `(x: Int) => x + 1`) или объекта, который называют *функциональным значением*.
- *Характеристика for* (for comprehension). Альтернативное название выражения `for`.
- *Хвостовая рекурсия* (tail recursive). Возникает, когда функция вызывает саму себя только в своей последней операции.
- *Целевая типизация* (target typing). Разновидность выведения типов, которая учитывает, какой тип ожидается в итоге. Например, в `nums.filter((x) => x > 0)` компилятор Scala определяет, что `x` — это тип элементов `nums`, поскольку метод `filter` вызывает функцию для каждого элемента `nums`.
- *Частично примененная функция* (partially applied function). Функция, которая используется в выражении с неполным списком своих аргументов. Например, если функция `f` имеет тип `Int => Int => Int`, то `f` и `f(1)` будут *частично примененными функциями*.
- *Член* (member). Любой именованный элемент шаблона класса, трейта или объекта-синглтона. Чтобы обратиться к члену, нужно указать имя его владельца, точку и затем его простое имя. Например, поля и методы верхнего уровня, определенные в классе, являются членами этого класса. Трейт, определенный внутри класса, является его членом. Тип, определенный в классе с помощью ключевого слова `type`, является членом этого класса. Класс является членом пакета, в котором он определен. Тогда как локальную переменную или функцию нельзя считать членом окружающего ее блока.
- *Шаблон* (template). Тело класса, трейта или объекта-одиночки. Определяет сигнатуру типа, поведение и начальное состояние класса, трейта или объекта.
- *Экземпляр* (instance). Экземпляр класса — объект, понятие, которое существует только во время выполнения программы.

Об авторах

Мартин Одерски, создатель языка Scala, — профессор Федеральной политехнической школы Лозанны, Швейцария (EPFL), и основатель Lightbend, Inc. Работает над языками программирования и системами, в частности над темой совмещения объектно-ориентированного и функционального подходов. С 2001 года сосредоточен на проектировании, реализации и улучшении Scala. Ранее внес вклад в разработку Java, выступив соавтором обобщенных типов и создателем текущего эталонного компилятора `javac`. Мартину присвоено звание действительного члена ACM.

Лекс Спун — разработчик программного обеспечения в Semmle, Ltd. Занимался разработкой Scala на протяжении двух лет в ходе постдокторантуры в EPFL. Свою докторскую степень получил в Технологическом институте Джорджии, где темой его исследований был статический анализ динамических языков. Помимо Scala, участвовал в разработке самых разнообразных языков, включая динамический язык Smalltalk, научный язык X10 и логический язык, лежащий в основе Semmle. Лекс проживает в Атланте вместе со своей женой, двумя кошками и чихуахуа.

Билл Веннерс занимает должность президента Artima, Inc. Занимается публикацией материалов на сайте Artima (www.artima.com), предоставляет консультации, учебные материалы, книги и инструменты, относящиеся к Scala. Автор книги *Inside the Java Virtual Machine*, в которой программисты могут познакомиться с архитектурой и внутренним устройством платформы Java. Его популярные статьи в журнале JavaWorld посвящены внутреннему строению Java, объектно-ориентированному проектированию и Jini. Билл занимает позицию представителя сообщества в консультативном совете Scala Center и является ведущим разработчиком и проектировщиком фреймворка тестирования ScalaTest и библиотеки Scalactic, предназначенной для функционального и объектно-ориентированного программирования.

М. Одерски, Л. Спун, Б. Веннерс

Scala. Профессиональное программирование

4-е издание

Перевели с английского *Н. Вильчинский, А. Павлов*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*Н. Искра*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректор	*Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 05.2021. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 30.04.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 58,050. Тираж 500. Заказ 0000.