TypeScript Быстро

Яков Файн
Антон Моисеев

MANNING

TypeScript Quickly

YAKOV FAIN AND ANTON MOISEEV

MANNING
SHELTER ISLAND

Яков Файн
Антон Моисеев

TypeScript Быстро

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2021

ББК 32.988.02-018
УДК 004.738.5
Ф17

Файн Яков, Моисеев Антон

Ф17 TypeScript быстро. — СПб.: Питер, 2021. — 540 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1725-3

«TypeScript быстро» научит вас секретам продуктивной разработки веб- или самостоятельных приложений. Она написана практиками для практиков.

В книге разбираются актуальные для каждого программиста задачи, объясняется синтаксис языка и описывается разработка нескольких приложений, в том числе нетривиальных — так вы сможете понять, как использовать TypeScript с популярными библиотеками и фреймворками. Вы разберетесь с превосходным инструментарием TypeScript и узнаете, как объединить в одном проекте TypeScript и JavaScript. Среди продвинутых тем, рассмотренных авторами, – декораторы, асинхронная обработка и динамические импорты. Прочитав эту книгу, вы поймете, что именно делает TypeScript особенным.

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

ББК 32.988.02-018
УДК 004.738.5

Права на издание получены по соглашению с Manning Publications. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1617295942 англ.
ISBN 978-5-4461-1725-3

© 2020 by Manning Publications Co. All rights reserved.
© Перевод на русский язык ООО Издательство «Питер», 2021
© Издание на русском языке, оформление ООО Издательство «Питер», 2021
© Серия «Для профессионалов», 2021

Краткое содержание

ЧАСТЬ 2
ИСПОЛЬЗОВАНИЕ TYPESCRIPT В БЛОКЧЕЙН-ПРИЛОЖЕНИИ

Оглавление

Часть 2
Использование TypeScript в блокчейн-приложении

Введение

Эта книга посвящена языку программирования TypeScript, который, по данным опроса разработчиков на Stack Overflow, является одним из популярнейших средств программирования (см. https://insights.stackoverflow.com/survey/2019). Как отмечает уважаемый ресурс *ThoughtWork's Technology Radar* (http://mng.bz/Ze5P): «TypeScript — это тщательно проработанный язык, чьи постоянно совершенствующиеся инструменты и поддержка IDE не перестают удивлять нас. Обращаясь к отличному репозиторию определений типов TypeScript, мы используем преимущества всего богатства библиотек JavaScript, получая при этом еще и безопасность типов».

Мы пользуемся TypeScript ежедневно, и он нам очень нравится. Мы действительно ценим его за предоставляемую возможность фокусироваться на основной задаче, а не на опечатках в именах свойств объектов. В программах TypeScript шансы получения ошибок среды выполнения существенно ниже в сравнении с кодом, изначально написанным в JavaScript. Нам также нравится, что IDE предлагают отличную поддержку TypeScript и буквально проводят нас через весь лабиринт API из сторонних библиотек, которые используются в проектах.

TypeScript крут, но поскольку он является языком, компилируемым в JavaScript, необходимо немного поговорить о JS. В мае 1995 года через 10 дней усердной работы, Брендан Эйх (Brendan Eich) создал язык программирования JavaScript. Этот скриптовый язык не нуждался в компиляторе, и предполагалось, что он будет использоваться в браузере Netscape Navigator.

Для развертывания программы JavaScript в браузере не требовались компиляторы. Добавление тега `<script>` с исходным кодом JavaScript (или ссылкой на файл с исходным кодом) давало браузеру команду загружать и считывать код, выполняя его в браузерном движке JavaScript. Люди наслаждались простотой языка, в котором отсутствовала необходимость объявлять типы переменных и использовать какие-либо инструменты. Вы могли просто написать код в простом текстовом редакторе и затем использовать его на веб-странице.

Когда вы начинаете изучать JavaScript, то запустить свою первую программу можете уже через пару минут. Ничего не нужно устанавливать или настраивать, а также нет необходимости компилировать созданную программу, так как JavaScript является интерпретируемым языком.

Кроме этого, JavaScript еще и динамически типизированный язык, что дает дополнительную свободу для разработчиков ПО. Нет необходимости заранее объявлять свойства объектов, поскольку если у объекта во время выполнения не окажется свойства, движок JavaScript создаст его автоматически.

На самом деле в JavaScript не существует способа объявить тип переменной. Движок этого языка угадывает тип на основе присвоенного значения (например, `var x = 123` означает, что `x` является числом). Если позднее этот скрипт будет изменен на `x = "678"`, то тип `x` изменится с числа на строку. Действительно ли вы хотели изменить тип `x` в данном случае или же это была ошибка? Об этом вы узнаете только во время выполнения, так как тут нет компилятора, который мог бы предупредить вас заранее.

JavaScript прощает очень многое, и это нельзя отнести к недостаткам, если кодовая база невелика, а над проектом вы работаете в одиночку. Вероятнее всего, вы будете помнить, что `x` должен быть числом, и подсказка тут не потребуется. Ну и конечно же, если вы будете работать на своего нынешнего работодателя до конца карьеры, то переменную `x` вы никогда не забудете.

Спустя годы JavaScript стал суперпопулярным и фактически стандартом языком веб-программирования. Однако 20 лет назад разработчики использовали его для отображения веб-страниц с небольшой долей интерактивного контента; сегодня же мы разрабатываем комплексные веб-приложения, содержащие тысячи строк кода, написанного командой разработчиков. Далеко не каждый в вашей команде будет помнить, что переменная `x` должна быть числом. Для минимизации числа ошибок среды выполнения JavaScript-разработчики создают юнит-тесты и производят код-ревью.

Для повышения продуктивности разработчики прибегают к помощи IDE, предлагающей возможности автоподстановки, легкий рефакторинг и т. д. Но как может IDE помочь вам с рефакторингом, если сам язык дает полную свободу в добавлении свойств объектам и изменении их типов буквально на ходу?

Веб-разработчики нуждались в лучшем языке, но замещение JavaScript аналогом, поддерживаемым различными браузерами, выглядело нереальным. Вместо этого были разработаны новые языки, компилируемые в JavaScript. Они были более совместимы с инструментами, но программу перед развертыванием попрежнему нужно было конвертировать в JavaScript, чтобы ее мог поддержать любой браузер. TypeScript является одним из таких языков, и прочитав эту книгу, вы поймете, что именно делает его особенным.

Благодарности

Яков благодарит своего лучшего друга Сэмми за теплую и уютную атмосферу в процессе работы над книгой. К сожалению, Сэмми не может говорить, но, как и любая другая собака, любит всех членов своей семьи больше, чем они сами себя.

Антон благодарит авторов и участников открытых проектов, которые упомянуты в этой книге. Ее создание оказалось бы невозможным, если бы эти люди не посвящали регулярно множество часов своим проектам, а также постоянной работе по развитию и поддержанию сообществ. Антон также благодарен семье за терпение, которое они проявили на время его работы над книгой.

Отдельную благодарность выражаем всем рецензентам книги, предоставившим ценную обратную связь: Ахмаду Субахи (Ahmad Subahi), Александросу Далласу (Alexandros Dallas), Брайану Дэйли (Brian Daley), Кэмерону Пресли (Cameron Presley), Кэмерону Синге (Cameron Singe), Денизу Вехби (Deniz Vehbi), Флорис Бушо (Floris Bouchot), Джорджу Онофрею (George Onofrei), Джорджу Томасу (George Thomas), Джеральду Джеймсу Стралко (Gerald James Stralko), Гаю Лэнгстону (Guy Langston), Джеффу Смиту (Jeff Smith), Джастину Кану (Justin Kahn), Кенту Р. Спиллнеру (Kent R. Spillner), Кевину Орру (Kevin Orr), Лукасу Пардью (Lucas Pardue), Марко Летичу (Marco Letic), Маттео Баттиста (Matteo Battista), Полу Брауну (Paul Brown), Полине Кесельман (Polina Keselman), Ричарду Таттлу (Richard Tuttle), Шридхари Шридхарану (Sridhari Sridharan), Тамаре Форза (Tamara Forza) и Томасу Оверби Хансену (Thomas Overby Hansen).

О книге

ДЛЯ КОГО ЭТА КНИГА

Эта книга написана для инженеров ПО, которые хотят повысить продуктивность разработки веб- или самостоятельных приложений. Мы, ее авторы, являемся практиками и книгу писали для таких же практиков. В ней не только объясняется синтаксис языка на простых примерах, но также описывается разработка нескольких приложений — так вы можете понять, как использовать TypeScript с популярными библиотеками и фреймворками.

В процессе создания этой книги мы проводили мастер-классы, используя приведенные в ней образцы кода. Это позволило получить раннюю обратную связь по содержанию книги. Надеемся, что вы получите удовольствие от процесса изучения TypeScript.

Этот материал, в свою очередь, предполагает, что у читателей уже есть практические познания в HTML, CSS и JavaScript, использующем последние нововведения из спецификации ECMAScript. Если вам знаком только синтаксис ECMAScript 5, рекомендуем просмотреть приложение, чтобы лучше понимать примеры кода, используемые в книге. Приложение выполняет роль введения в современный JavaScript.

СТРУКТУРА КНИГИ

Эта книга состоит из двух частей. В части 1 мы рассматриваем различные элементы синтаксиса TypeScript, приводя для наглядности небольшие образцы кода. В части 2 мы используем TypeScript в нескольких версиях блокчейн-приложения. Если ваша цель — как можно быстрее освоить синтаксис TypeScript и сопутствующие инструменты, тогда части 1 будет достаточно.

Глава 1 поможет начать разработку с помощью TypeScript. Вы скомпилируете и запустите самые простые программы, чтобы понять суть рабочего процесса — от

написания программ в TypeScript и до их компиляции в выполняемый JavaScript. Мы также рассмотрим преимущества программирования в TypeScript в сравнении с JavaScript и представим вам редактор Visual Studio Code.

Глава 2 поясняет, как объявлять переменные и функции с типами. Вы научитесь объявлять псевдонимы типов с помощью ключевого слова `type`, а также узнаете, как объявлять пользовательские типы с классами и интерфейсами. Это поможет понять разницу между номинальными и структурными системами типов.

Глава 3 рассказывает, как работает наследование классов и когда использовать абстрактные классы. Вы увидите, как с помощью интерфейсов в TypeScript можно принудить класс иметь методы с известными сигнатурами, не беспокоясь при этом о деталях реализации. Вы также узнаете, что означает «программирование через интерфейсы».

Глава 4 посвящена перечислениям и обобщенным типам. Она рассматривает преимущества использования перечислений, синтаксис для численных и строчных перечислений, назначение обобщенных типов, а также показывает, как прописывать классы, интерфейсы и поддерживающие их функции.

Глава 5 описывает декораторы, а также отображенные и условные типы. Все это относится к продвинутым типам TypeScript, поэтому для понимания этой главы следует ознакомиться с синтаксисом обобщенных типов.

Глава 6 посвящена сопутствующим инструментам. В ней мы рассказываем об использовании карт кода и TSLinter (несмотря на то, что TSLinter считается устаревшим, многие разработчики по-прежнему им пользуются). Затем мы покажем вам, как компилировать и обвязывать (bundle) приложения TypeScript с помощью Webpack. Вы также узнаете, как и зачем компилировать TypeScript с помощью Babel.

Глава 7 научит вас использовать библиотеки JavaScript в вашем TypeScript-приложении. Мы начнем с объяснения роли файлов определений типов, а затем представим небольшое приложение, использующее библиотеку JavaScript в приложении TypeScript. В завершение мы пройдем через весь процесс постепенного перевода имеющегося проекта JavaScript в TypeScript.

В части 2 мы используем TypeScript в блокчейн-приложении. Здесь вы можете подумать: «Ни одна из компаний, где я работал, не использует блокчейн, так зачем мне его изучать, если моя цель — освоение TypeScript?» Мы не хотели, чтобы наш образец приложения был очередным стандартным примером, поэтому озадачились поиском горячей технологии, в которой можно было бы применить различные элементы TypeScript, а также техники, представленные в части 1. Знакомство с тем, как TypeScript может быть использован в нетривиальном приложении, придаст содержанию больше практичности, даже если вы не собираетесь использовать блокчейн в ближайшем будущем.

В этой части книги вы разработаете несколько блокчейн-приложений: самостоятельное приложение, браузерное приложение, а также приложения на Angular, React.js и Vue.js. При этом можете выбрать к прочтению только интересующие вас главы, но обязательно прочитайте главы 8 и 10, где объясняются основные принципы.

Глава 8 представляет принципы блокчейн-приложений. В ней вы узнаете, для чего нужно хеширование функций, что означает блокчейн-майнинг и почему для добавления нового блока в блокчейн требуется подтверждение работы. После знакомства с основами блокчейна мы представим проект и поясним код, лежащий в основе простого блокчейн-приложения. Большинство глав в части 2 имеют рабочие проекты с подробным пояснением того, как они были написаны и как запускаются.

В главе 9 описывается создание веб-клиента для блокчейна. Это приложение не будет использовать никакие фреймворки; в нем мы задействуем только HTML, CSS и TypeScript. Мы также создадим небольшую библиотеку для генерации хеша, которую можно будет использовать как в веб-, так и в самостоятельных клиентах. Помимо этого, вы увидите, как производить отладку кода TypeScript в своем браузере.

Глава 10 рассматривает код блокчейн-приложения, использующего сервер обмена сообщениями для связи между членами блокчейна. Мы создадим Node.js и WebSocket сервер в TypeScript и покажем вам, как для достижения консенсуса блокчейн использует правило длиннейшей цепочки. Вы найдете практические примеры использования TypeScript интерфейсов, абстрактных классов, квалификаторов доступа, перечислений и обобщенных типов.

В главе 11 дается краткий обзор разработки веб-приложений в Angular при помощи TypeScript, а глава 12 рассматривает код блокчейн-веб-клиента, разработанного с помощью этого фреймворка.

Глава 13 дает краткое введение в разработку веб-приложений в React.js с помощью TypeScript, а глава 14 рассматривает код блокчейн-веб-клиента, разработанного с React.

Глава 15 аналогичным образом представляет разработку веб-приложений в Vue.js с помощью TypeScript, а глава 16 рассматривает блокчейн-веб-клиент, разработанный с использованием Vue.

О КОДЕ

Эта книга содержит много примеров исходного кода как в нумерованных листингах, так и в основном тексте. В обоих случаях исходный код имеет формат

`моноширинного шрифта` для выделения на фоне остального текста. Иногда код выделен **жирным**, обозначая те его части, которые изменились с момента их предыдущего появления в главе. Например, когда в существующую строчку кода добавляется новая функциональность.

Во многих случаях исходный код был переформатирован; мы добавили разрывы между строк и переработали отступы для соответствия ширине страницы. Но в каких-то случаях этого было недостаточно, поэтому некоторые листинги включают маркеры переноса строки (➥). Помимо этого, комментарии исходного кода были удалены из листингов, если его описание уже содержалось в самом тексте. Аннотации кода сопровождают многие листинги, подчеркивая важные принципы.

Часть 1 посвящена синтаксису языка, и большинство приведенных в ней образцов кода опубликованы онлайн в «песочнице» TypeScript — интерактивном инструменте, быстро проверяющем синтаксис фрагментов TypeScript кода и компилирующем его в JavaScript. Ссылки на такие фрагменты кода предоставлены в книге по мере необходимости.

Вторая часть состоит из нескольких проектов, где TypeScript используется для разработки приложений с помощью популярных библиотек и фреймворков (вроде Angular, React.js и Vue.js). Исходный код этих приложений размещен на GitHub по адресу https://github.com/yfain/getts.

Мы тщательно протестировали каждое приложение, описанное в книге, но нарушающие их работу изменения могут быть вызваны более новыми релизами TypeScript или библиотек. Если при попытке запустить один из проектов возникает ошибка, пожалуйста, создайте запрос в репозитории этой книги на GitHub.

ОБ АВТОРАХ

Яков Файн является сооснователем двух IT-компаний: Farata Systems и SuranceBay. Он автор и соавтор таких книг, как *Java Programming: 24-Hour Trainer*, *Angular Development with TypeScript*[1], *Java Programming for Kids* и др. Являясь чемпионом Java, провел множество классов и семинаров, посвященных веб- и Java-технологиям. Помимо этого, также был докладчиком на международных конференциях. Файн опубликовал более тысячи статей в своем блоге на yakovfain.com. В твиттере и инстаграме его можно найти по адресу @yfain. Он также публикует видео на YouTube.

[1] *Файн Я., Моисеев А.* Angular и TypeScript. Сайтостроение для профессионалов. — СПб.: Питер, 2018. — 464 с.: ил.

Антон Моисеев является ведущим разработчиком в SuranceBay. Провел за разработкой корпоративных приложений более десяти лет, работая с Java и .NET. Имеет обширный опыт и фокусируется на развитии веб-технологий, реализующих лучшие методики слаженной работы фронтенда и бэкенда. Некоторое время проводил обучение по фреймворкам AngularJS и Angular. Периодически Антон делает посты в блоге antonmoissev.com. В твиттере вы можете найти его по адресу @antonmoiseev.

ОБ ОБЛОЖКЕ

На обложке «TypeScript быстро» изображена «Буржуазная Флоренция». Эта иллюстрация позаимствована из коллекции костюмов различных стран, созданной Жаком Грассе де Сен-Совер (Jacques Grasset de Saint-Sauveur) (1757–1810) под названием «Costumes civils actuels de tuos les peoples connus», опубликованной во Франции в 1788 году. Каждая иллюстрация этой коллекции была нарисована и раскрашена от руки. Богатое разнообразие коллекции Грассе де Сен-Совер наглядно напоминает нам, насколько культурно разрознены были города и регионы всего мира еще 200 лет назад. Будучи изолированными друг от друга, люди говорили на различных диалектах и языках. На сельских же улицах было достаточно взглянуть на одеяние жителя, чтобы распознать, откуда он родом, кем работает и в каком жизненном положении находится.

Манера одеваться с тех пор сильно изменилась, и некогда богатое региональное разнообразие практически угасло. Сегодня стало сложно различить жителей разных континентов, не говоря уже о разных странах, регионах или городах. Можно предположить, что мы променяли культурное разнообразие на более разностороннюю личную жизнь, но при этом точно на более богатую и быстро меняющуюся жизнь мира технологий.

Во времена, когда стало тяжело отличить одну компьютерную книгу от другой, Manning выделяется изобретательностью и инициативой в компьютерном бизнесе, создавая обложки на основе богатого разнообразия жизни регионов двухвековой давности, которое было повторно явлено миру благодаря работе Грассе де Сен-Совер.

Часть 1

Основы синтаксиса TypeScript

Начнем часть 1 с объяснения преимуществ использования TypeScript в сравнении с JavaScript. После этого рассмотрим различные элементы синтаксиса TypeScript на примерах небольших фрагментов кода. Вы увидите, как использовать встроенные и объявлять пользовательские типы. Помимо этого, мы научимся применять классы, интерфейсы, а также обобщения, перечисления, декораторы, отображенные и условные типы. Вы познакомитесь с инструментами, которые используются разработчиками в TypeScript (например, компиляторы, линтеры, отладчики и бандлеры). В завершение мы покажем, как использовать в одном приложении код TypeScript совместно с JavaScript.

Для тех из вас, кто любит смотреть видео, Яков Файн опубликовал серию роликов на английском языке (см. http://mng.bz/m4M8), которые иллюстрируют содержимое части 1 этой книги. Если вы хотите как можно скорее освоить именно синтаксис TypeScript и его инструменты, часть 1 предоставит все, что для этого нужно.

1 Знакомство с TypeScript

В этой главе:

- ✓ Преимущества программирования в TypeScript в сравнении с JavaScript.
- ✓ Компиляция кода TypeScript в JavaScript.
- ✓ Работа с редактором Visual Studio Code.

Цель этой главы — помочь вам начать использовать TypeScript для разработки. Начнем с выражения почтения самому JavaScript, а затем обоснуем, почему вам стоит программировать именно в TypeScript. В завершение главы скомпилируем и запустим очень простую программу, чтобы вы могли проследить рабочий процесс от написания программы в TypeScript до ее компиляции в исполняемый JavaScript вариант. Если вы уже опытный JS-разработчик, то нужна весомая причина для перехода на TypeScript, который в любом случае необходимо компилировать в JavaScript перед развертыванием. Если вы бэкенд-разработчик, планирующий изучать фронтенд экосистему, то вам также нужна причина для изучения языка, отличного от JavaScript. Что ж, давайте рассмотрим эти причины.

1.1. ЗАЧЕМ ПРОГРАММИРОВАТЬ В TYPESCRIPT

TypeScript — это язык, компилируемый в JavaScript. Он был выпущен корпорацией Microsoft в 2012 году как открытый проект. Программа, написанная

в TypeScript, сперва должна быть *транспилирована* в JavaScript, и только потом ее можно выполнить в браузере или отдельном движке JavaScript.

Разница между транспиляцией и компиляцией в том, что последняя преобразовывает исходный код программы в машинный байт-код, в то время как транспиляция конвертирует программу из одного языка в другой (например, из TypeScript в JavaScript). Однако в сообществе TypeScript все же более распространен термин *компиляция*, поэтому его мы и будем использовать в этой книге для описания процесса преобразования TypeScript-кода в JavaScript.

Вам может стать интересно, зачем все эти сложности с написанием программы в TypeScript и его последующей компиляцией в JavaScript, если можно изначально написать эту программу в JavaScript. Чтоб ответить на этот вопрос, взглянем на TypeScript с высокоуровневой позиции.

TypeScript — это надмножество JavaScript, поэтому можно взять любой файл JavaScript вроде myProgram.js, изменить его расширение с .js на .ts, и получившийся файл myProgram.ts, скорее всего, станет рабочей программой TypeScript. Мы говорим «скорее всего», потому что оригинальный код JavaScript может иметь скрытые баги, связанные с типами (он может динамически изменять типы свойств объектов или добавлять новые после объявления объекта), а также другие проблемы, которые вскроются только после компиляции.

> **СОВЕТ** В разделе 7.3 мы приведем несколько советов по миграции кода JavaScript в TypeScript.

В общем смысле слово *надмножество* подразумевает, что это надмножество содержит все, что есть в самом множестве, плюс что-то еще. Рисунок 1.1 представляет TypeScript как надмножество ECMAScript, являющегося спецификацией для всех версий JavaScript. ES.Next представляет самые последние дополнения в ECMAScript, которые еще дорабатываются.

В дополнение к набору JavaScript, TypeScript также поддерживает *статическую типизацию*, в то время как JavaScript поддерживает только *динамическую*. Здесь слово «типизация» относится к присвоению типов переменным программы.

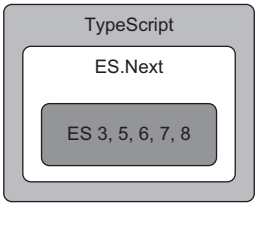


Рис. 1.1. TypeScript как надмножество

В языках программирования со статической типизацией тип переменной должен быть присвоен прежде, чем ее можно будет использовать. В TypeScript можно объявить переменную конкретного типа, и любая попытка присвоить ей значение другого типа приведет к появлению ошибки.

В JavaScript ситуация обстоит иначе, так как он ничего не знает о типах переменных вашей программы до начала ее выполнения. И даже в запущенной программе вы можете изменить тип переменной, просто присвоив ей значение другого типа. В TypeScript же, если вы объявляете переменную как `string`, то попытка присвоить ей численное значение приведет к *ошибке при компиляции*.

```
let customerId: string;
customerId = 123; // ошибка при компиляции
```

JavaScript определяет тип переменной в процессе выполнения, и тип может динамически изменяться, как это показано в следующем примере:

```
let customerId = "A15BN"; // OK, customerId является строкой.
customerId = 123; // OK, с этого момента она является числом.
```

Теперь давайте рассмотрим JavaScript-функцию, которая применяет к цене скидку. У нее есть два аргумента, которые должны быть числами.

```
function getFinalPrice(price, discount) {
    return price - price / discount;
}
```

Откуда вы знаете, что аргументы должны быть числами? Во-первых, вы написали эту функцию недавно и, обладая феноменальной памятью, просто помните типы всех ее аргументов. Во-вторых, вы использовали для аргументов описательные имена, которые подсказывают их типы. В-третьих, вы можете догадаться о типах, прочитав код самой функции.

Это достаточно простая функция, но давайте представим, что кто-то вызвал ее, сообщив скидку в виде строки. В этом случае функция в среде выполнения выдаст ошибку `NaN`.

```
console.log(getFinalPrice( 100, "10%")); // выводит NaN
```

Это пример ошибки среды выполнения, вызванной неправильным использованием функции. В TypeScript вы могли бы присвоить аргументам функции типы и избежать тем самым появления подобной ошибки. Если кто-либо пытается вызвать функцию с неверным типом аргумента, то эта ошибка будет перехвачена сразу во время набора. Рассмотрим это на практике.

Официальная страница TypeScript (www.typescriptlang.org) предлагает документацию к этому языку и песочницу, где вы можете вводить фрагменты кода TypeScript, которые будут тут же скомпилированы в JavaScript.

По ссылке http://mng.bz/Q0Mm вы увидите наш фрагмент кода в упомянутой песочнице TypeScript. В коде есть красное волнистое подчеркивание под `"10%"`. Если вы наведете указатель мыши на код с ошибкой, то увидите подсказку (рис. 1.2).

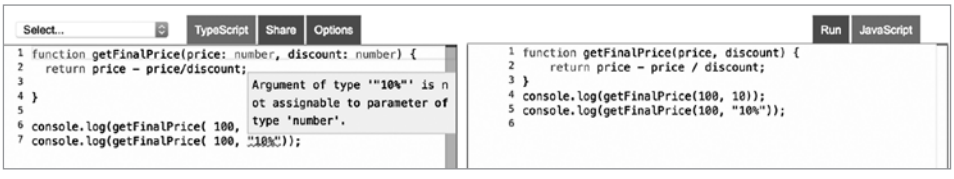


Рис. 1.2. Песочница TypeScript

Эта ошибка перехватывается статическим анализатором кода TypeScript во время вывода, еще до того, как будет произведена компиляция кода компилятором TypeScript (tsc). Более того, если вы укажете типы переменных, ваш редактор или IDE предложит автоподстановку имен аргументов и типов для функции `getFinalPrice()`.

Разве не прекрасно, что ошибки перехватываются до выполнения программы? Мы убеждены, что да. Многие разработчики, имеющие опыт работы в Java, C++ и C#, воспринимают такое обнаружение ошибок как само собой разумеющееся, и именно эта особенность является одной из основных причин выбора TypeScript.

> **ПРИМЕЧАНИЕ** Есть два вида программных ошибок: те, о которых сообщают вспомогательные инструменты в процессе набора кода, и те, о которых сообщают пользователи вашей программы. Программирование в TypeScript существенно снижает количество ошибок второго типа.

> **СОВЕТ** На сайте TypeScript (www.typescriptlang.org) есть раздел «Documentation and tutorials» (Документация и обучающий материал). Там вы найдете полезные советы по настройке TypeScript в различных средах вроде ASP.NET, React и пр.

Некоторые матерые JS-разработчики говорят, что TypeScript только замедляет рабочий процесс, требуя объявлений типов, и что в JavaScript их продуктивность выше. Но при этом стоит помнить, что типы в TypeScript используются по желанию, то есть вы можете продолжить писать на JavaScript, но с участием tsc в рабочем процессе. Зачем? Потому, что вы сможете использовать новейший синтаксис ECMAScript (вроде `async` и `await`), а затем транспилировать ваш JavaScript код в ES5, чтобы он мог быть запущен в более старых браузерах.

Но многие веб-разработчики, не являющиеся JavaScript-ниндзя, смогут по достоинству оценить помощь, предлагаемую TypeScript. По правде говоря, все

строго типизированные языки предоставляют лучшую поддержку инструментов, повышая тем самым производительность (даже для ниндзя). Упомянув это, мы бы хотели отметить, что TypeScript дает вам преимущества статически типизированных языков, где и когда это нужно, не препятствуя при этом использованию старых добрых динамических объектов.

Более сотни языков программирования компилируются в JavaScript (согласно списку: http://mng.bz.MO42). TypeScript же на их фоне выделяется тем, что его создатели следуют стандартам ECMAScript и реализуют намечающиеся функции JavaScript намного быстрее, чем поставщики браузеров.

Вы можете найти текущие предложения по введению новых возможностей ECMAScript на GitHub: https://github.com/tc39/proposals. Предложение должно пройти несколько стадий, чтобы быть включенным в итоговую версию следующей спецификации ECMAScript. Если предложение доходит до третьей стадии, то, скорее всего, будет включено в последнюю версию TypeScript.

Летом 2017 года в спецификацию ECMAScript ES2017 (ES8) были введены ключевые слова `async` и `await`. Ведущим браузерам при этом потребовалось более года, чтобы внедрить их поддержку, TypeScript же начал поддерживать их с ноября 2015 года. TypeScript-разработчики получили возможность использовать эти ключевые слова на три года раньше тех, кто ожидал появления их поддержки в браузерах. Самое лучшее в этом то, что вы можете использовать будущий синтаксис JavaScript в сегодняшнем коде TypeScript и транспилировать его в более старые версии синтаксиса JavaScript (вроде ES5), которые поддерживаются всеми браузерами.

С учетом сказанного хотелось бы сделать четкое разграничение между синтаксисом, описанным в последних спецификациях ECMAScript, и синтаксисом, присущим исключительно TypeScript. Рекомендуем для начала ознакомиться с приложением, чтобы знать, где заканчивается ECMAScript и начинается TypeScript.

Несмотря на то что движки JavaScript отлично справляются с угадыванием типов переменных по их значениям, инструменты разработки не могут оказать полноценную помощь, не зная этих типов. В средних и крупных приложениях этот недочет JavaScript снижает производительность разработчиков.

TypeScript следует новейшим спецификациям ECMAScript и добавляет к ним типы, интерфейсы, декораторы, переменные членов классов (поля), обобщенные типы, перечисления, ключевые слова `public`, `protected`, `private` и другие возможности. Изучите дорожную карту TypeScript (https://github.com/Microsoft/TypeScript/wiki/Roadmap), чтобы посмотреть доступные и ожидаемые возможности. И еще одно: JavaScript-код, сгенерированный из TypeScript, легко читается и выглядит как набранный вручную.

ПЯТЬ ФАКТОВ О TYPESCRIPT

- Главным разработчиком TypeScript является Андерс Хейлсберг, который также спроектировал Turbo Pascal, Delphi и был ведущим проектировщиком C# в Microsoft.
- В конце 2014 года корпорация Google обратилась к Microsoft с просьбой ввести в TypeScript декораторы, чтобы этот язык мог использоваться для разработки на фреймворке Angular 2. В Microsoft согласились, и это чрезвычайно повысило популярность TypeScript, учитывая, что сотни тысяч разработчиков используют Angular.
- По состоянию на декабрь 2019 года tsc имел несколько миллионов скачиваний в неделю с сайта npmjs.org, а это не единственный репозиторий TypeScript. Текущую статистику можете просмотреть по этому адресу: www.npmjs.com/package/typescript.
- Согласно Redmonk, авторитетной компании по аналитике ПО, TypeScript занял 12-е место в рейтинге языков программирования в январе 2019 года (рейтинг можно посмотреть здесь: http://mng.bz/4eow).
- Согласно опросу разработчиков, проведенному Stack Overflow в 2019 году, TypeScript занимает третье место в списке любимых языков (https://insights.stackoverflow.com/survey/2019).

Далее мы представим процесс настройки использования tsc на вашем компьютере.

1.2. ТИПИЧНЫЕ РАБОЧИЕ ПРОЦЕССЫ TYPESCRIPT

Давайте познакомимся с рабочим процессом TypeScript, начиная с написания кода и заканчивая развертыванием приложения. На рис. 1.3. представлен этот процесс. Весь исходный код написан в TypeScript.

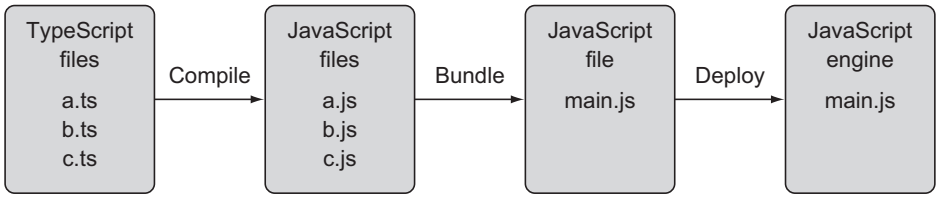


Рис. 1.3. Развертывание приложения, написанного в TypeScript

Как вы можете видеть, проект состоит из трех файлов TypeScript: a.ts, b.ts и c.ts. Эти файлы должны быть скомпилированы в JavaScript компилятором TypeScript (tsc), который сгенерирует три новых файла: a.js, b.js и c.js. Чуть позже мы покажем, как настроить компилятор на генерацию JavaScript конкретных версий.

Сейчас некоторые JavaScript-разработчики скажут: «TypeScript вынуждает меня вводить дополнительный шаг компиляции между написанием и выполнением кода». Но вопрос в том, действительно ли вы хотите придерживаться версии ES5 JavaScript, игнорируя весь новейший синтаксис, добавленный в ES6, 7, 8 вплоть до ES.Next? Если нет, то в любом случае будете использовать шаг компиляции в своем рабочем процессе, так как потребуется компилировать исходный код, написанный в более новой версии JavaScript, в широко поддерживаемый синтаксис ES5.

На рис. 1.3 изображено всего три файла, но в реальных проектах их число может достигать сотен и даже тысяч. Разработчики не хотят развертывать такое множество файлов на веб-сервере или в самостоятельном JavaScript-приложении, поэтому обычно мы связываем эти файлы (конкатенируем) вместе.

JavaScript-разработчики используют различные бандлеры вроде Webpack или Rollup, которые не только конкатенируют несколько файлов JavaScript, но также могут оптимизировать код и удалить его неиспользуемые части (выполняя перетряхивание дерева). Если ваше приложение состоит из нескольких модулей, каждый из них может быть развернут как отдельная связка.

На рис. 1.3 изображена всего одна развернутая связка — main.js. Если бы это было веб-приложение, то в нем бы был HTML-файл с тегом `<script src='main.js'>`. Если бы приложение запускалось в самостоятельном JavaScript-движке вроде Node.js, то вы могли бы начать со следующей команды (предполагается, что Node.js установлен):

```
node main.js
```

Экосистема JavaScript включает тысячи библиотек, которые не будут переписываться в TypeScript. Но при этом хорошо то, что приложение не обязательно должно быть создано только с помощью TypeScript и может использовать любые из доступных библиотек JavaScript.

Если вы просто добавите такую библиотеку в свое приложение, то tsc не предложит помощь в виде автоподстановки или сообщений об ошибках при использовании ее API. Поэтому существуют специальные файлы определений типов с расширением .d.ts (подробнее в главе 6), наличие которых активирует вышеописанную контекстную автоподстановку и сообщения об ошибках.

Рисунок 1.4 показывает пример рабочего процесса для приложения, использующего популярную JavaScript-библиотеку lodash.

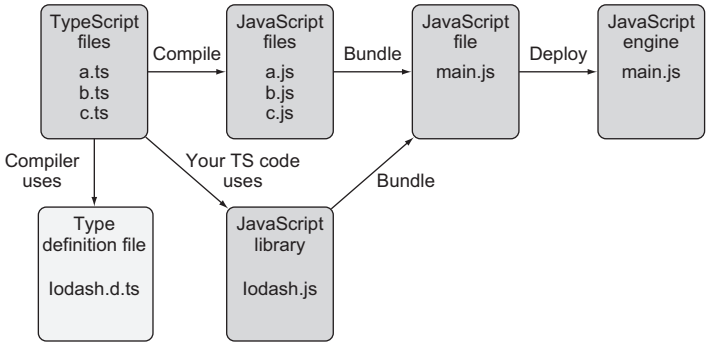


Рис. 1.4. Развертывание приложения, написанного совместно в TypeScript и JavaScript

Эта диаграмма включает файл определений типов lodash.d.ts, который используется tsc в процессе разработки. Он также включает актуальную библиотеку lodash.js, которая будет связана с остальной частью приложения во время его развертывания. Термин «связка» относится к процессу объединения нескольких файлов сценариев (скриптов) в один.

1.3. ИСПОЛЬЗОВАНИЕ КОМПИЛЯТОРА TYPESCRIPT

Пришло время изучить компиляцию простой программы TypeScript в ее JavaScript-версию. Компилятор (tsc) может быть связан с выбранной вами IDE или установлен как ее плагин. Мы же предпочитаем устанавливать его независимо от IDE, используя пакетный менеджер npm, поставляемый с Node.js. Node.js (или просто Node), — это не просто фреймворк или библиотека, но также и среда выполнения JavaScript. Мы используем ее для выполнения различных утилит вроде npm или запуска JavaScript-кода без браузера.

Для начала вам потребуется скачать и установить текущую версию Node.js с https://nodejs.org. Он установит node и npm.

С помощью npm вы можете устанавливать ПО либо локально в директорию вашего проекта, либо глобально, чтобы можно было использовать его в разных проектах. Мы будем использовать npm, чтобы установить tsc и другие пакеты из репозитория, размещенного на www.npmjs.com и содержащего более полумиллиона различных пакетов.

Вы можете установить tsc глобально (используя опцию `-g`), выполнив следующую команду в окне терминала:

```
npm install -g typescript
```

> **ПРИМЕЧАНИЕ** Для простоты в первой части книги мы будем использовать tsc, установленный глобально. Тем не менее в реальных проектах мы предпочитаем устанавливать tsc локально в директорию проекта посредством его добавления в раздел `devDependencies` файла package.json. Вы увидите, как это делается, в главе 8, когда мы начнем работать над примером блокчейн-проекта.

В примерах кода этой книги мы использовали TypeScript 3-й версии или новее. Чтобы проверить версию вашего tsc, выполните следующую команду в терминале:

```
tsc -v
```

Теперь давайте посмотрим, как скомпилировать простую программу из TypeScript в JavaScript. Выберите любую директорию и создайте в ней новый файл main.ts со следующим содержимым.

Листинг 1.1. Файл main.ts

```
function getFinalPrice(price: number, discount: number) {
    return price - price/discount;
}

console.log(getFinalPrice(100, 10));
console.log(getFinalPrice(100, "10%"));
```

Аргументы функций имеют типы

Верный вызов функции

Ошибочный вызов функции

Следующая команда скомпилирует main.ts в main.js:

```
tsc main
```

Она вызовет появление сообщения об ошибке «аргумент типа `'10%'` не может быть присвоен параметру типа `'number'`», но при этом все равно сгенерирует файл main.js со следующим содержимым:

Листинг 1.2. Получившийся файл main.js

```
function getFinalPrice(price, discount) {
    return price - price/discount;
}

console.log(getFinalPrice(100, 10));
console.log(getFinalPrice(100, "10%"));
```

Аргументы без типов

Верный вызов функции

Неверный вызов функции, но ошибка будет показана только при выполнении

Здесь вы можете спросить: «Какой смысл создания JavaScript-файла при наличии ошибки компиляции?» Что ж, с точки зрения Javascript содержимое файла main.js допустимо. Но в реальных TypeScript-проектах мы не хотим допускать генерацию кода для файлов с ошибками.

Для этого есть решение. Компилятор tsc предлагает десятки опций компиляции, описанных в документации TypeScript (http://mng.bz/rf14), одной из которых

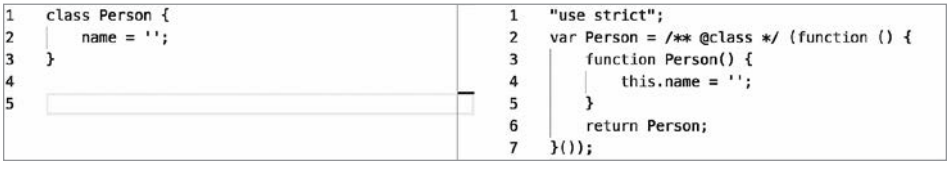
является `noEmitOnError`. Удалите файл main.js и попробуйте скомпилировать main.ts следующим образом:

```
tsc main --noEmitOnError true
```

В этот раз файл main.js сгенерирован не будет, пока мы не устраним ошибку в main.ts.

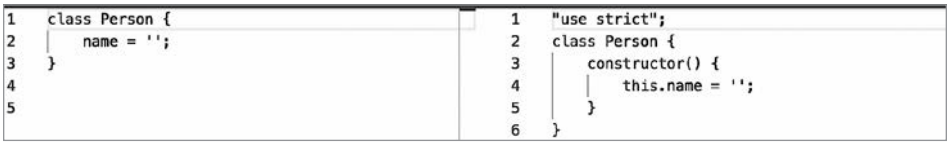
СРЕДА REPL ДЛЯ TYPESCRIPT

REPL расшифровывается как Real-Evaluate-Print-Loop (цикл чтение-вычисление-печать) и относится к простой интерактивной языковой оболочке, позволяющей быстро выполнять фрагменты кода. Интерактивная среда TS по адресу www.typescriptlang.org/play представляет собой вариант REPL, в котором вы можете писать, компилировать и выполнять фрагменты кода, находясь в браузере. Следующий пример показывает, как можно использовать песочницу для компиляции простого TS класса в версию ES5 JavaScript.



Транспиляция TypeScript в ES5

На следующем рисунке представлена компиляция того же кода в версию ES6.



Транспиляция TypeScript в ES6

В песочнице есть меню Options, где можно выбирать опции компилятора. В частности, можно выбрать целевую версию для компиляции, например ES2018 или ES5.

Если хотите запускать фрагменты кода из командной строки без участия браузера, установите TypeScript Node REPL, доступный по ссылке https://github.com/TypeStrong/ts-node.

СОВЕТ Включение опции `noEmitOnError` означает, что ранее сгенерированный файл не будет замещен, пока все ошибки в файлах TypeScript не будут устранены.

Опция компилятора `--t` позволяет устанавливать синтаксис целевого JavaScript-файла. Например, вы можете использовать один и тот же исходный файл и генерировать его в одноранговую версию JavaScript с синтаксисом ES5, ES6 или более новым. Так можно скомпилировать код в синтаксис ES5:

```
tsc --t ES5 main
```

Tsc позволяет вам предварительно настроить процесс компиляции (устанавливая исходную директорию и директорию назначения, целевую версию и т. д.). Если ваш файл tsconfig.json расположен в директории проекта, вы можете просто ввести tsc в командной строке, и компилятор считает из него все настройки. Пример файла tsconfig.json показан ниже.

Листинг 1.3. Файл tsconfig.json

```
{
    "compilerOptions": {
        "baseUrl": "src",
        "outDir": "./dist",
        "noEmitOnError": true,
        "target": "es5"
    }
}
```

Транспилировать файлы .ts, расположенные в директории src

Сохранять сгенерированные файлы .js в директории dist

Если в каких-либо файлах есть ошибки компиляции, не генерировать файлы JavaScript

Транспилировать файлы TypeScript в синтаксис ES5

СОВЕТ Опция компилятора target также используется для проверки синтаксиса. Например, если вы укажете целевую версию для компиляции как es3, TypeScript будет ругаться на методы-геттеры в вашем коде. Так произойдет потому, что он не знает, как компилировать геттеры в версию ECMAScript 3.

Давайте посмотрим, сможете ли вы проделать все описанное, следуя инструкциям:

1. Создайте файл tsconfig.json в каталоге, где расположен файл main.ts, и добавьте в него следующее:

```
{
    "compilerOptions": {
        "noEmitOnError": true,
        "target": "es5",
        "watch": true
    }
}
```

Обратите внимание на последнюю опцию — `watch`. Благодаря ей компилятор будет следить за вашими файлами TypeScript, и если они изменятся, tsc их перекомпилирует.

2. В терминале проследуйте до директории, где расположен файл tsconfig.json, и выполните следующую команду:

```
tsc
```

Вы увидите сообщение об ошибке, уже описанное в этом разделе ранее, но компилятор не совершит выход, так как запущен в режиме наблюдения. При этом файл main.js создан не будет.

3. Исправьте ошибку, и код будет автоматически перекомпилирован. Посмотрите и убедитесь, что на этот раз файл main.js будет создан.

Если вы хотите выйти из режима наблюдения, просто нажмите Ctrl+C на клавиатуре, находясь в окне терминала.

СОВЕТ Для начала нового проекта TypeScript запустите команду tsc `--init` в любой директории. Она создаст за вас файл tsconfig.json, содержащий все опции компилятора, большинство из которых будут закомментированы. Раскомментируйте их при необходимости.

ПРИМЕЧАНИЕ Файл tsconfig.json может наследовать конфигурацию из другого файла с помощью свойства `extends`. В главе 10 мы рассмотрим пример проекта, использующего три файла конфигурации: первый с обычными опциями компилятора tsc для всего проекта, второй для клиента и третий для серверной части проекта. Подробнее читайте в разделе 10.4.1.

1.4. ЗНАКОМСТВО С VISUAL STUDIO CODE

Интегрированные среды разработки (IDE) и редакторы кода повышают производительность разработчиков, при этом TypeScript отлично поддерживается следующими инструментами: Visual Studio Code, Web-Storm, Eclipse, Sublime Text, Atom, Emacs, Vim. Здесь мы решили использовать открытый и бесплатный редактор Visual Studio Code (VS Code), разработанный Microsoft. Вы же для работы с TypeScript можете использовать любой другой редактор или IDE.

ПРИМЕЧАНИЕ Согласно опросу разработчиков на Stack Overflow (https://insights.stackoverflow.com/survey/2019), VS Code является самой популярной средой разработки, которую используют более 50% пользователей. Кстати, этот редактор также написан на TypeScript.

В реальных же проектах хорошая контекстная помощь и поддержка рефакторинга очень важны. Переименование всех вхождений имени переменной или функции

в статически типизированных языках может быть выполнено с помощью IDE моментально, но это не касается JavaScript, который не поддерживает типы. Если вы допустите ошибку в имени функции, класса или переменной в TypeScript-коде, то она будет подчеркнута красным.

Можно скачать VS Code с https://code.visualstudio.com. Процесс установки будет зависеть от ОС вашего компьютера. Подробное описание этого процесса см. в разделе Setup документации VS Code (https://code.visualstudio.com/docs).

После установки запустите программу. Затем, используя опцию меню File>Open, откройте директорию chapter1/vscode, которая прилагается к примерам кода этой книги. В ней содержится файл main.js из предыдущего раздела и простой файл tsconfig.json. На рис. 1.5 аргумент `"10%"` подчеркнут, что указывает на ошибку. Если вы наведете указатель мыши на подчеркнутый код, то будет показано то же сообщение об ошибке, которое мы видели на рис. 1.2.

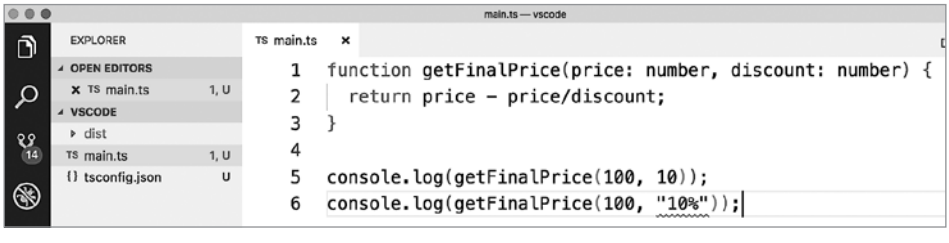


Рис. 1.5. Подчеркивание ошибок в VS Code

РЕЖИМЫ VS CODE ДЛЯ TYPESCRIPT

VS Code поддерживает два режима для TypeScript кода: *область файла* и *явный проект*. Режим области файла ограничен, так как не позволяет скрипту файла использовать переменные, объявленные в других файлах. Режим же явного проекта требует, чтобы файл tsconfig.json находился в директории проекта.

Файл tsconfig.json, относящийся к текущему разделу, прилагается.

Листинг 1.4. vscode/tsconfig.json

```
{
    "compilerOptions": {
        "outDir": "./dist",
        "noEmitOnError": true,
        "lib": ["dom", "es2015"]
    }
}
```

Сохраняет сгенерированные файлы JavaScript в директорию dist

Не генерирует JavaScript до исправления всех ошибок

Библиотеки добавлены, и tsc не будет ругаться на неизвестные API вроде console()

Если вы хотите открывать VS Code из командной строки, то его исполняемый файл нужно будет добавить в переменную окружения PATH вашего компьютера. В Windows процесс настройки должен сделать это автоматически. В MacOS запустите VS Code, выберите опцию меню View > Command Palette, напечатайте `shell command` и выберите эту опцию: Shell Command: Install 'code' Command in PATH. Затем перезапустите терминал и введите `code` из любой директории. Запустится VS Code, и вы сможете работать с файлами из директории, в которой находитесь.

В предыдущем разделе мы компилировали код в отдельном окне терминала, но VS Code имеет встроенный терминал. Благодаря этому исчезает необходимость покидать окно редактора для использования окна командной строки. Чтобы открыть терминал VS Code, выберите в меню View > Terminal или Terminal > New Terminal.

На рис. 1.6 показано, как выглядит интегрированный терминал после выполнения команды `tsc`. Стрелочка справа указывает на значок «плюс», который позволяет открывать любое количество окон терминалов. Мы закомментировали последнюю строку, где есть ошибка, и tsc сгенерирует файл main.js в директорию dist.

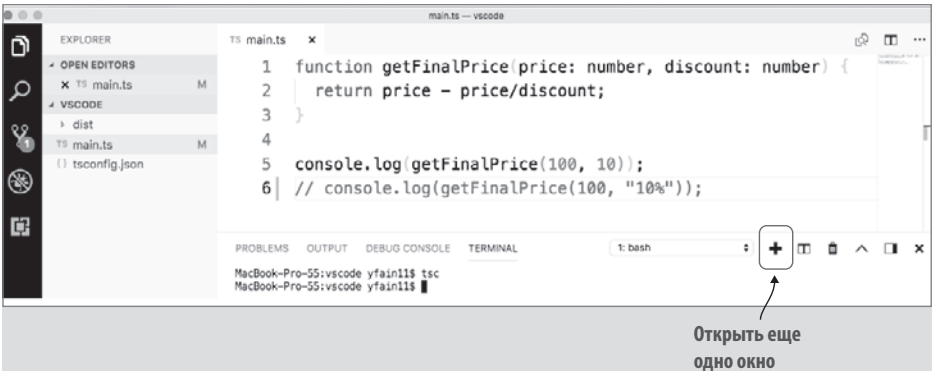


Рис. 1.6. Выполнение команды tsc в VS Code

СОВЕТ VS Code выбирает компилятор tsc, использующийся в Node.JS у вас на компьютере. Откройте любой файл TypeScript, и вы увидите версию tsc, указанную на нижней панели инструментов справа. Если предпочитаете использовать tsc, который установили глобально, щелкните по номеру версии в нижнем правом углу и выберите нужный вам tsc-компилятор.

На рис. 1.6 в нижней части черной панели слева вы можете увидеть квадратную иконку, которая используется для поиска и установки расширений с маркетплейса VS Code. Такие расширения могут сделать программирование в TypeScript более гибким:

- *ESLint* — интегрирует линтер JavaScript и проверяет код на читаемость и обслуживаемость.
- *Prettier* — обеспечивает устойчивый стиль, считывая код и переформатируя его согласно своим правилам.
- *Path Intellisense* — автоматически подставляет пути файлов.

Более подробно об использовании VS Code для TypeScript вы можете прочитать в документации продукта: https://code.visualstudio.com/docs/languages/typescript.

СОВЕТ Есть прекрасная онлайн-IDE под названием StackBlitz (https://stackblitz.com). Основана она на технологии VS Code и не требует установки на компьютер.

ПРИМЕЧАНИЕ Часть 2 книги содержит различные версии примера блокчейн-приложения. Несмотря на то что часть 2 необязательна к прочтению, рекомендуем ознакомиться хотя бы с главами 8 и 9.

ИТОГИ

- TypeScript — это надмножество JavaScript. Программа, написанная на TypeScript, должна быть сначала транспилирована в JavaScript, а только затем может быть выполнена в браузере или отдельном движке JavaScript.
- Ошибки перехватываются статическим анализатором кода TypeScript во время набора еще до компиляции кода компилятором tsc.
- TypeScript дает преимущества статически типизированных языков везде, где это нужно, не препятствуя при этом использованию старых добрых динамических объектов JavaScript.
- TypeScript следует новейшим спецификациям стандарта ECMAScript и добавляет к ним типы, интерфейсы, декораторы, переменные членов классов (поля), обобщенные типы, перечисления, ключевые слова `public`, `protected`, `private` и другие возможности. Ознакомьтесь с дорожной картой TypeScript здесь: https://github.com/Microsoft/TypeScript/wiki/Roadmap, чтобы узнать о текущих и будущих возможностях.
- Чтобы начать новый проект TypeScript, выполните команду `tsc --init` в любой директории. Она создаст за вас файл tsconfig.json, который будет содержать все опции компилятора, большинство из которых будут закомментированы. Раскомментируйте их по необходимости.

2 Базовые и пользовательские типы

В этой главе:

- ✓ Объявление переменных с типами и использование типов в объявлениях функций.
- ✓ Объявление псевдонимов типов с помощью ключевого слова type.
- ✓ Объявление пользовательских типов с помощью классов и интерфейсов.

Можно рассматривать TypeScript как JavaScript с типами, хотя это будет очень упрощенное представление, так как в TypeScript имеются элементы синтаксиса, отсутствующие в JavaScript (например, интерфейсы, обобщенные типы и др.). Тем не менее основная сила TypeScript заключается именно в типах.

Несмотря на то что очень желательно объявлять типы идентификаторов до их применения, это не обязательно. В этой главе вы начнете знакомство с разными способами использования встроенных и пользовательских типов. В частности, вы увидите, как использовать классы и интерфейсы для объявления пользовательских типов; далее же рассмотрение классов и интерфейсов продолжится в главе 3.

ПРИМЕЧАНИЕ Если вы не знакомы с синтаксисом современного JavaScript, то рекомендуем сначала прочитать приложение, а затем уже продолжать освоение TypeScript. Это приложение также поможет понять, какие элементы синтаксиса существуют в JavaScript, а какие были добавлены именно в TypeScript.

2.1. ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ С ТИПАМИ

Зачем объявлять типы переменных, если в JavaScript вы можете просто объявить имя переменной и хранить в ней данные любого типа? Написание кода в JS легче, чем в других языках, в основном потому, что вам не нужно указывать типы для идентификаторов, не так ли?

Более того, в JavaScript вы можете присваивать переменной сперва численное значение, а позднее текстовое. Так не выйдет в TypeScript, где, присвоив переменной тип, вы уже не сможете его изменить. Рисунок 2.1 показывает это на примере:

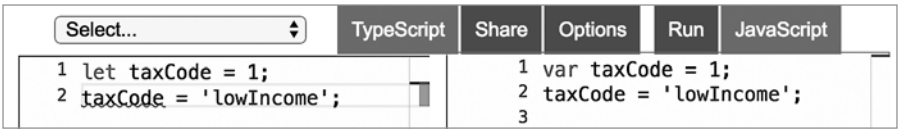


Рис. 2.1. Попытка изменить тип переменной в TypeScript (слева) и JavaScript (справа)

В левой части рис. 2.1 вы видите код TypeScript, написанный в интерактивной среде www.typescriptlang.org. Но где же мы объявили тип переменной `taxCode`? Мы не делали этого явно, но поскольку инициализировали ее с численным значением, TypeScript присвоил `taxCode` тип `number`.

Вторая строка подчеркнута волнистой линией, указывающей на ошибку. Если вы наведете курсор на это подчеркивание, то сообщение об ошибке укажет, что `'lowincome'` не может быть присвоено типу `'number'`. В мире TypeScript это означает, что если вы объявили переменную как хранящую численные значения, то уже не можете присваивать ей строчные. В правой части рис. 2.1 скомпилированный код JavaScript не показывает ошибки, так как JavaScript во время выполнения позволяет присваивать переменной значения различных типов.

Несмотря на то что объявление типов переменных вынуждает разработчиков писать больше кода, их продуктивность растет в долгосрочной перспективе, поскольку чаще всего, если программист пытается присвоить строковое значение переменной, уже хранящей число, — это ошибка. Спасает здесь то, что компилятор может перехватывать подобные ошибки в процессе разработки, а *не* в среде выполнения.

Тип может быть присвоен переменной явно самим разработчиком либо неявно (*вывод* типа) компилятором TypeScript. На рис. 2.1 мы объявили переменную `taxCode`, не присваивая ей тип. Присваивание значения 1 этой переменной позволяет компилятору понять, что ее тип — `number`. Это пример вывода типа. Большинство примеров кода в следующем разделе используют явные типы, за парой исключений, помеченных как выведенные.

2.1.1. Базовые аннотации типов

Когда вы объявляете переменную, то можете добавить двоеточие и аннотацию типа, чтобы указать тип этой переменной:

```
let firstName: string;
let age: number;
```

TypeScript предлагает следующие аннотации типов:

- `string` — для текстовых данных;
- `boolean` — для значений `true/false`;
- `number` — для численных значений;
- `symbol` — уникальное значение, создаваемое вызовом конструктора `Symbol`;
- `any` — для переменных, способных содержать значения разных типов, которые могут быть вам неизвестны в момент написания кода;
- `unknown` — аналог `any`, с которым нельзя производить никаких действий, не утвердив или сузив его до более конкретного типа;
- `never` — для представления невозможного кода (вскоре будет приведен пример);
- `void` — отсутствие значения.

Большинство базовых типов самоописательны и не требуют дополнительных пояснений. Отдельно стоит отметить, что в ECMAScript 2015 был добавлен примитивный тип данных `symbol`, который всегда уникален и неизменен. Например, в следующем фрагменте кода `sym1` не равен `sym2`:

```
const sym1 = Symbol("orderID");
const sym2 = Symbol("orderID");
```

Когда вы создаете новый символ (обратите внимание на отсутствие ключевого слова `new`), вы можете по желанию добавить его описание вроде `orderID`. Обычно символы используются для создания уникальных ключей свойств объектов.

Листинг 2.1. Символы как свойства объектов

```
const ord = Symbol('orderID');   <-- Создает новый символ

const myOrder = {   <-- Использует символ как свойство объекта
    ord: "123"

};
console.log (myOrder['ord']);   <-- Эта строка выводит «123»
```

Будучи надмножеством JavaScript, TypeScript также имеет два специальных значения: `null` и `undefined`. Переменная, которой не было присвоено значение, имеет значение `undefined`. Функция, не возвращающая значение, также имеет значение `undefined`. Значение `null` представляет намеренное отсутствие значения, как в `let someVar = null;`.

Вы можете присвоить значения `null` и `undefined` переменным любого типа, но чаще всего они используются в сочетании со значениями других типов. Следующий фрагмент кода показывает, как вы можете объявить функцию, возвращающую либо `string`, либо `null` (вертикальная черта представляет собой *объединенный тип*, который рассматривается в разделе 2.1.3):

```
function getName(): string | null {
    ...
}
```

Как и в большинстве языков программирования, если вы объявляете функцию, возвращающую `string`, то все равно можете вернуть `null`. Но явное указание того, что может вернуть функция, повышает читаемость кода.

Если вы объявляете переменную типа `any`, то можете присвоить ей любое значение: численное, текстовое, логическое или пользовательское вроде `Customer`. Тем не менее следует избегать использования `any`, так как иначе вы теряете все преимущества проверки типов и читаемость вашего кода также падает.

Тип `never` присваивается ничего не возвращающей функции, то есть той, которая либо выполняется бесконечно, либо просто выбрасывает ошибку. Стрелочная функция в следующем листинге ничего не возвращает, поэтому модуль проверки типов *выведет* (угадает) ее возвращаемый тип как `never`.

Листинг 2.2. Стрелочная функция, возвращающая тип never

```
const logger = () => {
    while (true) {  <-- Эта функция никогда не завершается
        console.log("The server is up and running");
    }
};
```

В приведенном листинге `logger` получает тип `() => never`. В листинге 2.9 вы увидите еще один пример присваивания типа `never`.

Тип `void` не используется при объявлении переменных, а применяется для объявления функции, не возвращающей значение:

```
function logError(errorMessage: string): void {
    console.error(errorMessage);
}
```

В отличие от типа `never`, функция `void` завершает свое выполнение, но не возвращает значение.

> **СОВЕТ** Если тело функции не имеет инструкции `return`, то она все равно возвращает значение `undefined`. Аннотация типа `void` может использоваться для предотвращения случайного возвращения программистами явного значения из функции.

Тот факт, что любая программа JavaScript является рабочей программой TypeScript, означает, что использование аннотаций типов в TypeScript опционально. Если некоторые переменные не имеют явных аннотаций типов, модуль проверки типов TypeScript постарается их вывести. Следующие две строки кода являются рабочим синтаксисом TypeScript:

```
let name1 = 'John Smith';
```
Объявляет и инициализирует переменную без явной аннотации типа

```
let name2: string = 'John Smith';
```
Объявляет и инициализирует переменную с аннотацией типа

Первая строка объявляет и инициализирует переменную `name1` в стиле JavaScript, и мы можем сказать, что для нее выводится тип `string`. Считаете ли вы, что вторая строка — это удачный пример объявления и инициализации переменной `name2` в TypeScript? Скажем так, с точки зрения стиля кода указание типа здесь излишне.

Хоть вторая строка и является корректным синтаксисом TypeScript, указывать тип `string` в ней не обязательно, поскольку переменная инициализирована со строкой и TypeScript выведет тип `name2` как `string`.

Следует избегать явных аннотаций типов там, где компилятор TypeScript сможет вывести их сам. Следующий фрагмент кода объявляет переменные `age` и `yourTax`. В нем нет необходимости указывать типы этих переменных, поскольку компилятор TypeScript и так их выведет.

Листинг 2.3. Идентификаторы с выведенными типами

```
const age = 25;
```
Константа age не объявляет свой тип

```
function getTax(income: number): number {
    return income * 0.15;
}

let yourTax = getTax(50000);
```
Переменная yourTax не объявляет свой тип

TypeScript также позволяет использовать в качестве типов литералы. Следующая строка объявляет переменную *типа* `John Smith`.

```
let name3: 'John Smith';
```

Мы можем сказать, что переменная `name3` имеет литеральный тип `John Smith` и допустит только одно значение — `John Smith`. Любая попытка присвоить другое значение переменной литерального типа будет обозначена модулем проверки типов как ошибка:

```
let name3: 'John Smith';

name3 = 'Mary Lou'; // ошибка: Тип '"Mary Lou"' не может быть присвоен типу
➥ '"John Smith"'
```

Маловероятно, что вы будете использовать строчные литералы для объявления типа, как в случае с переменной `name3`. Но можно использовать строчные литералы в объединениях (объясненных в разделе 2.1.3) и перечислениях (рассмотренных в главе 4).

Вот несколько примеров переменных, объявленных с явными типами:

```
let salary: number;
let isValid: boolean;
let customerName: string = null;
```

РАСШИРЕНИЕ ТИПОВ

Если вы объявляете переменную, не инициализируя ее с конкретным значением, TypeScript использует внутренние типы `null` или `undefined`, которые преобразуются в `any`. Это называется *расширением типов*.

Переменная принимает значение `undefined`:

```
let productId;
productId = null;
productId = undefined;
```

Компилятор TypeScript применяет расширение типов и присваивает тип `any` значениям `null` и `undefined`. Поэтому тип переменной `productId` — это `any`.

Стоит упомянуть, что компилятор TypeScript поддерживает опцию `--strictNullCheck`, запрещающую присвоение `null` переменным с известными типами. В следующем фрагменте кода `productId` имеет тип `number`, а вторая и третья строки не будут скомпилированы, если включить `--strictnullCheck`:

```
let productId = 123;
productId = null; // ошибка компилятора
productId = undefined; // ошибка компилятора
```

Опция `--strictnullCheck` также помогает в перехвате потенциально неопределенных значений. Например, функция может возвращать объект с опциональным свойством, а ваш код может ошибочно предположить, что это свойство присутствует, и попытаться применить к нему функцию.

СОВЕТ Добавляйте явные аннотации типов для сигнатур функций или методов, а также членов публичных классов.

ПРИМЕЧАНИЕ TypeScript включает и другие типы, используемые при взаимодействии с браузером, вроде `HTMLElement` и `Document`. Помимо этого, вы также можете использовать ключевые слова `type`, `class` и `interface`, чтобы объявлять собственные типы вроде `Customer` или `Person`. В следующем разделе мы покажем, как это делать. Там же вы узнаете, как можно комбинировать типы с помощью *объединений*.

Аннотации типов используются не только для объявления типов переменных, но и для объявления типов аргументов функций и их возвращаемых значений. Сейчас мы это рассмотрим.

2.1.2. Типы в объявлениях функций

Функции и функциональные выражения в TypeScript похожи на функции JavaScript, но позволяют вам явно объявлять типы их аргументов и возвращаемых значений.

Давайте начнем с написания JavaScript-функции (без аннотации типов), которая вычисляет налог. Функция в следующем листинге имеет три параметра и будет производить вычисление на основе штата проживания, дохода и количества иждивенцев. За каждого иждивенца гражданин имеет право на налоговый вычет в размере от 300 до 500 долларов, в зависимости от штата проживания.

Листинг 2.4. Вычисление налога в JavaScript

```
function calcTax(state, income, dependents) {
    if (state === 'NY') {
        return income * 0.06 - dependents * 500;
    } else if (state === 'NJ') {
        return income * 0.05 - dependents * 300;
    }
}
```

Аргументы функции не имеют аннотации типов

Вычисляет налог для Нью-Йорка

Вычисляет налог для Нью-Джерси

Предположим, что гражданин с доходом 50 000 долларов проживает в штате Нью-Джерси и имеет двух иждивенцев. Давайте вызовем `calcTax()`:

```
let tax = calcTax('NJ', 50000, 2);
```

Переменная `tax` получает значение 1,900, являющееся верным. Даже несмотря на то что `calcTax()` не объявляла типы для параметров функции, мы догадались, как вызвать эту функцию, исходя из имен параметров.

А если бы мы ошиблись в догадке? Давайте вызовем ее неверным способом, передав в качестве числа иждивенцев значение `string`:

```
let tax = calcTax('NJ', 50000, 'two');
```

Вы не узнаете о проблеме, пока не вызовете эту функцию. Переменная `tax` будет иметь значение `NaN` (не число). Баг прокрался только потому, что вы не смогли явно указать типы параметров, а компилятор не смог вывести типы аргументов функции.

Следующий листинг показывает TypeScript версию этой функции, использующую аннотации типов для аргументов функции и возвращаемого значения.

Листинг 2.5. Вычисление налога в TypeScript

```
function calcTax(state: string, income: number, dependents: number) : number {
    if (state === 'NY'){
        return income * 0.06 - dependents * 500;
    } else if (state ==='NJ'){
        return income * 0.05 - dependents * 300;
    }
}
```

Аргументы функции и ее возвращаемое значение имеют аннотации типов

Теперь уже не выйдет допустить ту же ошибку и передать для числа иждивенцев значение `string`:

```
let tax: number = calcTax('NJ', 50000, 'two');
```

Компилятор TypeScript отобразит ошибку: «Аргумент типа string не может быть присвоен параметру типа number». Более того, возвращаемое значение функции объявлено как `number`, что мешает вам допустить другую ошибку и присвоить результат вычисления налога не численной переменной:

```
let tax: string = calcTax('NJ', 50000, 'two');
```

ИСПРАВЛЕНИЕ ФУНКЦИИ CALCTAX()

В этом разделе приведены TypeScript и JavaScript версии функции `calcTax()`, но они обрабатывают только два штата: Нью-Йорк и Нью-Джерси. Вызов этих функций для любого другого штата вернет `undefined` в среде выполнения.

Компилятор TypeScript не предупредит вас о том, что функция в листинге 2.5 написана плохо и может вернуть `undefined`, но синтаксис TypeScript позволяет вам предупредить читающего код человека, что функция в листинге 2.5 может вернуть не только число, но и значение `undefined`, если будет вызвана со штатом, отличным от Нью-Йорка или Нью-Джерси. Для объявления подобного случая использования вам следует изменить сигнатуру функции так:

```
function calcTax(state: string, income: number, dependents: number) :
➥ number | undefined
```

Компилятор это перехватит и выдаст ошибку «Тип 'number' не может быть присвоен типу 'string': var tax: string». Такая проверка типов в процессе компиляции может сэкономить вам уйму времени работы над любым проектом.

2.1.3. Объединенный тип

Объединения позволяют вам выражать значение, которое может состоять из нескольких типов. Вы можете объявлять пользовательский тип, основанный на двух или более существующих типах. Например, вы можете объявить переменную типа, который может принимать либо значение `string`, либо значение `number` (вертикальная черта означает *объединение*):

```
let padding: string | number;
```

Несмотря на то что переменная `padding` может хранить значение одного из указанных типов, в любой конкретный момент времени она может иметь только один из них — либо `string`, либо `number`.

TypeScript поддерживает тип `any`, но предыдущий вариант объявления имеет преимущества в сравнении с объявлением `let padding: any`. Листинг 2.6 показывает один образец кода из документации TypeScript (см. http://mng.bz/5742). Эта функция может добавить левый отступ для предоставленной строки. Отступ может быть определен либо как строка, предшествующая указанному значению, либо как число пробелов, которые должны предшествовать указанной строке.

Листинг 2.6. padLeft с типом any

```
function padLeft(value: string, padding: any ): string {
    if (typeof padding === "number") {
        return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") {
        return padding + value;
    }
    throw new Error(`Expected string or number, got '${padding}'.`);
}
```

Предоставляет строку и padding (отступ) типа any

Для численного аргумента генерирует пробелы

Для строки использует конкатенацию

Если второй аргумент ни строка, ни число, выбрасывает ошибку

Следующий листинг демонстрирует использование `padLeft()`:

Листинг 2.7. Вызов функции padLeft

```
console.log( padLeft("Hello world", 4));
console.log( padLeft("Hello world", "John says "));
console.log( padLeft("Hello world", true));
```

Возвращает "Hello world"

Возвращает «John says Hello world»

Ошибка среды выполнения

ЗАЩИТЫ ТИПА TYPEOF И INSTANCEOF

Попытка применить условные инструкции для уточнения типа переменной называется *сужением типа*. В инструкции `if` листинга 2.6 мы использовали защиту типа `typeof` для сужения типа переменной, которая может хранить более одного типа. Мы использовали `typeof` для нахождения действительного типа `padding` в среде выполнения.

Схожим образом защита типа `instanceof` используется с пользовательскими типами (конструкторами), о чем подробнее рассказано в разделе 2.2. Защита `instanceof` позволяет проверить действительный тип при выполнении:

```
if (person instanceof Person) {...}
```

Различие между `typeof` и `instanceof` состоит в том, что первая используется со встроенными типами, а последняя — с пользовательскими.

СОВЕТ В разделе 2.2.4 мы объясним структурную систему типов, реализованную в TypeScript. Коротко говоря, объект, созданный с помощью синтаксиса объектного литерала (синтаксис с фигурными скобками), может быть использован там, где ожидается объект класса (вроде `Person`), если объектный литерал имеет те же свойства, что и `Person`. Из-за этого `if (person instanceof Person)` может дать вам ложное отрицание, если переменная `person` будет указывать на объект, созданный конструктором класса `Person`.

Если теперь мы изменим тип `padding` на объединение `string` и `number` (как показано в следующем листинге), то компилятор сообщит об ошибке, если вы попробуете вызвать `padLeft()`, предоставив что-либо, кроме `string` или `number`. Это также устранит необходимость выбрасывания исключения.

Листинг 2.8. padLeft с объединенным типом

```
function padLeft(value: string, padding: string | number ): string {
    if (typeof padding === "number") {
        return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") {
        return padding + value;
    }
}
```

Позволяет использовать в качестве второго аргумента только строку или число

Теперь вызов `padLeft()` с неверным типом для второго аргумента (например, `true`) вернет ошибку компиляции:

```
console.log( padLeft("Hello world", true)); // compilation error
```

СОВЕТ Если нужно объявить переменную, которая может содержать значения нескольких типов, не используйте тип `any`; используйте объединение вроде `let padding: string | number`. Еще один способ — это объявить две отдельные переменные: `let paddingStr: string; let paddingNum: number`.

Давайте изменим код в листинге 2.8, чтобы показать тип `never`, добавив случай `else` к инструкции `if`. Следующий листинг показывает, как модуль проверки типов выведет тип `never` для невозможного значения.

Листинг 2.9. Тип never невозможного значения

```
function padLeft(value: string, padding: string | number ): string {
    if (typeof padding === "number") {
        return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") {
        return padding + value;
    }
    else {
        return padding;  <-- Этот блок else никогда не выполняется
    }
}
```

Поскольку мы объявили в сигнатуре функции, что аргумент `padding` может быть либо `string`, либо `number`, любое другое значение для `padding` будет невозможным. Другими словами, случай `else` невозможен, и модуль проверки типов для переменной `padding` в случае `else` выведет тип `never`. Вы можете увидеть это сами, скопировав код из листинга 2.9 в игровую площадку TypeScript и наведя курсор на переменную `padding`.

ПРИМЕЧАНИЕ Еще одно преимущество объединенных типов в том, что IDE имеют функцию автоподстановки, которая предложит допустимые типы аргументов, и вы уже не сможете сделать в них ошибку.

Сравните код функций `padLeft()` в листингах 2.6 и 2.9. Каковы основные преимущества использования объединения `string | number` вместо типа `any` во втором аргументе? Если вы используете объединение, компилятор TypeScript предотвратит некорректные вызовы `padLeft()`, сообщив об ошибке во время компиляции.

Здесь мы использовали только объединение примитивных типов (`string` и `number`), но в следующем разделе вы увидите, как объявлять пользовательские объединенные типы.

2.2. ОПРЕДЕЛЕНИЕ ПОЛЬЗОВАТЕЛЬСКИХ ТИПОВ

TypeScript позволяет вам создавать пользовательские типы с помощью ключевого слова `type`, объявлением класса или интерфейса либо объявлением `enum` (рассматривается в главе 4). Мы начнем знакомство с ключевого слова `type`.

2.2.1. Использование type

Ключевое слово `type` позволяет вам объявлять новый тип или псевдоним типа для уже существующего. Предположим, что ваше приложение работает с пациентами, представленными по их именам, росту и весу. Рост и вес являются числами, но для повышения читаемости кода вы можете создать псевдонимы, указывающие единицы измерения этих показателей.

Листинг 2.10. Объявление псевдонимов типов Foot и Pound

```
type Foot = number;
type Pound = number;
```

Вы можете создать новый тип `Patient` и использовать приведенные псевдонимы в его объявлении.

Листинг 2.11. Объявление нового типа, использующего псевдонимы

```
type Patient = {   <-- Объявляет тип Patient
    name: string;
    height: Foot;   <-- Использует псевдоним Foot
    weight: Pound;   <-- Использует псевдоним Pound
}
```

Объявления псевдонимов типов не генерируют код в компилированном JavaScript. В TypeScript объявление и инициализация переменной типа `patient` может выглядеть так:

Листинг 2.12. Объявление и инициализация свойств типа

```
let patient: Patient = {   <-- Мы создаем экземпляр, используя нотацию объектного литерала
    name: 'Joe Smith',
    height: 5,
    weight: 100
}
```

Что, если в процессе инициализации переменной `patient` вы забудете указать значение одного из свойств, например `weight`?

Листинг 2.13. Забыли добавить свойство weight

```
let patient: Patient = {
    name: 'Joe Smith',
    height: 5
}
```

TypeScript выдаст ошибку:

```
"Тип '{ name: string; height: number; }' не может быть присвоен типу 'Patient'.
  Свойство 'weight' упущено в типе '{ name: string; height: number; }'."
```

Если вы хотите объявить некоторые свойства опциональными, то должны добавить вопросительный знак к их именам. В следующем объявлении типа предоставление значения для свойства `weight` опционально и ошибок его отсутствие не вызовет:

Листинг 2.14. Объявление опциональных свойств

```
type Patient = {
    name: string;
    height: Height;
    weight?: Weight;  <-- Свойство weight опционально
}

let patient: Patient = {  <-- Переменная patient инициализирована без weight
    name: 'Joe Smith',
    height: 5
}
```

> **СОВЕТ** Вы можете использовать вопросительный знак для определения опциональных свойств также в классах или интерфейсах, с которыми познакомитесь в этом же разделе чуть позже.

Еще можно использовать ключевое слово `type`, чтобы объявлять псевдонимы типов для сигнатуры функции. Представьте, что пишете фреймворк, который должен позволять создавать элементы управления и присваивать им функции-валидаторы. Функция-валидатор должна иметь конкретную сигнатуру — она должна принимать объект типа `FormControl` и возвращать либо объект, описывающий ошибки значения элемента управления, либо `null`, если значение допустимо. Можно объявить новый тип `ValidatorFn` так:

```
type ValidatorFn =
    (c: FormControl) => { [key: string]: any }| null
```

Здесь `{ [key: string]: any }` означает объект, который может иметь свойства любого типа, но ключ `(key)` должен либо иметь тип строки, либо допускать преобразование в строку.

Конструктор `FormControl` может иметь параметры для функции-валидатора и использовать пользовательский тип `ValidatorFn` следующим образом:

```
class FormControl {

    constructor (initialValue: string, validator: ValidatorFn | null) {...};
}
```

СОВЕТ В приложении можно найти синтаксис для объявления опциональных параметров функций в JavaScript. Предыдущий фрагмент кода показывает способ объявления опционального параметра в TypeScript при помощи объединенного типа.

2.2.2. Использование классов в качестве пользовательских типов

Предположим, что вы знакомы с классами JavaScript, рассмотренными в приложении. В этом разделе мы начнем с показа дополнительных возможностей, привносимых TypeScript в классы JavaScript. В главе 3 мы продолжим более подробное рассмотрение классов.

JavaScript не имеет синтаксиса для объявления свойств классов, но в TypeScript такой синтаксис присутствует. На рис. 2.2 слева вы можете видеть, как мы объявили и инстанцировали класс `Person` с тремя свойствами. Справа видна версия ES6 кода, произведенного компилятором TypeScript.

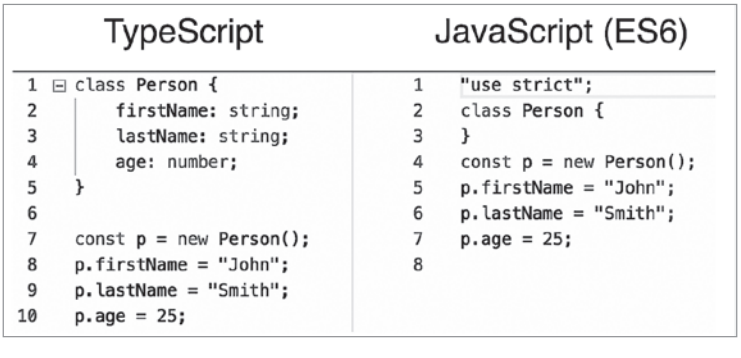


Рис. 2.2. Класс Person, скомпилированный в JavaScript (ES6)

В версии JavaScript нет свойств. Также, поскольку класс `Person` не объявляет конструктор, пришлось инициализировать его свойства после инстанцирования. Конструктор — это особая функция, выполняемая один раз при создании экземпляра класса.

Объявление конструктора с тремя аргументами позволило бы инстанцировать класс `Person` и инициализировать его свойства в одной строке. В TypeScript вы можете обеспечить аннотации типов для аргументов конструктора, но это еще не все.

TypeScript предлагает квалификаторы уровня доступа `public`, `private` и `protected` (рассматриваются в главе 3), и если вы используете любой из них с аргументами конструктора, компилятор TypeScript сгенерирует код для добавления этих аргументов в качестве свойств в скомпилированный объект JavaScript (рис. 2.3).

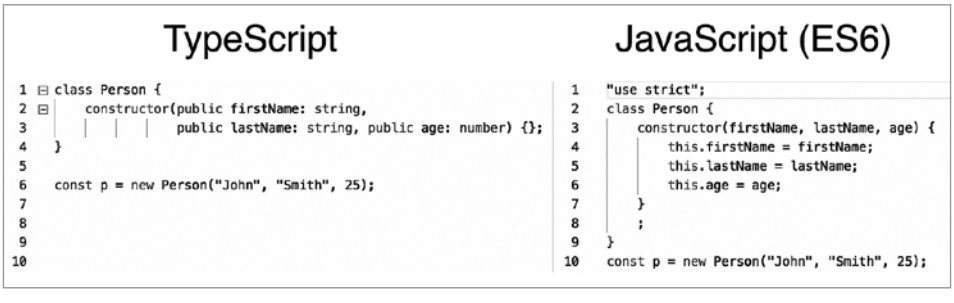


Рис. 2.3. Класс Person с конструктором

Теперь код класса TypeScript (слева) более лаконичен, а сгенерированный код JavaScript создает три свойства в конструкторе. Обратите внимание на строку 6 на рис. 2.3 слева. Мы объявили константу, не указав для нее тип, но также можем переписать эту строку, обозначив тип `p` явно:

```
const p: Person = new Person("John", "Smith", 25);
```

Такая аннотация типа здесь необязательна. Так как вы объявляете константу `p` и тут же инициализируете ее с объектом известного типа (`Person`), модуль проверки типов сможет легко вывести и присвоить ее тип. Сгенерированный код JavaScript будет выглядеть так же, независимо от того, укажете вы тип `p` или нет. Вы можете увидеть, как инстанцировать класс `Person` без явного объявления его типа, в интерактивной среде TypeScript по адресу http://mng.bz/zlV1. Наведите курсор на переменную `p` — ее тип отобразится как `Person`.

> **СОВЕТ** На рис. 2.3 с каждым аргументом конструктора класса мы использовали уровень доступа `public`. Это означает, что к соответствующим сгенерированным свойствам может обратиться любой код, расположенный как внутри, так и вне класса.

Когда вы объявляете свойства класса, то можете также отметить их как `readonly`. Такие свойства могут быть инициализированы либо в точке объявления, либо

в конструкторе класса, после чего их значения уже не допускают изменения. Квалификатор `readonly` аналогичен ключевому слову `const`, но последнее не может быть использовано со свойствами класса.

В главе 8 мы начнем разработку блокчкейн-приложения, а любое такое приложение состоит из блоков с неизменяемыми свойствами. Это приложение будет включать класс `Block`, чей фрагмент приведен ниже.

Листинг 2.15. Свойства класса Block

```
class Block {

    readonly nonce: number;
    readonly hash: string;

    constructor (
        readonly index: number,
        readonly previousHash: string,
        readonly timestamp: number,

        readonly data: string
    ) {
        const { nonce, hash } = this.mine();
        this.nonce = nonce;
        this.hash = hash;
    }
    // Оставшаяся часть кода опущена в целях сокращения
}
```

Это свойство инициализировано в конструкторе

Значение для этого свойства предоставлено конструктором во время инстанцирования

Использует деструктуризацию для извлечения значений из объекта, возвращаемого методом mine()

Класс `Block` включает шесть свойств `readonly`. Обратите внимание, что нам не нужно явно объявлять свойства класса для аргументов конструктора, которые имеют квалификаторы `readonly`, `private`, `protected` или `public`, как мы делали бы в других объектно-ориентированных языках. В листинге 2.15 два свойства класса объявлены явно, а четыре неявно.

2.2.3. Интерфейсы в качестве пользовательских типов

Многие объектно-ориентированные языки включают синтаксические конструкции под названием `interface`, которые используются для обеспечения реализации указанных свойств или методов в объекте. JavaScript не поддерживает интерфейсы, но в TypeScript они есть. В этом разделе мы покажем, как использовать их для объявления пользовательских типов, а в главе 3 вы увидите, как использовать интерфейсы, чтобы гарантировать реализацию классом указанных членов.

TypeScript осуществляет поддержку интерфейсов с помощью ключевых слов `interface` и `implements`, но сами интерфейсы в JavaScript-код не компилируются. Они только помогают избежать использования неверных типов в процессе

разработки. Давайте познакомимся с использованием ключевого слова `interface` для объявления пользовательского типа.

Предположим, нужно написать функцию, которая сможет сохранять информацию о людях в отдельное хранилище. Эта функция должна получать объект, представляющий человека, и вам нужно обеспечить, чтобы у этого объекта были свойства конкретных типов. Можно объявить интерфейс `Person` так:

Листинг 2.16. Объявление пользовательского типа с помощью интерфейса

```
interface Person {
    firstName: string;
    lastName: string;
    age: number;
}
```

Скрипт в левой части рис. 2.2 объявляет аналогичный пользовательский тип `Person`, но при помощи ключевого слова `class`. В чем же разница? Если вы объявите пользовательский тип как `class`, то сможете использовать его как значение, то есть сможете инстанцировать его, используя ключевое слово `new`, как это показано на рис. 2.2 и 2.3.

Помимо этого, при использовании `class` в коде TypeScript в сгенерированном JavaScript-коде появится соответствующий ему код (функция в ES5 и класс в ES6). Если же вы используете ключевое слово `interface`, никакой соответствующий ему код в JavaScript создан не будет (рис. 2.4).

```
1  interface Person {
2      firstName: string;
3      lastName: string;
4      age: number;
5  }
6
7  function savePerson (person: Person): void {
8      console.log('Saving ', person);
9  }
10
11 const p: Person = {
12             firstName: "John",
13             lastName: "Smith",
14              age: 25 };
15
16 savePerson(p);
```

```
1  "use strict";
2  function savePerson(person) {
3      console.log('Saving ', person);
4  }
5  const p = {
6      firstName: "John",
7      lastName: "Smith",
8      age: 25
9  };
10 savePerson(p);
11
```

Рис. 2.4. Пользовательский тип Person как интерфейс

В правой части рис. 2.4 нет никакого упоминания интерфейса и JavaScript более лаконичен, что хорошо для любого развертываемого кода. Однако в процессе разработки компилятор проверит, чтобы объект, предоставленный в качестве

аргумента функции `savePerson()`, включал все свойства, объявленные в интерфейсе `Person`.

КАКОЕ КЛЮЧЕВОЕ СЛОВО ИСПОЛЬЗОВАТЬ: TYPE, INTERFACE ИЛИ CLASS?

Мы показали вам, что пользовательский тип может быть объявлен с помощью ключевых слов `type`, `class` либо `interface`. Какое же из них лучше всего использовать для объявления пользовательского типа `Person`?

Если этот тип не нужен для инстанцирования объектов в среде выполнения, используйте `interface` или `type`. В противном случае используйте `class`. Другими словами, используйте `class` для создания пользовательского типа, если он будет задействоваться в качестве значения.

Например, если вы объявите интерфейс `Person` и при этом у вас будет функция, получающая аргумент типа `Person`, то вы не сможете применить к этому аргументу оператор `instanceof`:

```
interface Person {
    name: string;
}

function getP(p: Person){
      if (p instanceof Person){ // ошибка компиляции
     }
}
```

Модуль проверки типов выдаст ошибку, что `Person` относится только к типу, но используется здесь как значение.

Если вы объявляете пользовательский тип просто для дополнительной безопасности, предлагаемой модулем проверки типов в TypeScript, используйте `type` или `interface`. Ни интерфейсы, ни типы, объявленные с ключевым словом `type`, не имеют представления в генерируемом JavaScript-коде, что уменьшает размер кода среды выполнения (байтовый). Если же для объявления типов вы используете классы, то они оставят свой след в итоговом JavaScript.

Определение пользовательского типа с ключевым словом type предлагает те же возможности, что и `interface`, плюс некоторые дополнительные. Например, вы не можете использовать типы, объявленные как интерфейсы, в объединениях или пересечениях. В главе 3 вы также узнаете об условных типах, которые не могут быть объявлены с помощью интерфейсов.

Поэкспериментируйте с фрагментом кода рис. 2.4 в песочнице TypeScript по следующей ссылке: http://mng.bz/MOpB. Например, удалите свойство `lastName`,

определенное на строке 13, и модуль проверки типов тут же подчеркнет переменную p красной волнистой линией. Наведите курсор на переменную p, и вы увидите следующее сообщение:

```
Тип '{ firstName: string; age: number; }' не совместим с типом 'Person'.
  Свойство 'lastName' упущено в типе '{ firstName: string; age: number; }'.
```

Продолжайте экспериментировать. Постарайтесь обратиться к `person.lastName` внутри `savePerson()`. Если интерфейс `Person` не объявит свойство `lastName`, TypeScript сообщит об ошибке компиляции, но JavaScript в таком случае дал бы сбой при выполнении.

Попробуйте еще кое-что: удалите аннотацию типа `Person` в строке 11. Код по-прежнему будет рабочим, и в строке 17 не появятся ошибки. Почему TypeScript позволил вызвать функцию `savePerson()` с аргументом, которому не был явно присвоен тип `Person`? Причина в том, что TypeScript использует структурную систему типов, а это означает, что если два разных типа включают одинаковые члены, то эти типы считаются совместимыми. В следующем разделе рассмотрим структурную систему типов более подробно.

2.2.4. Структурная система типов против номинальной

Примитивный тип должен иметь имя (например, `number`), в то время как более сложный тип вроде объекта или класса имеет не только имя, но и некоторую структуру, представленную в виде свойств (например, класс `Customer`, скорее всего, будет иметь свойства *имя* и *адрес*).

Как можно понять, являются два типа одинаковыми или нет? В Java (использующем *номинальную систему типов*) два типа одинаковы, если имеют одинаковые имена, объявленные в одном и том же пространстве имен (так называемых пакетов). В номинальной системе типов, если вы объявляете переменную типа `Person`, то можете присвоить ее только объекту типа `Person` или его потомку. В Java последняя строка следующего листинга не скомпилируется, так как имена классов не совпадают, хотя структура у них одинакова.

Листинг 2.17. Фрагмент Java кода

```
class Person {  <-- Объявляет класс Person (считайте «тип»)
    String name;
}

class Customer {  <-- Объявляет класс Customer
    String name;
}

Customer cust = new Person();  <-- Ошибка синтаксиса: имена классов слева и справа не совпадают
```

Но TypeScript и некоторые другие языки используют *структурную систему типов*. Следующий листинг показывает предыдущий фрагмент кода, переписанный в TypeScript.

Листинг 2.18. Фрагмент TypeScript-кода

```
class Person {  <-- Объявляет класс Person (считайте «тип»)
    name: string;
}

class Customer {  <-- Объявляет класс Customer
    name: string;
}

const cust: Customer = new Person();  <-- Никаких ошибок: структуры типов совпадают
```

Код не сообщает об ошибках, так как TypeScript использует структурную систему типов, и поскольку классы `Person` и `Customer` имеют одинаковую структуру, допустимо присваивать экземпляр одного класса переменной другого.

Более того, вы можете использовать объектные литералы для создания объектов и присваивания их `class`-типизированным переменным или константам до тех пор, пока форма этих объектных литералов будет такой же. Следующий код скомпилируется без ошибок:

Листинг 2.19. Совместимые типы

```
class Person {
    name: String;
}

class Customer {
    name: String;
}

const cust: Customer = { name: 'Mary' };
const pers: Person = { name: 'John' };
```

> **ПРИМЕЧАНИЕ** Модификаторы уровня доступа влияют на совместимость типов. Например, если мы объявим свойство `name` класса `Person` как `private`, то код в листинге 2.19 не скомпилируется.

Наши классы не определяли методы, но если бы они оба определили методы с одинаковой сигнатурой (имя, аргументы и возвращаемый тип), то опять же были бы совместимы.

Что, если структуры `Person` и `Customer` не будут полностью одинаковы? Давайте добавим свойство `age` в класс `Person`.

Листинг 2.20. Когда классы не одинаковы

```
class Person {
    name: String;
    age: number;  <-- Мы добавили это свойство
}

class Customer {
    name: String;
}

const cust: Customer = new Person(); // по-прежнему нет ошибок
```

По-прежнему никаких ошибок! TypeScript видит, что `Person` и `Customer` имеют одинаковую форму (что-то общее). Мы просто хотели использовать константу типа `Customer` (у которой есть свойство `name`), чтобы указать на объект типа `Person`, у которого также есть свойство `name`.

Что вы можете сделать с объектом, представленным переменной `cust`? Вы можете написать что-то вроде `cust.name= 'John'`. Экземпляр `Person` имеет свойства `name`, поэтому компилятор не ругается.

> **ПРИМЕЧАНИЕ** Поскольку мы можем присвоить объект типа `Person` переменной типа `Customer`, то можно сказать, что тип `Person` может быть *присвоен* типу `Customer`.

Рассмотрите этот код в песочнице TS: http://mng.bz/adQm. Нажмите Ctrl-пробел сразу после точки в выражении `cust.`, и вы увидите, что доступно только свойство `name`, несмотря на то что класс `Person` также имеет свойство `age`.

В листинге 2.20 класс `Person` имел больше свойств, чем `Customer`, и код компилировался без ошибок. Скомпилируется ли код следующего листинга, если класс `Customer` будет иметь больше свойств, чем `Person`?

Листинг. 2.21. У экземпляра больше свойств, чем у ссылочной переменной

```
class Person {
    name: string;
}

class Customer {
    name: string;
    age: number;
}

const cust: Customer = new Person();  <-- Типы не совпадают
```

Код в листинге 2.21 не скомпилируется, так как ссылочная переменная `cust` будет указывать на объект `Person`, который даже не выделит память для свойства `age`, и присваивание вроде `cust.age = 29` окажется невозможным. В этом случае тип `Person` уже *не может быть присвоен* типу `Customer`.

ПРИМЕЧАНИЕ Мы вернемся к структурной типизации TypeScript в разделе 4.2, когда будем обсуждать обобщенные типы.

2.2.5. Пользовательские объединения типов

В предыдущем разделе мы представили объединенные типы, которые позволяют объявлять, что переменная может иметь один из перечисленных типов. Например, в листинге 2.8 мы указали, что аргумент функции `padding` может быть *либо* `string`, *либо* `number`. Это пример объединения, включающего примитивные типы.

Давайте взглянем на объявление пользовательских объединенных типов. Представьте, что приложение может выполнять различные действия в ответ на активность пользователя. Каждое действие представлено классом с уникальным именем. Каждое действие должно иметь тип и при необходимости может нести полезную нагрузку (`payload`), например поисковый запрос. Следующий листинг включает объявления для трех классов действий и объединенного типа `SearchActions`.

Листинг 2.22. Использование объединения для представления действий в файле actions.ts

```
export class SearchAction {  ◄── Класс с actionType и payload
    actionType = "SEARCH";

    constructor(readonly payload: {searchQuery: string}) {}
}

export class SearchSuccessAction {  ◄── Класс с actionType и payload
    actionType = "SEARCH_SUCCESS";

    constructor(public payload: {searchResults: string[]}) {}
}

export class SearchFailedAction {  ◄── Класс с actionType, но без payload
    actionType = "SEARCH_FAILED";
}

export type SearchActions = SearchAction | SearchSuccessAction |
➥ SearchFailedAction;  ◄── Объявление объединенного типа
```

СОВЕТ Код в листинге 2.22 нуждается в доработке, поскольку простое утверждение, что каждое действие должно иметь свойство, его описывающее, — это больше в стиле программирования на JavaScript. В TypeScript подобное утверждение может быть применено программно, и мы рассмотрим это в разделе 3.2.1.

Размеченные объединения включают членов типов, которые имеют общее свойство — *дискриминант*. В зависимости от значения дискриминанта, вы можете выбрать выполнение разных действий.

Объединение, показанное в листинге 2.22, является примером размеченного объединения типов, поскольку каждый член имеет дискриминант `actionType`. Давайте создадим другое размеченное объединение из двух типов — `Rectangle` и `Circle`.

Листинг 2.23. Использование объединения с дискриминантом для различения форм

```
interface Rectangle {
    kind: "rectangle";  <-- Дискриминант
    width: number;
    height: number;
}

interface Circle {
    kind: "circle";  <-- Дискриминант
    radius: number;
}

type Shape = Rectangle | Circle;  <-- Объединение
```

ЗАЩИТА ТИПА IN

Защита типа `in` действует как сужающее выражение для типов. Например, если у вас есть функция, которая может получать аргумент объединенного типа, то вы можете проверить фактический тип, заданный во время вызова функции.

Следующий код демонстрирует два интерфейса с разными свойствами. Функция `foo()` может получать в качестве аргумента объект `A` или `B`. При помощи защиты типа `in` функция `foo()` перед использованием передаваемого объекта может проверить наличие у него конкретного свойства.

```
interface A { a: number };
interface B { b: string };

function foo(x: A | B) {
    if ("a" in x) {  <-- Проверяет наличие конкретного свойства с помощью in
        return x.a;
    }
    return x.b;
}
```

Проверяемое свойство должно быть строкой, например `"a"`.

Тип `Shape` является размеченным объединением типов, а `Rectangle` и `Circle` имеют общее свойство — `kind`. В зависимости от значения свойства `kind`, мы можем по-разному вычислить площадь `Shape`.

Листинг 2.24. Использование размеченного объединения

```
function area(shape: Shape): number {
    switch (shape.kind) {
        case "rectangle": return shape.height * shape.width;
        case "circle": return Math.PI * shape.radius ** 2;
    }
}

const myRectangle: Rectangle = { kind: "rectangle", width: 10, height: 20 };
console.log(`Rectangle's area is ${area(myRectangle)}`);

const myCircle: Circle = { kind: "circle", radius: 10};
console.log(`Circle's area is ${area(myCircle)}`);
```

Переключает значение дискриминатора

Применяет формулу для прямоугольников

Применяет формулу для кругов

Вы можете запустить этот пример кода в песочнице по адресу http://mng.bz/gVev.

2.3. ТИПЫ ANY И UNKNOWN, А ТАКЖЕ ПОЛЬЗОВАТЕЛЬСКИЕ ЗАЩИТЫ ТИПОВ

В начале этой главы мы упомянули о типах `any` и `unknown`. В этом разделе мы покажем вам их различия. Вы также увидите, как писать пользовательские защиты типов в дополнение к использованию `typeof`, `instanceof` и `in`.

Объявление переменной типа `any` позволяет присваивать ей значение любого типа. Это напоминает написание кода в JavaScript, где вы не указываете тип. Схожим образом попытка обратиться к несуществующему свойству объекта типа `any` может привести к неожиданным результатам при выполнении.

Тип `unknown` был введен в TypeScript 3.0. Если вы объявите переменную типа `unknown`, компилятор вынудит сузить ее тип прежде, чем позволит обращаться к ее свойствам, спасая тем самым от возможных сюрпризов в среде выполнения.

Для демонстрации разницы между `any` и `unknown` давайте предположим, что мы объявили тип `Person` во фронтенде, и он заполняется данными, поступающими из бэкенда в формате `JSON`. Чтобы преобразовать строку JSON в объект, мы используем метод `JSON.parse()`, возвращающий `any`.

Листинг 2.25. Использование типа any

```
type Person = {
    address: string;
}

let person1: any;

person1 = JSON.parse('{ "adress": "25 Broadway" }');

console.log(person1.address);
```

Объявляет псевдоним типа

Объявляет переменную типа any

Считывает строку JSON

Выводит в консоль undefined

Последняя строка выведет `undefined`, потому что мы допустили ошибку в `"address"` в строке `JSON`. Метод `parse` возвращает объект JavaScript, имеющий свойство `adress`, но не `address` для `person1`. Чтобы столкнуться с описанной проблемой, вам потребуется запустить этот код.

Теперь давайте посмотрим, как тот же случай использования работает с переменной типа `unknown`.

Листинг 2.26. Ошибка компиляции с типом unknown

```
let person2: unknown;
person2 = JSON.parse('{ "adress": "25 Broadway" }');

console.log(person2.address);
```

Объявление переменной типа unknown

Попытка использования переменной типа unknown приводит к ошибке компиляции

На этот раз последняя строка даже не скомпилируется, потому что мы попытались использовать переменную `person2` типа `unknown`, не сужая ее тип.

TypeScript позволяет писать пользовательские защиты типов, которые могут проверять, имеет ли объект конкретный тип. Это будет функция, которая возвращает что-то вроде «this FunctionArg is SomeType» (этот аргумент функции имеет тип...). Давайте напишем защиту типа `isPerson()`, которая предполагает, что если проверяемый объект имеет свойство `address`, то это `person`.

Листинг 2.27. Первый вариант защиты типа isPerson

```
const isPerson = (object: any): object is Person => "address" in object;
```

Эта защита типа возвращает `true`, если передаваемый объект содержит свойство `address`. Вы можете применить ее, как показано в следующем листинге.

Листинг 2.28. Применение защиты типа isPerson

```
if (isPerson(person2)) {
    console.log(person2.address);
} else {
    console.log("person2 is not a Person");
}
```

Применяет защиту типа

Безопасно обращается к свойству address

В этом коде отсутствуют ошибки компиляции, и он будет работать как ожидалось, пока защита `isPerson()` не получит в качестве аргумента ложный (вызывающий `false`) объект. Например, передача `null` в `isPerson()` приведет к ошибке среды выполнения в выражении `"address"` в объекте.

Следующий листинг показывает более безопасный вариант защиты `isPerson()`. Оператор двойной восклицательный знак `!!` гарантирует, что переданный объект является верным (возвращает `true`).

Листинг 2.29. Защита типа isPerson

```
const isPerson = (object: any): object is Person => !!object && "address"
➥ in object;
```

Вы можете поработать с этим кодом в песочнице: http://mng.bz/eDaV.

В этом примере мы предположили, что проверки существования свойства `address` достаточно для идентификации типа `Person`, но в некоторых случаях проверки всего одного свойства будет мало. Например, классы `Organization` или `Pet` тоже могут иметь свойства `address`. Поэтому может потребоваться проверять несколько свойств для определения, соответствует ли объект конкретному типу.

Более простым решением будет объявить ваше собственное свойство-дискриминатор, которое будет идентифицировать этот тип как `person`:

```
type Person = {
    discriminator: 'person';
    address: string;
}
```

В этом случае ваша пользовательская защита типа будет выглядеть так:

```
const isPerson = (object: any): object is Person => !!object &&
➥ object.discriminator === 'person';
```

Что ж, мы рассмотрели достаточно примеров синтаксиса TypeScript, относящегося к типам. Теперь настало время применить теорию на практике.

2.4. МИНИ-ПРОЕКТ

Если предпочитаете учиться на деле, то предлагаем небольшое задание, сопровождаемое решением. Не будем предоставлять подробное разъяснение этого решения, так как описание задания должно быть достаточно ясным.

Напишите программу с двумя пользовательскими типами, `Dog` и `Fish`, объявленными с помощью классов. Каждый из этих типов должен иметь свойство `name`. Класс `Dog` должен иметь метод `sayHello(): string`, а класс `Fish` — метод `dive(howDeep: number): string`.

Объявите новый тип `Pet` в качестве объединения `Dog` и `Fish`. Напишите функцию `talkToPet(pet: Pet): string`, которая будет использовать защиты типов и либо вызывать метод `sayHello()` для экземпляра `Dog`, либо выводить сообщение «Fish cannot talk, sorry». (Извините, рыбы не разговаривают.)

Вызовите `talkTopet()` трижды, в первый раз передав объект `Dog`, затем `Fish` и в заключение объект, не являющийся ни `Dog`, ни `Fish`.

Наше решение приведено в следующем листинге:

Листинг 2.30. Решение

```
class Dog {  <-- Объявляет пользовательский тип Dog
     constructor(readonly name: string) { };

    sayHello(): string {
      return 'Dog says hello!';
    }
}

class Fish {  <-- Объявляет пользовательский тип Fish

     constructor(readonly name: string) { };

     dive(howDeep: number): string {
      return `Diving ${howDeep} feet`;
    }

}

type Pet = Dog | Fish;  <-- Создает объединение Dog и Fish

function talkToPet(pet: Pet): string | undefined {

    if (pet instanceof Dog) {  <-- Использует защиту типа
        return pet.sayHello();
    } else if (pet instanceof Fish) {
        return 'Fish cannot talk, sorry.';
    }
}

const myDog = new Dog('Sammy');  <-- Создает экземпляр Dog
const myFish = new Fish('Marry');  <-- Создает экземпляр Fish

console.log(talkToPet(myDog));
console.log(talkToPet(myFish));  <-- Вызывает talkToPet(), передавая Pet
talkToPet({ name: 'John' });  <-- Не скомпилируется из-за неверного типа параметра
```

В действии этот сценарий можно увидеть на CodePen: http://mng.bz/pyjK.

ИТОГИ

- Несмотря на то что объявление типов переменных вынуждает разработчиков писать больше кода, их продуктивность растет в долгосрочной перспективе.

- TypeScript предлагает ряд аннотаций типов, но можно также объявлять пользовательские.
- Можно создавать новые типы, объявляя объединение существующих.
- Можно объявлять пользовательские типы с помощью ключевых слов `type`, `interface` и `class`. В главе 4 вы увидите еще один способ объявления пользовательских типов, уже с использованием ключевого слова `enum`.
- TypeScript использует структурную систему типов в противоположность Java или C#, которые используют номинальную.

3 Объектно-ориентированное программирование с классами и интерфейсами

В этой главе:

- ✓ Принцип работы наследования классов.
- ✓ Где и для чего используются абстрактные классы.
- ✓ Как интерфейсы могут принудить класс иметь методы с известными сигнатурами, не беспокоясь о деталях реализации.
- ✓ Программирование через интерфейсы.

В главе 2 мы познакомились с использованием классов и интерфейсов для создания пользовательских типов. В текущей главе мы продолжим изучение классов и интерфейсов с позиции объектно-ориентированного программирования (ООП). ООП — это стиль программирования, когда ваши программы фокусируются на обработке объектов вместо составления действий (то есть функций). Конечно же, некоторые из этих функций также будут создавать объекты, но в ООП объекты являются центром всего творения.

Разработчики, работающие с объектно-ориентированными языками, используют интерфейсы как способ обусловить классы конкретными API. Помимо этого, в диалогах программистов вы можете часто услышать фразу «программирование через интерфейсы». В этой главе мы объясним, что она означает. Эта глава является быстрым обзором ООП с использованием TypeScript.

3.1. РАБОТА С КЛАССАМИ

Давайте вспомним, что вы узнали о классах TypeScript в главе 2:

- Вы можете объявлять классы со свойствами, которые в других объектно-ориентированных языках называются *переменными-членами*.
- Как и в JavaScript, классы могут объявлять конструкторы, которые вызываются один раз при инстанцировании.
- Компилятор TypeScript преобразует классы в функции-конструкторы JavaScript, если в качестве целевого синтаксиса указан ES5. Если же указана версия ES6 или более поздняя, то классы TypeScript будут перекомпилированы в JavaScript-классы.
- Если конструктор класса определяет аргументы, использующие такие ключевые слова, как `readonly`, `public`, `protected` или `private`, TypeScript создает свойства класса для каждого аргумента.

Однако это еще не все, что касается классов. В текущей главе мы рассмотрим наследование классов, узнаем, для чего нужны абстрактные классы и модификаторы доступа `public`, `protected` и `private`.

3.1.1. Знакомство с наследованием классов

В реальной жизни каждый человек наследует определенные черты от своих родителей. Сходным образом в мире TypeScript вы можете создать новый класс на основе существующего. Например, можно создать класс `Person` с рядом свойств, а затем создать класс `Employee`, который будет *наследовать* все свойства `Person` и при этом объявлять дополнительные. Наследование — это одна из основных особенностей любого объектно-ориентированного языка, в которой слово `extends` объявляет, что один класс наследует от другого.

Рисунок 3.1 — это скриншот песочницы TypeScript (http://mng.bz/O9Yw). Обратите внимание, что мы не использовали явные типы в объявлении свойств класса `Person`. Мы инициализировали свойства `firstName` и `lastName` с пустыми строками и `age` с `0`. Компилятор TypeScript сам выведет типы, основываясь на изначальных значениях.

> **СОВЕТ** В меню настроек песочницы TypeScript опция компилятора `strictPropertyInitialization` включена. Это означает, что если свойство класса не инициализировано в конструкторе класса или во время объявления, то компилятор сообщает об ошибке.

Строка 7 на рис. 3.1 показывает, как вы можете объявить класс `Employee`, расширяющий класс `Person` и объявляющий дополнительное свойство `department`. В строке 11 мы создаем экземпляр класса `Employee`.

Рис. 3.1. Наследование классов в TypeScript

Этот скриншот был сделан после того, как на строке 13 мы ввели `empl.`, после которой нажали Ctrl-пробел. Статический анализатор TypeScript распознает, что тип `Employee` наследован от `Person`, поэтому он предлагает свойства, определенные в обоих этих классах, `Person` и `Employee`.

В нашем примере класс `Employee` является подклассом `Person`, и наоборот, класс `Person` является суперклассом `Employee`. Иначе вы еще можете сказать, что класс `Person` является *предком*, а `Employee` *потомком* `Person`.

> **ПРИМЕЧАНИЕ** Изнутри JavaScript поддерживает *объектное* наследование через прототипы, когда один объект может быть присвоен другому в качестве его прототипа, — это происходит при выполнении. Как только TypeScript-код, использующий наследование, скомпилирован, получившийся JavaScript использует синтаксис прототипного наследования.

В дополнение к свойствам класс может включать *методы* — с помощью которых мы вызываем функции, объявленные внутри классов. При этом метод, объявленный в суперклассе, будет унаследован подклассом, если только он не был объявлен с модификатором доступа `private`, который мы рассмотрим несколько позднее.

Следующая версия класса `Person` показана на рис. 3.2 и включает метод `sayHello()`. (Вы можете найти этот код в песочнице http://mng.bz/YeNz.) Как вы

можете видеть в строке 18, статический анализатор включил этот метод в меню автоподстановки.

Вам может стать интересно: есть ли способ контролировать, какие свойства и методы класса будут доступны из других сценариев? Да! И именно для этого используются ключевые слова `private`, `protected` и `public`.

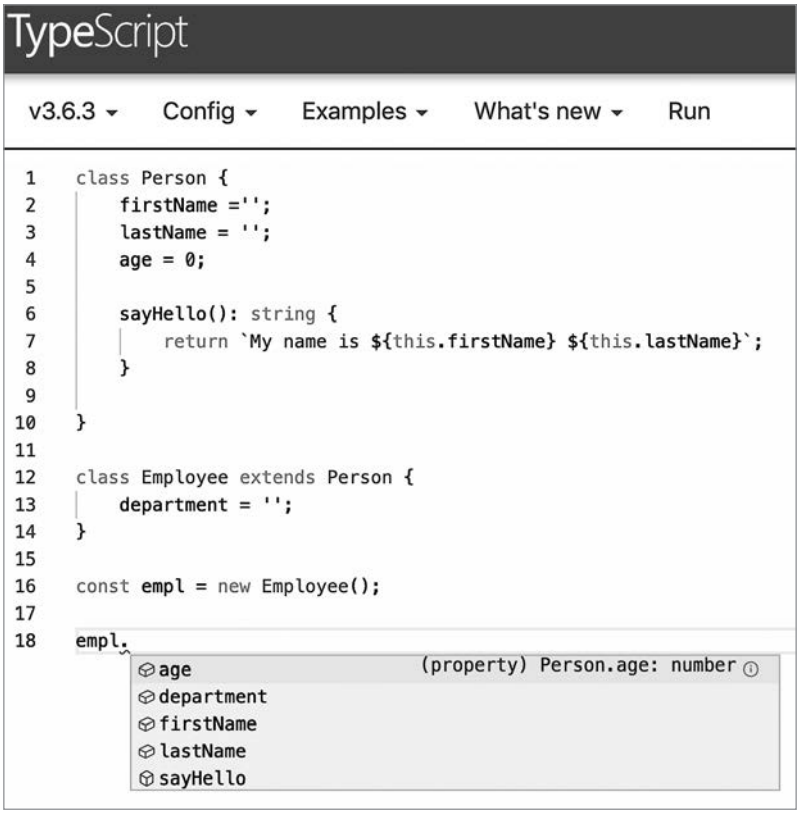


Рис. 3.2. Метод sayHello(), принадлежащий суперклассу, видимый

3.1.2. Модификаторы доступа public, private, protected

TypeScript включает ключевые слова `public`, `protected` и `private` для управления доступом к членам класса (свойствам и методам).

- `public` — к членам класса, отмеченным как `public`, можно обратиться как из внутренних методов класса, так и из внешних сценариев. Это уровень доступа по умолчанию, поэтому если вы поместите ключевое слово `public` перед

свойством или методом класса `Person`, приведенном на рис. 3.2, доступность этих членов класса не изменится.

- `protected` — к членам класса, отмеченным как `protected`, можно обратиться либо из внутреннего кода класса, либо из наследников этого класса.
- `private` — члены класса `private` видимы только внутри класса.

> **ПРИМЕЧАНИЕ** Если вы знаете языки вроде Java или C#, то знакомы с ограничением уровня доступа посредством ключевых слов `private` и `protected`. TypeScript же является надмножеством JavaScript, который не поддерживает ключевое слово `private`, поэтому `private`, `protected` (а также `public`) удаляются при компиляции кода. Итоговый JavaScript-код не будет содержать их, поэтому вы можете рассматривать эти ключевые слова просто как помощь при разработке.

На рис. 3.3 показаны модификаторы доступа `protected` и `private`. В строке 15 мы можем обратиться к методу `sayHello()`, принадлежащему `protected`-предку, потому что мы делаем это из его потомка. Но когда мы нажмем Ctrl-пробел после `this`. в строке 20, переменная `age` не будет показана в списке автоподстановки, так как объявлена как `private` и доступна только внутри класса `Person`.

Этот образец кода показывает, что подкласс не может обратиться к private-члену суперкласса (проверьте это в песочнице http://mng.bz/07gJ). Здесь к private-членам класса может обратиться только метод из класса `Person`.

Несмотря на то что protected члены класса доступны из кода потомка, они недоступны для экземпляра класса. Например, следующий код не скомпилируется и выдаст ошибку «Property 'sayHello' is protected and accessible within class 'Person' and its subclasses». (Свойство `'sayHello'` является `protected` и доступно только внутри класса `'Person'` и его подклассов.)

```
const empl = new Employee(); empl.sayHello(); // ошибка
```

Давайте посмотрим другой пример класса `Person`, имеющий конструктор, два public-свойства и одно private-свойство, как показано на рис. 3.4 (либо в песочнице http://mng.bz/KEgX).

В строке 7 на рис. 3.5 мы создаем экземпляр класса `Person`, передавая изначальные значения свойств его конструктору, который будет присваивать эти значения соответствующим свойствам объекта. На строке 9 мы хотели выводить в консоль значения свойства `firstName` и `age` объекта, но последнее было подчеркнуто красной линией, поскольку `age` является приватным.

Сравните рис. 3.4 и 3.5. На рис. 3.4 класс `Person` явно объявляет три свойства, которые мы инициализируем в конструкторе. На рис. 3.5 класс `Person` не имеет явных объявлений свойств и явных инициализаций в конструкторе.

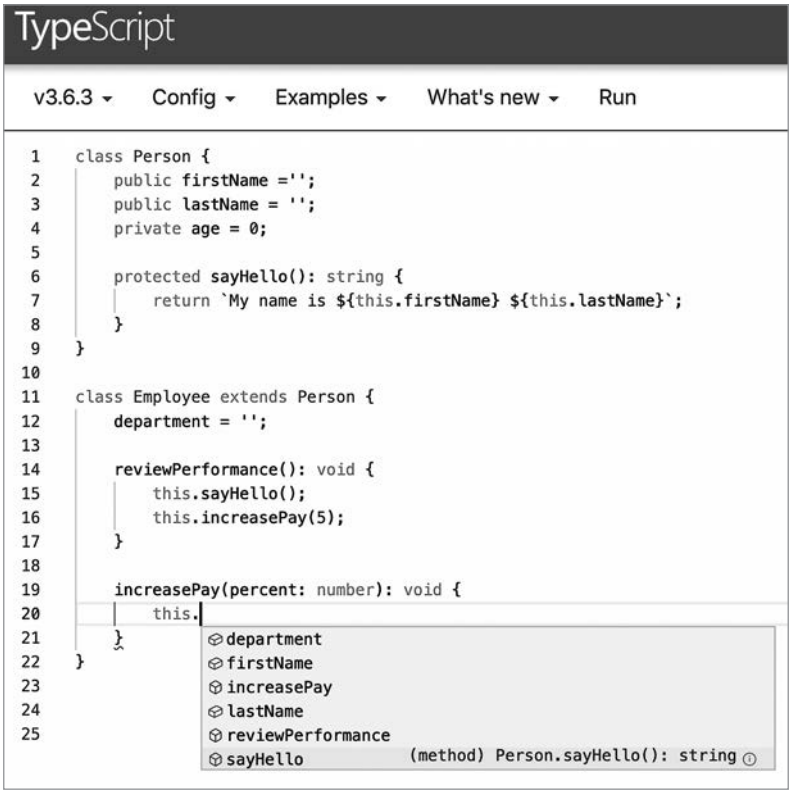


Рис. 3.3. Приватное свойство age невидимо

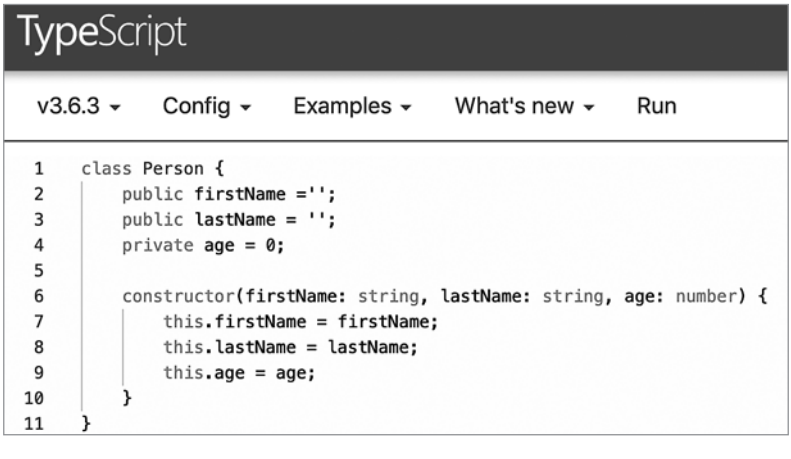


Рис. 3.4. Многословная версия класса Person

Эта версия объявления класса многословна, потому что в ней мы явно объявили три свойства класса. Конструктор в классе `Person` выполняет утомительную работу по присвоению значений из его аргументов к соответствующим свойствам этого класса.

```
class Person {
    constructor(public firstName: string,
                public lastName: string,
                private age: number) { }
}

const pers = new Person('John', 'Smith', 29)

console.log(`${pers.firstName} ${pers.lastName} ${pers.age}`);
```

Рис. 3.5. Использование квалификаторов доступа с аргументами конструктора

Теперь давайте объявим более сжатую версию класса `Person`, как это показано на рис. 3.5 (или в песочнице http://mng.bz/9w9j). Используя квалификаторы доступа с аргументами конструктора, мы можем дать команду компилятору TypeScript создавать свойства класса, имеющие те же имена, что и аргументы конструктора. Компилятор будет автоматически генерировать JavaScript-код, присваивающий значения, переданные конструктору, свойствам класса.

Итак, что же лучше — явное или неявное объявление свойств класса? В пользу обоих этих стилей программирования есть свои доводы. Явное объявление и инициализация свойств класса могут повысить читаемость кода, в то время как неявное объявление делает код класса более сжатым. Но вы не обязаны склоняться именно к одному из этих способов. Например, вы можете объявить `public` свойства явно, а `private` и `protected` неявно. Мы же чаще всего используем неявные объявления, если только инициализация свойства не задействует логику.

3.1.3. Статические переменные и пример Одиночки

В версии ES6, когда свойство должно быть совместно использовано каждым экземпляром класса, мы можем объявить его как `static` (см. раздел A.9.3 приложения). При этом TypeScript, будучи надмножеством JavaScript, тоже поддерживает ключевое слово `static`. В текущем разделе мы рассмотрим простой пример, а затем с помощью свойства `static` и конструктора `private` реализуем паттерн проектирования Одиночка.

Предположим, что на крыше задания находится банда гангстеров (не волнуйтесь — это всего лишь игра). Наша задача — отслеживать общее количество оставшихся у них патронов. Каждый раз, когда гангстер стреляет, это значение

должно уменьшаться на один. При этом общее число патронов должно быть известно каждому гангстеру.

Листинг 3.1. Гангстер со статическим свойством

```
class Gangsta {
    static totalBullets = 100;  ◄── Объявляет и инициализирует статическую переменную

    shoot(){
        Gangsta.totalBullets--;  ◄── Обновляет количество патронов после каждого выстрела
        console.log(`Bullets left: ${Gangsta.totalBullets}`);
    }
}

const g1 = new Gangsta();  ◄── Создает новый экземпляр Gangsta
g1.shoot();  ◄── Этот гангстер стреляет один раз

const g2 = new Gangsta();  ◄── Создает новый экземпляр Gangsta

g2.shoot();  ◄── Этот гангстер стреляет один раз
```

После выполнения кода, приведенного в листинге 3.1 (его можно найти здесь: http://mng.bz/j5Ya), консоль браузера напечатает следующее:

```
Bullets left: 99
Bullets left: 98
```

Оба экземпляра класса `Gangsta` используют одну переменную `totalBullets`. В связи с этим она обновляется независимо от того, какой из гангстеров стреляет.

Обратите внимание, что в методе `shoot()` мы не прописывали `this.totalBullets`, так как это не переменная экземпляра. Вы обращаетесь к static-членам класса, предваряя их имена именем класса, например `Gangsta.totalBullets`.

> **ПРИМЕЧАНИЕ** Статические члены класса не используются подклассами. Если вы создадите класс `SuperGangsta`, включающий `Gangsta` как подкласс, то он получит свою собственную копию свойства `totalBullets`. Мы привели этот пример в песочнице: http://mng.bz/WO8g.

> **СОВЕТ** Зачастую у нас есть несколько способов выполнения схожих действий. Например, может понадобиться написать десяток функций для проверки пользовательского ввода в различных полях UI. Вместо создания раздельных функций вы можете сгруппировать их в класс с десятком `static`-методов.

Теперь давайте рассмотрим другой пример. Представьте, что вам нужно хранить в памяти важные данные, представляющие текущее состояние приложения. К этому хранилищу могут обращаться различные сценарии, но вам нужно обеспечить, чтобы для всего приложения создавался только один такой объект, который будет *единственным источником истины*. В таком случае вы прибегаете

к использованию Одиночки — паттерна проектирования, ограничивающего инстанцирование класса одним объектом.

Как же создать класс, который можно инстанцировать всего один раз? Это тривиальная задача для любого объектно-ориентированного языка программирования, поддерживающего квалификатор доступа `private`. Суть в том, что вам нужно написать класс, который не будет позволять использование ключевого слова `new`, потому что с помощью `new` вы можете создать неограниченное количество экземпляров. Идея проста: если конструктор класса будет `private`, то оператор `new` работать не будет.

Как же тогда вообще можно создать хотя бы один экземпляр такого класса? Если конструктор класса является приватным, вы можете обратиться к нему только внутри класса, и, будучи автором этого класса, вы ответственно создадите его только один раз, вызвав оператор `new` из метода этого класса.

Но как вы сможете вызвать метод для класса, который не был инстанцирован? Вы можете осуществить это, сделав метод класса `static`, чтобы он принадлежал не какому-то конкретному экземпляру объекта, а самому классу.

Листинг 3.2 показывает нашу реализацию Одиночки в классе `AppState`, имеющем свойство `counter`. Давайте предположим, что `counter` представляет состояние нашего приложения и может быть обновлено несколькими его сценариями. Этот одиночный экземпляр `AppState` должен быть единственным местом, хранящим значение `counter`. Любой сценарий, которому потребуется узнать последнее значение `counter`, получит его из этого экземпляра `AppState`.

Листинг 3.2. Класс Одиночка

```
class AppState {
    counter = 0;
    private static instanceRef: AppState;
    private constructor() { }
    static getInstance(): AppState {
        if (AppState.instanceRef === undefined) {
            AppState.instanceRef = new AppState();
        }
        return AppState.instanceRef;
    }
}

// const appState = new AppState(); // ошибка из-за private
➥ constructor

const appState 1 = AppState.getInstance();
const appState2 = AppState.getInstance();
```

Это свойство представляет состояние приложения

Это свойство хранит ссылку на одиночный экземпляр AppState

Приватный конструктор препятствует использованию оператора new с AppState

Это единственный способ получить экземпляр AppState

Инстанцирует объект AppState, если он еще не существует

Эта переменная получает ссылку на экземпляр AppState

```
appState1.counter++;
appState1.counter++;
appState2.counter++;
appState2.counter++;
```

Модифицирует counter (мы используем две ссылочные переменные)

```
console.log(appState1.counter);
console.log(appState2.counter);
```

Выводит в консоль значение counter (мы используем две ссылочные переменные)

Класс `AppState` имеет private-конструктор. Это означает, что никакой другой сценарий не сможет инстанцировать его, используя инструкцию `new`. Вполне приемлемо вызвать подобный конструктор изнутри класса `AppState`, что мы и делаем в нашем статическом методе `getInstance()`. Оба вызова `console.log()` выведут в консоль 4, так как есть только один экземпляр AppState. Этот пример кода можно посмотреть здесь: http://mng.bz/8zKK.

3.1.4. Метод super() и ключевое слово super

Давайте продолжим рассмотрение наследования классов. В строке 15 на рис. 3.3 мы вызвали метод `sayHello()`, объявленный в суперклассе. А что, если и суперкласс, и подкласс имеют методы с одинаковыми именами? Что, если они оба имеют конструкторы? Можем ли мы контролировать, какой именно метод выполняется? Если и у суперкласса, и у подкласса есть конструкторы, то находящийся в подклассе должен вызывать конструктор суперкласса, используя метод `super()`.

Листинг 3.3. Вызов конструктора суперкласса

```
class Person {

    constructor(public firstName: string,
                public lastName: string,
                private age: number) {}
}

class Employee extends Person {

    constructor (firstName: string, lastName: string,
                 age: number, public department: string) {
        super(firstName, lastName, age);

    }
}

const empl = new Employee('Joe', 'Smith', 29, 'Accounting');
```

Конструктор суперкласса Person

Подкласс Employee

Конструктор подкласса Employee

Вызывает конструктор суперкласса

Инстанцирует подкласс

ПРИМЕЧАНИЕ В разделе A.9.2 приложения мы рассматриваем использование `super()` и `super` в JavaScript. В этом же разделе мы предоставим простой пример, но для TypeScript.

Оба класса определяют конструкторы, и мы должны обеспечить, чтобы каждый из них был вызван с правильными параметрами. Конструктор класса `Employee` автоматически вызывается, когда мы используем оператор `new`, но конструктор суперкласса `Person` нам придется вызывать вручную. Класс `Employee` имеет конструктор с четырьмя аргументами, но только один из них — `department` — необходим для создания объекта `Employee`. Три других параметра нужны для создания объекта `Person`, и мы передаем их в `Person` с помощью вызова метода `super()` с тремя аргументами. Вы можете дополнительно поиграть с этим кодом здесь: http://mng.bz/E14q.

Теперь давайте рассмотрим ситуацию, где и суперкласс, и подкласс имеют методы с одинаковым именем. Если метод в подклассе хочет вызвать метод с таким же именем, определенный в суперклассе, то при обращении к этому методу ему нужно использовать ключевое слово `super` вместо `this`.

Предположим, что класс `Person` содержит метод `sellStock()`, подключающийся к фондовой бирже и продающий заданное число акций указанного фонда. В классе `Employee` мы бы хотели использовать эту функциональность повторно, но каждый раз, когда работники продают акции, они должны сообщать об этом в контрольный отдел фирмы.

Мы можем сперва объявить метод `sellStock()` в классе `Employee`, чтобы этот метод вызывал `sellStock()` для `Person`, а затем объявить свой собственный метод `reportToCompliance()`, как это показано в следующем листинге.

Листинг 3.4. Использование ключевого слова super

```
class Person {

    constructor(public firstName: string,
                public lastName: string,
                private age: number) { }

    sellStock(symbol: string, numberOfShares: number) {   <-- Метод sellStock() в предке
      console.log(`Selling ${numberOfShares} of ${symbol}`);
    }
}

class Employee extends Person {

    constructor (firstName: string, lastName: string,
                 age: number, public department: string) {
        super(firstName, lastName, age);   <-- Вызывает конструктор предка
    }

    sellStock(symbol: string, shares: number) {   <-- Метод sellStock() в потомке
      super.sellStock(symbol, shares);   <-- Вызывает sellStock() для предка

      this.reportToCompliance(symbol, shares);
    }
```

```
    private reportToCompliance(symbol: string, shares: number) {
      console.log(`${this.lastName} from ${this.department} sold ${shares}
➥ shares of ${symbol}`);
    }
}

const empl = new Employee('Joe', 'Smith', 29, 'Accounting');
empl.sellStock('IBM', 100);
```

Приватный метод reportToCompliance()

Вызывает sellStock() для объекта Employee

Обратите внимание, что мы объявили метод `reportToCompliance()` как `private`, потому что хотим, чтобы он вызывался всегда только внутренним методом класса `Employee`, но никогда внешним сценарием. Вы можете запустить эту программу в песочнице: http://mng.bz/NeOE, и браузер выведет в консоль следующее:

```
Selling 100 of IBM
Smith from Accounting sold 100 shares of IBM
```

С помощью ключевого слова `super` мы повторно использовали функциональность из метода, объявленного в суперклассе, и, помимо этого, добавили новую.

3.1.5. Абстрактные классы

Если вы добавите ключевое слово `abstract` в объявление класса, то он не сможет быть инстанцирован. Абстрактный класс может включать как методы, которые были реализованы, так и те, что были *только объявлены*.

Зачем вам может понадобиться создавать класс, не допускающий инстанцирования? Вы можете захотеть делегировать реализацию некоторых методов его подклассам и убедиться, что эти методы будут иметь конкретные сигнатуры.

Давайте взглянем на способы использования абстрактных классов. Предположим, что у компании есть внутренние сотрудники и подрядчики и нам нужно спроектировать классы так, чтобы они представляли всех работников компании. Любой объект работника должен поддерживать следующие методы:

- `constructor(name: string)`
- `changeAddress(newAddress: string)`
- `giveDayOff()`
- `promote(percent: number)`
- `increasePay(percent: number)`

В этом сценарии `promote` (повышение) означает один дополнительный выходной и подъем зарплаты на указанный процент. Метод `increasePay()` должен

повышать годовую зарплату для сотрудников, но для подрядчиков должен увеличивать почасовой тариф. Не имеет значения, как мы реализуем эти методы, но важно, чтобы каждый из них имел всего одну инструкцию `console.log()`.

Давайте поработаем над этим присвоением. Понадобится создать классы `Employees` и `Contractor`, которые должны иметь общую функциональность. Например, изменение адресов и назначение выходных должно работать одинаково как для подрядчиков, так и для сотрудников, но повышение их зарплаты уже требует разных реализаций для этих категорий работников.

Вот наш план: мы создадим класс `abstractPerson` с двумя потомками — `Employee` и `Contractor`. Класс `Person` будет реализовывать методы `changeAddress()`, `giveDayOff()_` и `promote()`. Этот класс будет также включать объявление `abstract` метода `increasePay()`, который будет реализован (уже иначе) в подклассах `Person`, как показано в следующем листинге.

Листинг 3.5. Абстрактный класс Person

```
abstract class Person {  <-- Объявляет абстрактный класс

      constructor(public name: string) { };

    changeAddress(newAddress: string ) {  <-- Объявляет и реализует метод
      console.log(`Changing address to ${newAddress}`);
    }

      giveDayOff() { 2((CO5-3))  <-- Объявляет и реализует метод
            console.log(`Giving a day off to ${this.name}`);
    }

      promote(percent: number) {  <-- Объявляет и реализует метод
            this.giveDayOff();
            this.increasePay(percent);  <-- «Вызывает» абстрактный метод
    }

    abstract increasePay(percent: number): void;  <-- Объявляет абстрактный метод
}
```

> **СОВЕТ** Если не хотите допускать вызов метода `giveDayOff()` из внешних сценариев, добавьте `private` в его объявление. Если хотите, чтобы он мог быть вызван только из класса `Person` и его потомков, сделайте этот метод `protected`.

Обратите внимание, что вы можете написать строку, которая, казалось бы, вызывает абстрактный метод. Но поскольку класс абстрактный, он не может быть «инстанцирован», и абстрактный (нереализованный) метод никак не сможет быть выполнен. Если вы захотите создать потомка абстрактного класса, который может быть инстанцирован, то должны реализовать все абстрактные методы этого предка.

Следующий листинг показывает нашу реализацию классов `Employee` и `Constructor`.

Листинг 3.6. Потомки класса Person

```
class Employee extends Person {
    increasePay(percent: number) {  ◄──  Реализует метод increasePay() для сотрудников
        console.log(`Increasing the salary of ${this.name} by ${percent}%`);
    }
}

class Contractor extends Person {
    increasePay(percent: number) {  ◄──  Реализует метод increasePay() для подрядчиков
        console.log(`Increasing the hourly rate of ${this.name} by
➥ ${percent}%`);
    }
}
```

В разделе 2.2.4 при рассмотрении листинга 2.20 мы использовали выражение «*может быть присвоен*». Когда мы имеем `class A extends class B`, это означает, что `class B` более обобщен, а `class A` более конкретен (например, он добавляет больше свойств).

Более конкретный тип может быть присвоен более обобщенному. Именно поэтому вы можете объявить переменную типа `Person` и присвоить ей объект `Employee` или `Contractor`, как это показано в листинге 3.7.

Давайте создадим массив работников с одним сотрудником и одним подрядчиком, а затем произведем итерацию по этому массиву, вызывая для каждого объекта метод `promote()`.

Листинг 3.7. Запуск кампании по повышению

```
const workers: Person[] = [];  ◄──  Объявляет массив типа суперкласса

workers[0] = new Employee('John');
workers[1] = new Contractor('Mary');

workers.forEach(worker => worker.promote(5));  ◄──  Вызывает promote() для каждого объекта
```

Массив `workers` имеет тип `Person`, что позволяет нам также хранить в нем экземпляры объектов-потомков.

> **СОВЕТ** Так как потомки `Person` не объявляют свои собственные конструкторы, конструктор предка будет вызываться автоматически при инстанцировании `Employee` и `Contractor`. Если бы любой из потомков объявил свой собственный конструктор, нам бы пришлось использовать `super()`, чтобы обеспечить вызов конструктора `Person`.

Запустите этот образец кода в песочнице: http://mng.bz/DNvy, и консоль браузера покажет следующее:

```
Giving a day off to John
Increasing the salary of John by 5%
Giving a day off to Mary
Increasing the hourly rate of Mary by 5%
```

Код в листинге 3.7 создает впечатление, что, вызывая `Person.promote()`, мы перебираем объекты типа `Person`. Но некоторые из этих объектов могут иметь тип `Employee`, в то время как другие являются экземплярами `Contractor`. Фактический тип объекта выявляется только при выполнении, что объясняет, почему верная реализация `increasePay()` вызывается для каждого объекта. Это пример полиморфизма — особенности, присущей всем объектно-ориентированным языкам.

PROTECTED-КОНСТРУКТОРЫ

В разделе 3.1.3 мы объявили private-конструктор для создания одиночки — класса, который может быть инстанцирован только один раз. Здесь также есть применение и для protected-конструкторов. Предположим, вам нужно объявить класс, который нельзя инстанцировать, но при этом можно инстанцировать его подклассы. Для этого вы можете объявить protected-конструктор в суперклассе и вызвать его из конструктора подкласса, используя `super()`.

Это имитирует одну из возможностей абстрактных классов. Но класс с protected-конструктором не позволит вам объявлять абстрактные методы, если только сам этот класс не будет объявлен как абстрактный.

В разделе 10.6.1 вы увидите еще один пример использования абстрактного класса для обработки сообщений WebSocket в блокчейн-приложении.

3.1.6. Перегрузка метода

Объектно-ориентированные языки вроде Java и C# поддерживают *перегрузку методов*. Это означает, что класс может *объявлять* более одного метода с одним именем, но разными аргументами. Например, вы можете написать две версии метода `calculateTax()`: одну с двумя аргументами, такими как доход гражданина и количество иждивенцев, а вторую с одним аргументом типа `Customer`, содержащим все нужные данные этого гражданина.

В строго типизированных языках возможность перегружать методы, указывая типы аргументов и возвращаемое значение, очень важна, так как не дает вам просто вызвать метод класса и передать аргумент произвольного типа и при этом

также допускает различное число аргументов. Как бы то ни было, TypeScript — это синтаксический сахар для JavaScript, позволяющий вызывать функцию, передавая при этом большее или меньшее число аргументов, чем объявляет ее сигнатура. JavaScript не будет ругаться, и он не нуждается в поддержке перегрузок функций. Конечно, может возникнуть ошибка среды выполнения, если метод не обработает переданный объект должным образом, но это произойдет при выполнении. TypeScript же предлагает синтаксис для явного объявления каждой допустимой перегруженной сигнатуры метода, избавляя вас от сюрпризов в среде выполнения.

Давайте проверим, будет ли работать следующий код:

Листинг 3.8. Попытка перегрузки метода с ошибкой

```
class ProductService {

    getProducts() {  ◄—— Метод getProducts() без аргументов
        console.log(`Getting all products`);
    }

    getProducts(id: number) { // ошибка  ◄—— Метод getProducts() с одним аргументом
        console.log(`Getting the product info for ${id}`);
    }
}

const prodService = new ProductService();

prodService.getProducts(123);

prodService.getProducts();
```

Компилятор TypeScript сообщит об ошибке: «Duplicate function implementation» (Повторная реализация функции) для второго объявления `getProduct()`, как показано на рис. 3.6.

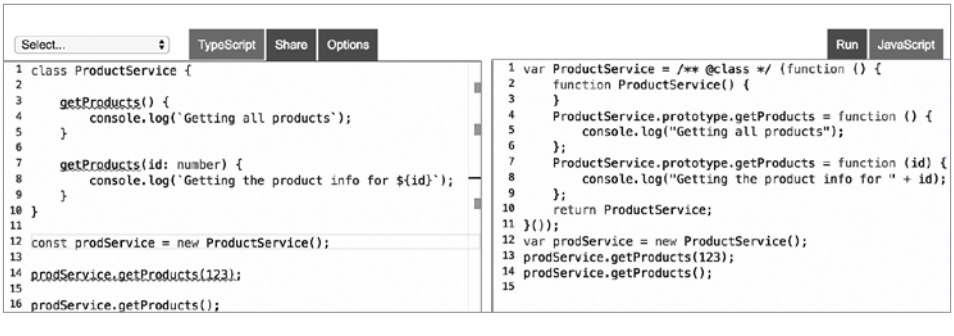


Рис. 3.6. Ошибочный TypeScript, но рабочий JavaScript

Синтаксис в TypeScript-коде ошибочен, но синтаксис JavaScript справа полностью в порядке. Первый вариант метода `getProducts` (строка 4) был замещен вторым (строка 7), поэтому в процессе выполнения вариант JavaScript этого сценария имеет только одну версию `getProducts(id)`.

Давайте проигнорируем ошибки компиляции и попробуем запустить сгенерированный JavaScript в песочнице TypeScript. Консоль браузера выведет сообщения только из метода `getProducts(id)`, даже несмотря на то что мы хотели вызвать другую версию этого метода:

```
Getting the product info for 123
Getting the product info for undefined
```

В скомпилированном JavaScript-коде метод (или функция) может иметь только одно тело, учитывающее все допустимые параметры методов. Тем не менее TypeScript предлагает синтаксис для определения перегрузки метода.

Для этого делается объявление всех допустимых сигнатур методов без реализации самих методов, которое сопровождается одним реализованным методом.

Листинг 3.9. Корректный синтаксис для перегрузки методов

```
class ProductService {

    getProducts(): void;  <-- Объявляет допустимую сигнатуру метода
    getProducts(id: number): void;  <-- Объявляет допустимую сигнатуру метода
    getProducts(id?: number) {  <-- Реализует метод
        if (typeof id === 'number') {
            console.log(`Getting the product info for ${id}`);
        } else {
            console.log(`Getting all products`);
        }
    }
}

const prodService = new ProductService();

prodService.getProducts(123);
prodService.getProducts();
```

Обратите внимание на вопросительный знак после аргумента `id` в реализованном методе. Этот вопросительный знак объявляет, что аргумент опционален. Если бы мы не сделали его опциональным, компилятор выдал бы ошибку «Overload signature is not compatible with function implementation» (Перегруженная сигнатура несовместима с реализацией функции). В нашем примере кода это означает, что если мы объявим сигнатуру метода `getProducts()` без аргумента, реализация метода должна позволить нам вызвать эту функцию также без аргументов.

ПРИМЕЧАНИЕ Опущение двух первых объявлений в листинге 3.9 не изменит такое поведение программы. Эти строки просто помогают IDE предоставить лучшие варианты автоподстановки для функции `getProducts()`.

Поработайте с этим образцом кода в песочнице: http://mng.bz/lozj. Обратите внимание: сгенерированный JavaScript имеет всего одну функцию `getProducts()`, как показано на рис. 3.7.

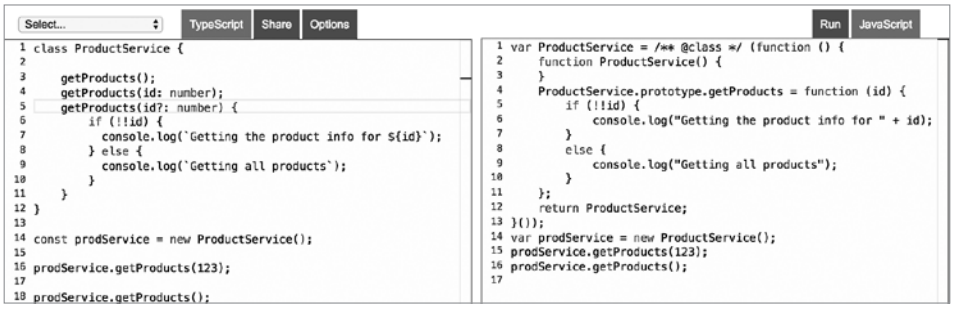


Рис. 3.7. Верный синтаксис для перегрузки метода

Сходным образом можно перегрузить сигнатуру метода, чтобы указать, что он может не только иметь различные аргументы, но и возвращать значения разных типов. Листинг 3.10 показывает сценарий с перегруженным методом `getProducts()`, который может быть вызван двумя способами:

- передачей `description` продукта и возвращением массива типа `Product`;
- передачей `id` продукта и возвращением одиночного объекта типа `Product`.

Посмотреть и выполнить приведенный образец кода можно здесь: http://mng.bz/BYov. Вывод, получаемый в консоли браузера, показан на рис. 3.8.

Давайте поэкспериментируем с кодом в листинге 3.10. Если вы закомментируете две строки, объявляющие сигнатуры `getProducts()`, программа по-прежнему будет работать. Вы можете вызывать этот метод, передавая в качестве аргумента либо число, либо строку, и этот метод по-прежнему будет возвращать либо один `Product`, либо их массив.

Вопрос в том, зачем вообще объявлять перегруженные сигнатуры, если вы можете просто реализовать один метод, используя объединения в типах аргументов и возвращаемых значениях? Перегрузка методов помогает компилятору TypeScript правильно отобразить переданные типы аргументов в типы возвращаемых значений. Когда перегруженные сигнатуры методов объявлены, статический анализатор верно предположит способы вызова перегруженного метода.

Листинг 3.10. Различные аргументы и возвращаемые типы

```
interface Product {                                    Определяет тип Product
    id: number;
    description: string;
}

class ProductService {

      getProducts(description: string): Product[];     Первая перегруженная сигнатура getProducts()
      getProducts(id: number): Product;                Вторая перегруженная сигнатура getProducts()
      getProducts(product: number | string): Product[] | Product{
          if (typeof product === "number") {
              console.log(`Getting the product info for id ${product}`);
                  return { id: product, description: 'great product' };
          } else if (typeof product === "string") {
            console.log(`Getting product with description ${product}`);
            return [{ id: 123, description: 'blue jeans' },
                    { id: 789, description: 'blue jeans' }];
          } else {
            return { id: -1,
                     description: 'Error: getProducts() accept only number or
➥ string as args' };
          }
     }
}

const prodService = new ProductService();

console.log(prodService.getProducts(123));

console.log(prodService.getProducts('blue jeans'));
```

Реализация getProducts()

Смотрит, был ли метод вызван с помощью description продукта

Смотрит, был ли метод вызван с помощью id продукта

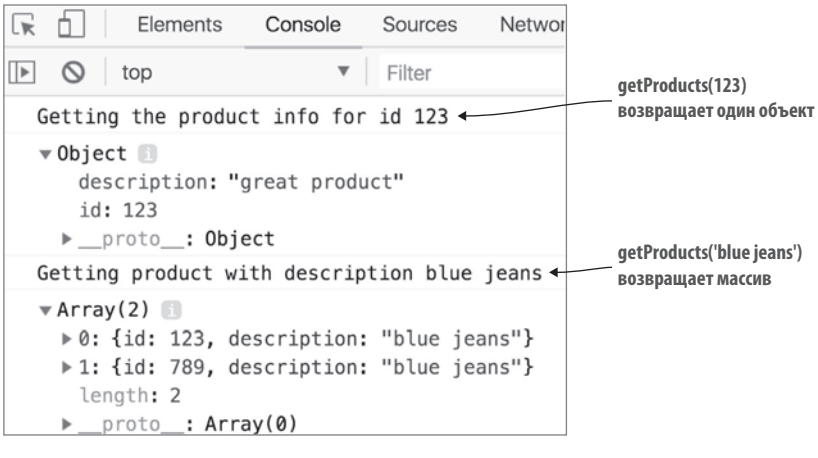


Рис. 3.8. Перегрузка с разными возвратами

На рис. 3.9 (скриншот из VS Code) показано первое предположение (обратите внимание на 1/2) для вызова метода `getProducts()`.

Рис. 3.9. Предположение первой сигнатуры метода

На рис. 3.10 показана вторая подсказка (обратите внимание на 2/2) для вызова метода `getProducts()` с другим параметром и возвращаемым типом.

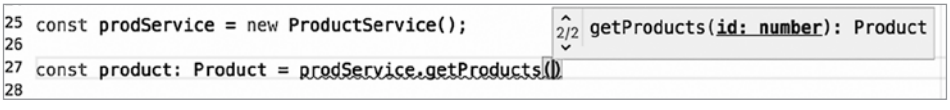


Рис. 3.10. Предположение второй сигнатуры метода

Если бы мы закомментировали объявления сигнатур `getProducts()`, сформировать эти предположения было бы не так просто. Стало бы сложно понять, какой тип аргумента приводит к возврату значения какого типа, как показано на рис. 3.11.

Рис. 3.11. Предположение без перегрузки

Вы можете поспорить, что выгоды, предлагаемые перегрузкой методов в TypeScript, не слишком убедительны, и мы согласимся, что может быть легче объявить два метода с разными именами, например `getProduct()` и `getProducts()`, не объединяя их аргументы и возвращаемые типы. Это верно для всех случаев, кроме одного: перегрузка конструкторов.

В классах TypeScript вы можете дать конструктору только одно имя — `constructor`. Если вы хотите создать класс с несколькими конструкторами, имеющими разную природу, то можете предпочесть использование синтаксиса для перегрузки, как показано в листинге 3.11.

ПРИМЕЧАНИЕ Так как в предыдущем листинге мы хотели допустить конструктор без аргументов, то сделали все аргументы в реализации конструктора опциональными.

Листинг 3.11. Перегрузка конструкторов

```
class Product {
    id: number;
    description: string;

    constructor();                                          // Объявление конструктора без аргументов
    constructor(id: number);                                // Объявление конструктора с одним аргументом
    constructor(id: number, description: string);           // Объявление конструктора с двумя аргументами
    constructor(id?: number, description?: string) {
        // Constructor implementation goes here )          // Реализация конструктора, обрабатывающего все возможные аргументы
    }
}
```

Но опять же, перегрузка конструкторов — это не единственный способ инициализации свойств объекта. Например, вы можете объявить один интерфейс для представления всех возможных параметров конструктора. Следующий листинг объявляет интерфейс со всеми опциональными свойствами и класс с одним конструктором, получающим один опциональный аргумент.

Листинг 3.12. Один конструктор с опциональным аргументом

```
interface ProductProperties {                               // Интерфейс ProductProperties с двумя опциональными свойствами
    id?: number;
    description?: string;
}

class Product {
  id: number;
  description: string;

  constructor(properties?: ProductProperties ) {            // Конструктор класса с опциональным аргументом типа ProductProperties
    // Constructor implementation goes here
  }
}
```

При перегрузке метода или конструктора в TypeScript прислушивайтесь к здравому смыслу. Несмотря на то что перегрузка обеспечивает несколько способов вызова метода, его логика может легко стать непонятной. В нашей современной работе в TypeScript мы редко пользуемся перегрузкой.

3.2. РАБОТА С ИНТЕРФЕЙСАМИ

В главе 2 мы использовали интерфейсы только для объявления пользовательских типов и в итоге выработали главное правило: «Если вам нужен пользовательский тип, включающий конструктор, используйте класс; в противном случае используйте интерфейс». В текущем разделе мы покажем вам, как использовать интерфейсы TypeScript, чтобы гарантировать реализацию классом конкретного API.

3.2.1. Обеспечение выполнения контракта

Интерфейс может объявлять не только свойства, но также и методы (однако без реализаций). Затем объявление класса может включать ключевое слово `implements`, сопровождаемое именем интерфейса. Другими словами, в то время как интерфейс содержит только сигнатуры методов, класс может содержать их реализации.

Предположим, у вас есть Toyota Camry. Она состоит из тысяч деталей, выполняющих различные действия, но как водителю вам требуется знать только набор управления — как заводить и глушить двигатель, как газовать и тормозить, как включать радио и т. д. Все это управление в совокупности считается *публичным интерфейсом*, предоставленным вам проектировщиками Toyota Camry.

Теперь представьте, что вы арендовали машину и взяли Ford Taurus, который ранее вам водить не приходилось. Будете ли вы знать, как им управлять? Да, потому что у него знакомый *интерфейс*: ключ зажигания, педали газа и тормоза и т. д. Когда вы арендуете машину, то можете даже запросить модель, имеющую конкретный интерфейс, например коробку-автомат.

Давайте смоделируем интерфейс автомобиля с помощью синтаксиса TypeScript. Следующий листинг показывает интерфейс `MotorVehicle`, объявляющий пять методов.

Листинг 3.13. Интерфейс MotorVehicle

```
interface MotorVehicle {
    startEngine(): boolean;
    stopEngine(): boolean;
    brake(): boolean;
    accelerate(speed: number): void;
    honk(howLong: number): void;
}
```

Объявляет сигнатуру метода, который должен быть реализован классом

Обратите внимание, что ни один из методов `MotorVehicle` не реализован. Теперь мы можем объявить класс `Car`, который будет реализовывать все методы, объявленные в интерфейсе `MotorVehicle`. С помощью ключевого слова `implements` мы объявляем, что класс реализует конкретный интерфейс:

```
class Car implements MotorVehicle {

}
```

Это простое объявление класса не скомпилируется. Вы получите ошибку: «Class Car incorrectly implements interface MotorVehicle» (Класс Car неверно реализует интерфейс MotorVehicle). Когда вы объявляете, что класс реализует некий интерфейс, вы должны реализовать каждый метод, объявленный в нем. Другими

словами, предыдущий фрагмент кода утверждает: «Я обещаю, что класс `Car` реализует API, объявленный в интерфейсе `MotorVehicle`». Следующий листинг показывает упрощенную реализацию этого интерфейса в классе `Car`.

Листинг 3.14. Класс, реализующий MotorVehicle

```
class Car implements MotorVehicle {
    startEngine(): boolean {
        return true;
    }
    stopEngine(): boolean{
        return true;
    }
    brake(): boolean {
        return true;
    }
    accelerate(speed: number): void;
        console.log(`Driving faster`);
    }
    honk(howLong: number): void {
        console.log(`Beep beep yeah!`);
    }
}

const car = new Car();
car.startEngine();
```

Реализует методы из интерфейса

Инстанцирует класс Car

Использует API Car, чтобы запустить двигатель

Обратите внимание, что мы не объявляли явно тип константы `car` — это пример вывода типа. Явно мы могли бы объявить тип `car` так (хотя это не обязательно):

```
const car: Car = new Car();
```

Мы также могли бы объявить для константы `car` тип `MotorVehicle`, поскольку наш класс `car` реализует этот пользовательский тип:

```
const car: MotorVehicle = new Car();
```

В чем разница между этими двумя объявлениями константы `car`? Предположим, что класс `Car` реализует восемь методов: пять из них берутся из интерфейса `MotorVehicle`, а остальные просто произвольны. Если константа `car` имеет тип `Car`, вы можете вызвать все восемь методов для экземпляра объекта, представленного `car`. Но если `car` имеет тип `MotorVehicle`, то с помощью константы `car` возможно вызвать только пять методов, объявленных в этом интерфейсе.

Мы можем сказать, что *интерфейс обеспечивает выполнение определенного контракта*. В нашем примере это означает, что мы вынуждаем класс `Car` реализовывать каждый из пяти методов, объявленных в интерфейсе `MotorVehicle`, в противном случае код не скомпилируется.

Теперь спроектируем интерфейс для машины Джеймса Бонда. Все верно, для агента 007. Эта особая машина должна уметь летать, а также плавать. Для начала объявим пару интерфейсов.

Листинг 3.15. Интерфейсы Flyable и Swimmable

```
interface Flyable {
    fly(howHigh: number);
    land();
}

interface Swimmable {
    swim(howFar: number);
}
```

Класс может реализовывать более одного интерфейса, поэтому давайте убедимся, что наш класс реализует эти два.

Листинг 3.16. Машина с тремя интерфейсами

```
class Car implements MotorVehicle, Flyable, Swimmable {
    // Реализуйте здесь все методы из трех интерфейсов.
}
```

Делать каждую машину летающей и плавающей — не очень хорошая идея, поэтому давайте не будем модифицировать класс `Car` из листинга 3.14. Вместо этого используем наследование класса и создадим класс `SecretServiceCar`, расширяющий `Car` и добавляющий больше возможностей.

Листинг 3.17. Класс, расширяющий и реализующий

```
class SecretServiceCar extends Car implements Flyable, Swimmable {
// Реализуйте здесь все методы из двух интерфейсов.
}
```

Реализуя все методы, объявленные в `Flyable` и `Swimmable`, наш класс `SecretServiceCar` превращает обычный автомобиль в летающе-плавающий объект. Класс `Car` при этом продолжает представлять обычный авто с функциональностью, объявленной в интерфейсе `MotorVehicle`.

3.2.2. Расширение интерфейсов

Как вы видели в предыдущем разделе, совмещение классов и интерфейсов привносит гибкость в проектирование кода. Давайте рассмотрим еще одну опцию — расширение интерфейсов.

В предыдущем примере, когда возник запрос разработать машину для секретной службы, у нас уже был интерфейс `MotorVehicle` и класс `Car`, который этот

интерфейс реализовывал. В листинге 3.17 класс `SecretServiceCar` унаследовал от `Car` и реализовал два дополнительных интерфейса.

Но когда вы проектируете машину для секретной службы, то захотите реализовать все методы в интерфейсе `MotorVehicle` по-разному. В таком случае можно объявить класс `SecretServiceCar` так:

Листинг 3.18. Класс, реализующий три интерфейса

```
class SecretServiceCar implements MotorVehicle, Flyable, Swimmable {
    // Реализуйте здесь все методы из трех интерфейсов.

}
```

С другой стороны, наш летающий объект также является автомобилем, поэтому мы можем объявить интерфейс `Flyable` следующим образом.

Листинг 3.19. Расширение интерфейса

```
interface Flyable extends MotorVehicle{
    fly(howHigh: number);
    land();
}
```

Один интерфейс расширяет другой

Объявляет сигнатуру метода для реализации в классе

Теперь если класс включит в свое объявление `implements Flyable`, то должен будет реализовать пять методов, объявленных в интерфейсе `MotorVehicle` (см. листинг 3.13) наряду с двумя методами из `Flyable` (листинг 3.19), — итого семь методов. Наш класс `SecretServiceCar` должен реализовать эти семь методов плюс один из `Swimmable`.

Листинг 3.20. Класс, реализующий Flyable и Swimmable

```
class SecretServiceCar implements Flyable, Swimmable {

    startEngine(): boolean {
        return true;
    };
    stopEngine(): boolean{
        return true;
    };
    brake(): boolean {
        return true;
    };
    accelerate(speed: number) {
        console.log(`Driving faster`);
    }

    honk(howLong: number): void {
        console.log(`Beep beep yeah!`);
    }
```

Реализует метод из MotorVehicle

```
    fly(howHigh: number) {
        console.log(`Flying ${howHigh} feet high`);
    }

    land() { 2((CO16-7))
        console.log(`Landing. Fasten your belts.`);
    }

    swim(howFar: number) {
        console.log(`Swimming ${howFar} feet`);
    }
}
```

Реализует метод из Flyable

Реализует метод из Swimmable

ПРИМЕЧАНИЕ Даже если `Swimmable` также будет расширять `MotorVehicle`, компилятор TypeScript не станет ругаться.

Объявление интерфейсов, включающее сигнатуры методов, улучшает читаемость кода, потому что любой интерфейс описывает грамотно определенные возможности, которые могут быть реализованы одним или несколькими конкретными классами.

Тем не менее интерфейс не сообщит точно, как именно класс его реализует. Разработчики, практикующие объектно-ориентированный подход, используют такую мантру: «Программируйте через интерфейсы, а не через реализации». В следующем разделе вы узнаете, что это значит.

3.2.3. Программирование через интерфейсы

Чтобы понять значение и преимущества программирования через интерфейсы, рассмотрим случай, в котором эта техника не использовалась. Представьте, что вам нужно написать код, который будет извлекать информацию о конкретном продукте (или обо всех продуктах) из источника данных.

Вы знаете, как писать классы, поэтому можете сразу начать их реализовывать. Можете определить пользовательский тип `Product` и класс `ProductService` с двумя методами.

Листинг 3.21. Программирование через реализации

```
class Product {
    id: number;
    description: string;
}

class ProductService {
```

Пользовательский тип Product

Конкретная реализация ProductService

```
    getProducts(): Product[]{ ◄── Реализованный метод
      // Здесь должен находиться код для получения информации
      // о продуктах из реального источника данных.

      return [];
}

getProductById(id: number): Product { 3((CO17-4))
      // Здесь должен находиться код для получения информации
      // о продуктах из реального источника данных.

      return { id: 123, description: 'Good product' };
   }
}
```

Затем в нескольких точках вашего приложения вы можете инстанцировать `ProductService` и использовать его методы:

```
const productService = new ProductService(); const products = productService.
getProducts();
```

Это было легко, не так ли? Вы гордо коммитите этот код в репозиторий, но ваш менеджер говорит, что ребята из бэкенда откладывают реализацию сервера, который должен был предоставлять данные для `ProductService`. Он просит вас создать другой класс, `MockProductService`, *с тем же* API, который сможет возвращать жестко закодированные данные продукта.

Без проблем. Вы пишете другую реализацию сервиса продуктов.

Листинг 3.22. Другая реализация сервиса продуктов

```
class MockProductService {  ◄── Конкретная реализация MockProductService

  getProducts(): Product[]{                                    ◄┐
    // Сюда размещается код для                                  │
    // получения жестко закодированных данных продукта.          │  Реализованный
                                                                 │  метод
    return [];                                                   │
}                                                                │

  getProductById(id: number): Product {                        ◄┘

    return { id: 456, description: 'Not a real product' };
  }
}
```

ПРИМЕЧАНИЕ Вам может понадобиться создать `MockProductService` не только потому, что ребята из бэкенда запаздывают, но также и для его использования в тестировании модулей, в котором вы не задействуете реальные сервисы.

Вы создали две конкретные реализации сервиса продуктов. Надеемся, что вы не допустили ошибок в процессе объявления методов в `MockProductService`. Эти методы должны полностью совпадать с их прообразами в `ProductService`, иначе вы рискуете нарушить код, использующий `MockServiceProduct`.

Нам не нравится слово «надеемся» в предыдущем абзаце. Мы уже знаем, что интерфейс позволяет обеспечить выполнение контракта классом — в данном случае классом `MockProductService`. Но интерфейсы мы здесь не объявляли.

Звучит странно, но в TypeScript вы можете объявить класс, реализующий другой класс. Более грамотным (но не самым лучшим) подходом будет начать прописывать `MockProductService` следующим образом:

```
class MockProductService implements ProductService {
    // Здесь идет реализация.
}
```

TypeScript достаточно сообразителен, чтобы понять, что если вы используете имя класса после слова `implement`, то хотите использовать его как интерфейс и обеспечить выполнение реализации всех публичных методов `ProductService`. В этом случае вы точно не забудете реализовать что-либо и точно не допустите ошибки в сигнатуре `getProducts()` или `getProductById()`. Код не скомпилируется до тех пор, пока вы должным образом не реализуете эти методы в классе `MockProductServcie`.

Однако наилучшим подходом будет сразу начать программировать через интерфейсы. Когда вы получите требование написать `ProductService` с двумя методами, то должны начать с объявления интерфейса с этими методами, не озадачиваясь их реализацией.

Давайте назовем этот интерфейс `IProductService` и объявим в нем две сигнатуры метода. После этого объявим класс `ProductService`, реализующий этот интерфейс.

Листинг 3.23. Программирование через интерфейс

```
interface Product {  <-- Объявляет пользовательский тип с помощью интерфейса
    id: number;
    description: string;
}

interface IProductService {  <-- Объявляет API как интерфейс
    getProducts(): Product[];
    getProductById(id: number): Product
}

class ProductService implements IProductService {  <-- Реализует интерфейс

    getProducts(): Product[]{
```

```
        // здесь идет код для получения продукта
        // из реального источника данных.

        return [];
    }

    getProductById(id: number): Product {
        // здесь идет код для получения продукта по id.
        return { id: 123, description: 'Good product' };
    }
}
```

Объявление API как интерфейса показывает, что вы потратили время на размышления о необходимой функциональности и только затем позаботились о конкретной реализации. Теперь, если потребуется реализовать новый класс, например `MockProductService`, то вы начнете так:

```
class MockProductService implements IProductService {
    // Здесь идет другая конкретная реализация методов интерфейса.
}
```

Вы заметили, что пользовательский тип `Product` по-разному реализован в листингах 3.21 и 3.23? Используйте ключевое слово `interface` вместо `class`, если не нуждаетесь в инстанцировании этого пользовательского типа (например, `Product`), и отпечаток JavaScript будет меньше. Опробуйте код из этих двух листингов в песочнице и сравните сгенерированный JavaScript. Версия, где `Product` является интерфейсом, окажется короче.

> **СОВЕТ** Мы назвали интерфейс `IProductService`, начав с заглавной `I`, в то время как имя класса было `ProductService`. Некоторые предпочитают использовать суффикс `impl` для конкретных реализаций (например, `ProductServiceimpl`) и называть интерфейс просто `ProductService`.

Еще один хороший пример программирования через интерфейсы — это фабричные функции, которые реализуют логику бизнеса, а затем возвращают нужный экземпляр объекта. Если нам понадобится написать фабричную функцию, возвращающую `ProductService` или `MockProductService`, мы используем интерфейс в качестве возвращаемого типа.

Листинг 3.24. Фабричная функция

```
function getProductService(isProduction: boolean): IProductService {
    if (isProduction) {
        return new ProductService();
    } else {
        return new MockProductService();
    }
}
```

Фабричная функция, использующая в качестве возвращаемого типа интерфейс

```
const productService: IProductService;   <-- Константа, имеющая тип интерфейса

...

const isProd = true;   <-- В реальном приложении это бы не кодировалось жестко
const productService: IProductService = getProductService(isProd);
                                                  Получает правильный
                                                  экземпляр сервиса продукта
const products = productService.getProducts();
                                  Вызывает метод для
                                  сервиса продукта
```

В этом примере мы использовали константу `isProd` с жестко закодированным значением `true`. В реальных же приложениях это значение было бы получено из файла свойств или переменной среды.

Изменяя это свойство с `true` на `false`, можно изменить поведение приложения при выполнении.

Несмотря на то что фактический тип возвращаемого объекта будет либо `ProductService`, либо `MockProductService`, в сигнатуре функции мы использовали абстракцию `IProductService`. Это делает нашу фабричную функцию более гибкой и расширяемой: если в будущем нам понадобится изменить тело этой функции для возвращения объекта типа `AnotherProductService`, то мы можем просто убедиться, что это новое объявление класса включает в себя `implements IProductService`, и код, использующий нашу фабричную функцию, скомпилируется без дополнительных изменений. Программируйте через интерфейсы!

> **СОВЕТ** В главе 16 есть врезка «Снова о программировании через интерфейсы», в которой вы увидите другой случай использования, где программирование через интерфейсы может предотвратить ошибки среды выполнения.

> **ПРИМЕЧАНИЕ** В разделе 11.5 наряду с объяснением специфики использования внедрения зависимостей с помощью фреймворка Angular мы вернемся к идее программирования через абстракции. Там вы увидите, что интерфейсы TypeScript не могут быть использованы, но вместо них вы можете прибегнуть к абстрактным классам.

ИТОГИ

- Вы можете создавать класс, взяв за основу другой. Мы называем это *наследованием класса*.
- Подкласс может использовать `public`- или `protected`-свойства суперкласса.
- Если свойство класса объявлено как `private`, то может быть использовано только внутри этого класса.

- С помощью `private`-конструктора вам доступно создание класса, который может быть инстанцирован только один раз.
- Если в суперклассе и подклассе существуют методы с одинаковыми сигнатурами, мы называем это *переопределением методов*. Конструкторы классов тоже могут быть переопределены. Ключевое слово `super` и метод `super()` позволяют подклассу вызывать членов суперкласса.
- Вы можете объявить для метода несколько сигнатур, что известно как *перегрузка метода*.
- Интерфейсы могут включать сигнатуры методов, но не могут содержать их реализации.
- Вы можете выполнить наследование одного интерфейса от другого.
- В процессе реализации класса проверьте, могут ли какие-то из методов быть объявлены в отдельном интерфейсе. В таком случае ваш класс будет вынужден реализовать этот интерфейс. Такой подход обеспечивает чистый способ отделить объявление функциональности от реализации.

4 Перечисления и обобщенные типы

В этой главе:

- ✓ Преимущества перечислений.
- ✓ Синтаксис для численных и строчных перечислений.
- ✓ Назначение обобщенных типов.
- ✓ Как писать классы, интерфейсы и функции, поддерживающие обобщенные типы.

В главе 2 мы познакомили вас с объединениями, позволяющими создавать пользовательский тип, совмещая несколько уже существующих. В этой главе вы научитесь использовать `enums` (перечисления) — способ создания нового типа, основанного на ограниченном наборе значений.

Также рассмотрим обобщенные типы, которые позволяют устанавливать ограничения типа для членов класса, параметров функций или их возвращаемых типов.

4.1. ИСПОЛЬЗОВАНИЕ ENUMS

Перечисления позволяют создавать ограниченные наборы именованных констант, имеющих что-то общее. Такие константы могут быть числами или строками.

4.1.1. Численные значения enums

В неделе семь дней, и для их представления можно назначать числа от 1 до 7. Но какой из дней в неделе первый?

Согласно стандарту элементов данных и форматов обмена ISO 8601, первым днем недели считается понедельник, что не мешает таким странам, как США, Канада и Австралия, первым считать воскресенье. А что если кто-то присвоит переменной, хранящей этот день, число 8? Мы не хотим, чтобы такое случилось, и используем вместо чисел имена, делая код более читаемым. Использование только чисел для представления дней может оказаться не лучшей идеей. С другой стороны, их использование для хранения дней более эффективно, чем использование имен. Нам же одновременно нужны читаемость, способность ограничивать значения до заданного набора и эффективность хранения данных. Именно тут и помогают перечисления.

В TypeScript есть ключевое слово `enum`, которое может определять ограниченный набор констант. Мы можем объявить новый тип `Weekdays` (дни недели) следующим образом:

Листинг 4.1. Определение Weekdays с помощью enum

```
enum Weekdays { Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
    Sunday = 7
}
```

Этот код определяет новый тип — `Weekdays`, который имеет ограниченный набор значений. Мы инициализировали каждый член перечисления с численным значением, и к дням недели можно обращаться, используя точечную нотацию:

```
let dayOff = Weekdays.Tuesday;
```

Значение переменной `dayOff` равно 2, но если вы напечатаете предыдущую строку в своей IDE или в песочнице TS, то получите подсказки с возможными значениями, как показано на рис. 4.1.

Использование членов перечисления `Weekdays` не дает допустить ошибку и присвоить переменной `dayOff` неверное значение (например, 8). Конечно, ничто не мешает вам проигнорировать это `enum` и написать `dayOff = 8`, но это уже будет серьезный проступок.

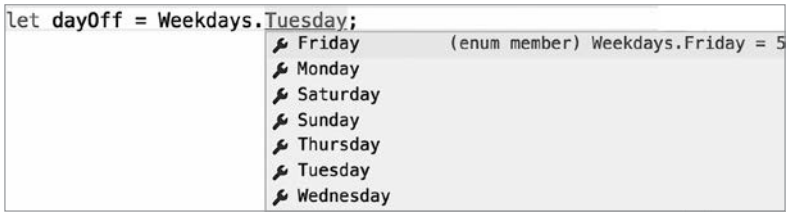


Рис. 4.1. Автоподстановка с enums

В листинге 4.1 мы могли бы инициализировать с 1 только понедельник, а остальные значения дней были бы присвоены с использованием автоматического увеличения: вторник был бы инициализирован с 2, среда с 3 и т. д.

Листинг 4.2. enum с автоматически увеличиваемыми значениями

```
enum Weekdays {
    Monday = 1,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
```

По умолчанию перечисления начинаются с 0. Поэтому если мы не инициализируем `Monday` с 1, то его значение будет 0.

ОБРАТНОЕ ОТОБРАЖЕНИЕ ЧИСЛЕННЫХ ENUMS

Если вы знаете значение численного `enum`, то можете найти имя члена этого `enum`. Например, есть функция, возвращающая число дня недели, и вы бы хотели вывести в консоль его имя. Используя это значение в роли индекса, вы можете извлечь имя этого дня.

```
enum Weekdays {  <-- Объявление численных enum
    Monday = 1,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}

console.log(Weekdays[3]);  <-- Получает имя члена, соответствующего 3
```

В последней строке этого кода мы извлекли имя дня 3 и в консоли увидим `Wednsday`.

В некоторых случаях вас не будет интересовать, какие численные значения присвоены членам `enum`, — следующая функция `convertTempreture()` иллюстрирует это. Она преобразует температуру из градусов по Фаренгейту в градусы по Цельсию или наоборот. В этой версии `convertTemperature()` мы не будем использовать перечисления, но после мы перепишем ее с их участием.

Листинг 4.3. Преобразование температуры без enums

```
function convertTemperature(temp: number, fromTo: string): number {

    return ('FtoC' === fromTo) ?
           (temp - 32) * 5.0/9.0:
            temp * 9.0 / 5.0 + 32;
}

console.log(`70F is ${convertTemperature(70, 'FtoC')}C`);
console.log(`21C is ${convertTemperature(21, 'CtoF')}F`);
console.log(`35C is ${convertTemperature(35, 'ABCD')}F`);
```

Эта функция получает два параметра: температура и направление преобразования

Преобразует градусы из шкалы Фаренгейта в Цельсия

Преобразует из шкалы Цельсия в Фаренгейт

Преобразует 70 градусов по Фаренгейту

Преобразует 21 градус по Цельсию

Вызывает функцию с бессмысленным параметром fromTo

Функция в листинге 4.3 преобразует значение из шкалы Цельсия в Фаренгейт, если вы передаете для `fromTo` любое значение, кроме `FtoC`. В последней строке мы намеренно передали ошибочное значение `ABCD` в качестве параметра `fromTo`, и функция по-прежнему преобразовала температуру из шкалы Цельсия в Фаренгейт. Посмотрите на этот пример в действии на CodePen: http://mng.bz/JzaK. Попытки вызвать функцию с ошибочными значениями должны быть перехвачены компилятором, и именно для этого используются перечисления в TypeScript.

В следующем листинге мы объявили `enum Direction`, ограничивающее допустимые константы до значений `FtoC` или `CtoF`. Мы также поменяли тип параметра `fromTo` со `string` на `Direction`.

Листинг 4.4. Преобразование температуры с помощью enums

```
enum Direction {
    FtoC,
    CtoF
}

function convertTemperature(temp: number, fromTo: Direction): number {

        return (Direction.FtoC === fromTo) ?
               (temp - 32) * 5.0/9.0:
                temp * 9.0 / 5.0 + 32;
}
```

Объявляет Direction enum

Тип второго параметра — Direction

```
console.log(`70F is ${convertTemperature(70, Direction.FtoC)}C`);
console.log(`21C is ${convertTemperature(21, Direction.CtoF)}F`);
```

Вызывает функцию, используя члены enum

Поскольку тип второго параметра функции — это `Direction`, мы должны вызвать эту функцию, передав одним из членов перечисления, например `Direction.CtoF`. Нас не интересует численное значение этого члена. Цель этого перечисления — просто предоставить ограниченный набор констант: `CtoF` и `FtoC`. IDE предложит вам два возможных значения для второго параметра, оградив от возможности опечатки.

> **СОВЕТ** Использование типа `Direction` для второго аргумента не предотвращает другое нарушение — вызов этой функции как `convertTemperature(50.0, 99)`.

Члены перечисления инициализированы со значениями (явными или неявными). Во всех примерах этого раздела члены перечислений были инициализированы с числами, но TypeScript позволяет создавать `enums` со строковыми значениями, и мы это увидим в следующих примерах.

4.1.2. Строковые перечисления

В некоторых случаях может понадобиться объявить ограниченный набор строчных констант, для чего удобно использовать строчные `enums` — перечисления, чьи члены инициализированы со строчными значениями. Предположим, вы программируете компьютерную игру, где игрок может перемещаться в четырех направлениях.

Листинг 4.5. Объявление строкового перечисления

```
enum Direction {
    Up = "UP",
    Down = "DOWN",
    Left = "LEFT",
    Right = "RIGHT",
}
```

Инициализирует член enum со строковым значением

Когда вы объявляете строковое перечисление, то обязаны инициализировать каждый его член. Возникает вопрос: «Почему бы просто не использовать численное `enum`, чтобы TypeScript смог автоматически инициализировать его членов с любыми числами?» Причина в том, что в некоторых случаях нужно присвоить членам перечисления информативные значения. Например, если потребуется произвести отладку программы, то вы уже увидите, что последнее движение было не `0`, а `UP`.

Вы можете задать и такой вопрос: «Зачем объявлять перечисление `Direction`, если я могу просто объявить четыре строчные константы со значениями `Up`, `DOWN`, `LEFT`, `RIGHT`?» Сделать так можно, но давайте предположим, что у нас есть функция со следующей сигнатурой:

```
move(where: string)
```

Разработчик может допустить ошибку (или опечатку) и вызвать эту функцию как `move("North")`. Но `North` не является допустимым направлением, поэтому безопаснее объявить эту функцию, используя перечисление `Direction`:

```
move(where: Direction)
```

Мы допустили ошибку и в строке 15 передали `"North"` (рис. 4.2). Ошибка при компиляции будет гласить: «Argument of type “North” is not assignable to parameter of type ‘Direction’» (Аргумент типа “North” не может быть присвоен параметру типа ‘Direction’). В строке 18, использующей перечисление `Direction`, IDE предлагает выбор действительных членов перечисления, поэтому передать ошибочный аргумент вы не сможете.

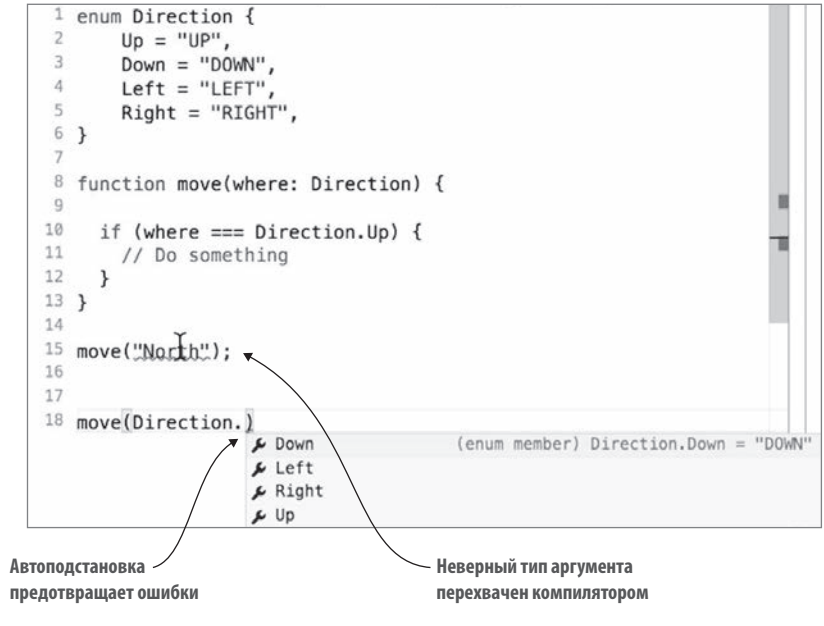


Рис. 4.2. Перехват ошибочных вызовов функции

В качестве альтернативы перечислениям вы можете использовать объединенный тип. Например, сигнатура функции `move()` может выглядеть так:

```
function move(direction: 'Up' | 'Down' | 'Left' | 'Right') { }
move('North'); // ошибка компиляции
```

Еще одной альтернативой может послужить определение пользовательского типа:

```
type Direction = 'Up' | 'Down' | 'Left' | 'Right';
function move(direction: Direction) {}
move('North'); // ошибка компиляции
```

Теперь представим, что нужно отслеживать изменение состояния приложения. Пользователь может инициировать ограниченное число действий в каждом из представлений приложения, а вы хотите регистрировать действия, производимые в представлении Товаров (Products). Первым делом приложение старается загрузить товары, и это действие может закончиться либо успехом, либо провалом. Пользователь также может воспользоваться поиском товара. Для отображения состояний представления товаров вы можете объявить строчное перечисление.

Листинг 4.6. Объявление строчного enum для мониторинга действий

```
enum ProductsActionTypes {
    Search = 'Products Search',        <-- Инициализирует член Search
    Load = 'Products Load All',        <-- Инициализирует член Load
    LoadFailure = 'Products Load All Failure',   <-- Инициализирует член LoadFailure
    LoadSuccess = 'Products Load All Success'    <-- Инициализирует член LoadSuccess
}

// Если функция, загружающая товары,
// проваливается...
console.log(ProductsActionTypes.LoadFailure);   <-- Выводит в консоль «Products Load All Failure»
```

Когда пользователь щелкает по кнопке загрузки товаров, вы можете зарегистрировать значение члена ProductsActionTypes.Load, которое выведет текст "Products Load All". Если товары не были успешно загружены, вы можете зарегистрировать значение productsActionTypes.LoadFailure, которое выведет текст "Products Load All Failure".

> **ПРИМЕЧАНИЕ** Некоторые управляющие состоянием фреймворки требуют, чтобы приложение отправляло действия, когда состояние приложения должно измениться. Мы можем объявить строчное перечисление, как в листинге 4.6, и отправлять действия ProductsActionTypes.Load, ProductsActionTypes.LoadSuccess и т. д.

Строчные перечисления могут быть легко отображены в строчные значения, поступающие с сервера или из базы данных (статус заказа, тип учетной записи и т. д.), без написания дополнительного кода, обеспечивая вас всеми преимуществами строгой типизации. Мы покажем это в листинге 4.17 в конце главы.

ПРИМЕЧАНИЕ В отличие от численных, строчные перечисления не допускают обратного отображения — вы не можете найти имя члена по его значению.

4.1.3. Использование перечислений const

Если вы используете ключевое слово `const` при объявлении `enum`, то его значения будут встроены и код JavaScript сгенерирован не будет.

Давайте сравним сгенерированный JavaScript `enum` и `const` enum. В левой части рис. 4.3 показано `enum`, объявленное без `const`, а в правой — итоговый JavaScript. Для наглядности в последней строке мы указали следующее направление движения.

```
enum Direction {
    Up = "UP",
    Down = "DOWN",
    Left = "LEFT",
    Right = "RIGHT",
}

const theNextMove = Direction.Down;
```

```
var Direction;
(function (Direction) {
    Direction["Up"] = "UP";
    Direction["Down"] = "DOWN";
    Direction["Left"] = "LEFT";
    Direction["Right"] = "RIGHT";
})(Direction || (Direction = {}));
var theNextMove = Direction.Down;
```

Рис. 4.3. enum без ключевого слова const

Теперь на рис. 4.4 мы добавим перед `enum` ключевое слово `const` и сравним сгенерированный JavaScript (справа) с тем, что получился на рис. 4.3.

```
const enum Direction {
    Up = "UP",
    Down = "DOWN",
    Left = "LEFT",
    Right = "RIGHT",
}

const theNextMove = Direction.Down;
```

```
var theNextMove = "DOWN" /* Down */;

```

Рис. 4.4. enum с ключевым словом const

Как вы видите, на рис. 4.4 для `enum Direction` код JavaScript сгенерирован не был. Однако значение члена перечисления из TypeScript кода (`Direction.Down`) было встроено в JavaScript.

ПРИМЕЧАНИЕ Врезка «Обратное отображение численных enums» содержит листинг, где мы развернули третий член перечисления: `Weekdays[3]`. Такое невозможно для const `enum`, так как они не представлены в сгенерированном JavaScript-коде.

Использование `const` с `enum` выражается в более лаконичном JavaScript, но имейте в виду, что так как ваше `enum` не представляется в JavaScript-коде, вы можете столкнуться с некоторыми ограничениями. Например, не сможете извлечь имя численного члена перечисления через его значение.

В целом же использование перечислений повышает читаемость ваших программ.

4.2. ИСПОЛЬЗОВАНИЕ ОБОБЩЕНИЙ

Известно, что в TypeScript есть встроенные типы и что вы также можете создавать пользовательские. Однако это еще не все. Звучит странно, но типы могут быть *параметризованы*, то есть в качестве параметра можно передавать тип (а не значение).

Легко объявить функцию, получающую параметры конкретных типов вроде числа или строки:

```
function calctTax(income: number, state: string){...}
```

Но *обобщения* TypeScript позволяют писать функции, которые могут работать с разными типами. Другими словами, вы можете объявить функцию, работающую как обобщенный тип, а конкретный тип может быть указан позднее вызывающим эту функцию компонентом.

В TS можно писать обобщенные функции, классы или интерфейсы. Обобщенный тип может быть представлен произвольной буквой (или буквами), например `T` в `Array<T>`, и при объявлении конкретного массива вы передадите в угловые скобки нужный тип, к примеру `number`:

```
let lotteryNumbers: Array<numbe>;
```

В этом разделе вы научитесь использовать обобщенный код, написанный кем-то другим, а также узнаете, как создавать собственные классы, интерфейсы и функции, которые могут работать с обобщенными типами.

4.2.1. Разъяснение обобщений

Обобщение — это часть кода, способная обрабатывать значения нескольких типов, которые указываются при использовании этого кода (в процессе вызова функции или инстанцирования класса).

Давайте рассмотрим массивы TypeScript, которые могут быть объявлены так:

- указанием типа элемента массива, сопровождаемого `[]`:

```
const someValues: number[];
```

- использованием обобщенного `Array`, сопровождаемого параметром типа в угловых скобках:

```
const someValues: Array<number>;
```

Если все элементы массива имеют один тип, то эти объявления будут эквивалентны друг другу, но первый вариант синтаксиса легче читать. Во втором варианте синтаксиса угловые скобки представляют параметр типа. Можно инстанцировать этот массив, как и любой другой, ограничив при этом его тип допустимыми значениями (в нашем примере `number`). Несколько позже мы покажем вам, когда объявление массива с обобщенным типом оказывается лучшим подходом.

Следующий фрагмент кода создает массив, который изначально будет содержать десять объектов типа `Person`, а выведенным типом переменной `people` будет `Person[]`.

```
class Person{ }

const people = new Array<Person>(10);
```

Массивы TypeScript могут содержать объекты любого типа, но если вы решите использовать обобщенный тип `Array`, то должны указать, какие типы значений допускаются для массива, например `Array<Person>`. Сделав это, вы устанавливаете ограничение на этот экземпляр массива. Если вы попытаетесь добавить в этот массив объект иного типа, компилятор TS укажет на ошибку. В другой части кода можно использовать массив с другим параметром типа, например `Array<Customer>`.

Следующий листинг объявляет класс `Person`, его потомка `Employee` и класс `Animal`. Затем инстанцирует каждый класс и пытается хранить все эти объекты в массиве `workers`, используя нотацию обобщенного массива с параметром типа `Array<Person>`.

Листинг 4.7. Использование обобщенного типа

```
class Person {                                   ← Объявляет класс Person
    name: string;
}

class Employee extends Person {                  ← Объявляет подкласс для Person
    department: number;
}

class Animal {                                   ← Объявляет класс Animal
    breed: string;
}

const workers: Array<Person> = [];               ← Объявляет и инициализирует обобщенный массив с конкретным параметром

workers[0] = new Person();
workers[1] = new Employee();                     ← Добавляет в массив объект
workers[2] = new Animal(); // Ошибка при компиляции
```

В листинге 4.7 последняя строка не скомпилируется, потому что массив `workers` был объявлен с параметром типа `Person`, а `Animal` не является `Person`. Но класс `Employee` расширяет `Person` и считается *подтипом* `Person`, поэтому вы можете использовать подтип `Employee` везде, где допустим супертип `Person`.

Итак, используя обобщенный массив `workers` с параметром `<Person>`, мы заявляем о своем намерении хранить в нем только экземпляры класса `Person` или объектов совместимых с ним типов. Попытка сохранить экземпляр класса `Animal` (как он был определен в листинге 4.7) в этом же массиве приведет к следующей ошибке компиляции: «Type `Animal` is not assignable to type `Person`. Property name is missing in type `Animal`» (Тип `Animal` не может быть присвоен типу `Person`. В типе `Animal` потеряно свойство `name`). Другими словами, использование обобщений в TypeScript помогает избегать ошибок, связанных с использованием неверных типов.

ВАРИАНТНОСТЬ ОБОБЩЕНИЙ

Выражение «*вариантность обобщений*» относится к правилам использования подтипов и супертипов в любом конкретном месте программы. Например, в Java массивы *ковариантны*, то есть вы можете использовать `Employee[]` (подтип) там, где допустим массив `Person[]` (супертип).

Так как TypeScript поддерживает структурную типизацию, то там, где ожидается тип `Person`, вы можете использовать либо `Employee`, либо любой другой объектный литерал, совместимый с типом `Person`. Другими словами, вариантность обобщений применяется к объектам, которые структурно одинаковы. Учитывая важность анонимных типов в JavaScript, понимание этого необходимо для оптимального использования обобщений в TypeScript.

Чтобы увидеть, может ли тип A быть использован там, где ожидается тип B, прочитайте о структурной подтипизации в документации к TypeScript: http://mng.bz/wla2.

СОВЕТ Для объявления идентификатора `workers` в листинге 4.7 мы использовали `const` (а не `let`), потому что его значение никогда не меняется. Добавление новых объектов в массив `workers` не меняет адрес массива в памяти, поэтому значение идентификатора `workers` остается одним и тем же (то есть неизменным).

Если вы знакомы с обобщениями в Java или C#, то обобщения TypeScript могут тоже показаться знакомыми. Однако здесь есть один важный нюанс. В то время как Java и C# используют *номинальную* систему типов, TypeScript использует *структурную*, о чем мы говорили в разделе 2.2.4.

В номинальной системе типов типы проверяются по их именам, но в структурной определение происходит согласно их структуре. В языках, использующих номинальную систему типов, следующая строка *всегда* будет вызывать ошибку:

```
let person: Person = new Animal();
```

В структурной же системе типов вы можете присваивать объект одного типа переменной другого типа до тех пор, пока структуры этих типов будут одинаковы. Давайте добавим свойство `name` в класс `Animal`.

Листинг 4.8. Обобщения и структурная система типов

```
class Person {
    name: string;
}

class Employee extends Person {
    department: number;
}

class Animal {
    name: string;   <-- Единственная дополнительная строка по сравнению с листингом 4.7
    breed: string;
}

const workers: Array<Person> = [];

workers[0] = new Person();
workers[1] = new Employee();
workers[2] = new Animal(); // никаких ошибок
```

Теперь компилятор TS не ругается на присваивание объекта `Animal` переменной типа `Person`. Переменная типа `Person` ожидает объект со свойством `name`, а у объекта `Animal` как раз есть такое свойство. Здесь не то чтобы `Person` и `Animal` представляют одинаковые типы, но их типы совместимы.

Более того, вам даже не нужно создавать новый экземпляр класса `Person`, `Employee` или `Animal` — вместо этого вы можете использовать синтаксис объектных литералов. Добавление следующей строки в листинг 4.8 абсолютно нормально, поскольку структура объектного литерала совместима со структурой типа `Person`:

```
workers[3] = { name: "Mary" };
```

С другой стороны, попытка присвоить объект `Person` переменной типа `Animal` приведет к ошибке компиляции:

```
const worker: Animal = new Person(); // ошибка компиляции
```

Сообщение об ошибке будет гласить: «Property breed is missing in type Person» (Свойство breed отсутствует в типе Person), и в этом есть смысл, потому что если вы объявите переменную `worker` типа `Animal`, но создадите экземпляр объекта `Person`, не имеющий свойства `breed`, то не сможете написать `worker.breed`. Отсюда и ошибка компиляции.

> **ПРИМЕЧАНИЕ** Предыдущее пояснение может раздражать бывалых JavaScript-разработчиков, привыкших к добавлению свойств объекта вроде `worker.breed`. Если свойство `breed` не существует в объекте `worker`, то движок JavaScript просто его создаст, ведь так? Это сработает в динамически типизируемом коде, но если вы решите использовать выгоды статической типизации, то придется играть по ее правилам.

Обобщения могут использоваться в различных сценариях. Например, вы можете создать функцию, получающую значения разных типов, но в процессе ее вызова вы должны указать конкретный тип. Чтобы использовать обобщения с классом, интерфейсом или функцией, этот класс, интерфейс или функция должны быть написаны особым образом, чтобы поддерживать обобщенные типы.

КОГДА ИСПОЛЬЗОВАТЬ ОБОБЩЕННЫЕ МАССИВЫ

Мы начали этот раздел с показа двух разных способов объявления массива чисел. Давайте рассмотрим другой пример:

```
const values1: string[] = ["Mary", "Joe"];
const values2: Array<string> = ["Mary", "Joe"];
```

Когда все элементы массива имеют один тип, вы можете использовать синтаксис, применяемый для объявления `values1`, — его легче писать и читать. Но если массив может хранить элементы различных типов, вы можете использовать обобщения, чтобы ограничить допустимые для этого массива типы.

Например, вы можете объявить массив, допускающий только строки и числа. В следующем фрагменте кода строка, объявляющая `values3`, приведет к ошибке компиляции, так как логические значения в этом массиве не допускаются.

```
const values3: Array<string | number> = ["Mary", 123, true]; // error
const values4: Array<string | number> = ["Joe", 123, 567]; // no errors
```

Откройте файл определений типов (lib.d.ts) из GitHub репозитория TypeScript по адресу http://mng.bz/qXvJ, и вы увидите объявление интерфейса `Array`, как это показано на рис. 4.5.

`<T>` в строке 1008 является заместителем для конкретного типа, который должен быть передан разработчиком приложения в процессе объявления массива,

как мы и сделали в листинге 4.8. TypeScript требует, чтобы вы вместе с `Array` объявили параметр типа, и при каждом добавлении в этот массив новых элементов компилятор будет проверять, совпадает ли их тип с используемым в объявлении.

```
1008  interface Array<T> {
1009      /**
1010        * Gets or sets the length of the array. This is a number one higher than the h
1011        */
1012      length: number;
1013      /**
1014        * Returns a string representation of an array.
1015        */
1016      toString(): string;
1017      toLocaleString(): string;
1018      /**
1019        * Appends new elements to an array, and returns the new length of the array.
1020        * @param items New elements of the Array.
1021        */
1022      push(...items: T[]): number;
1023      /**
1024        * Removes the last element from an array and returns it.
1025        */
1026      pop(): T;
```

Рис. 4.5. Фрагмент lib.d.ts, описывающий API Array

В листинге 4.8 в роли заместителя обобщенного параметра, представленного буквой `<T>`, мы использовали конкретный тип `<Person>`:

```
const workers: Array<Person>;
```

Но так как обобщения не поддерживаются в JavaScript, вы не увидите их в сгенерированном компиляторе кода — обобщения (и любые другие типы) стираются. Использование параметров типов — это просто сетка безопасности для разработчика на стадии компиляции.

Вы можете увидеть больше обобщенных типов `T` в строках 1022 и 1026 на рис. 4.5. Когда обобщенные типы указаны с аргументами функции, угловые скобки не используются — вы увидите этот синтаксис в листинге 4.9.

В TypeScript нет типа `T`. `T` означает, что методы `push()` и `pop()` позволяют вам включать или исключать объекты того типа, который был передан в процессе объявления массива. Например, в следующем фрагменте кода мы объявили

массив, используя в качестве заместителя `T` тип `Person`, и поэтому можем использовать экземпляр `Person` в роли аргумента метода `push()`:

```
const workers: Array<Person>; workers.push(new Person());
```

> **ПРИМЕЧАНИЕ** Буква `T` обозначает тип, что вполне логично, однако при объявлении обобщенного типа может использоваться любая буква или слово. В картах разработчики зачастую используют букву `K` для `key` (ключа) и `V` для `value` (значения).

Присутствие типа `T` в API интерфейса `Array` говорит нам, что его создатель применил поддержку обобщений. Даже если вы не планируете создавать собственные обобщенные типы, очень важно, чтобы при чтении чьего-то кода или документации TypeScript вы понимали синтаксис обобщений.

4.2.2. Создание собственных обобщенных типов

Вы можете создавать собственные обобщенные классы, интерфейсы или функции. В этом разделе мы создадим обобщенный интерфейс, но сопутствующие ему пояснения также применимы и к созданию классов.

Предположим, у вас есть класс `Rectangle` и вам нужно добавить возможность сравнения размеров двух прямоугольников. Если вы не освоили принцип программирования через интерфейсы (объясненный в разделе 3.2.3), то можете просто добавить в класс `Rectangle` метод `compareRectangles()`.

Однако вооружившись принципом программирования через интерфейсы, вы будете мыслить иначе: «Сегодня мне нужно сравнить прямоугольники, а завтра они попросят меня сравнить другие объекты. Я проявлю находчивость и объявлю интерфейс с функцией `compareTo()`. Тогда класс `Rectangle` и любой другой класс в будущем сможет реализовать этот интерфейс. Алгоритмы сравнивания прямоугольников будут отличаться от сравнивания, скажем, треугольников, но по крайней мере у них будет что-то общее, а сигнатура метода `compareTo()` будет выглядеть одинаковой».

Итак, у вас есть объект некоего типа, и он нуждается в методе `compareTo()`, который сравнит этот объект с другим объектом того же типа. Если этот объект окажется больше, чем второй, `compareTo()` вернет положительное число; если же он будет меньше — вернет отрицательное; в случае их равенства будет возвращен `0`.

Если бы вы не были знакомы с обобщенными типами, то определили бы интерфейс так:

```
interface Comparator {
    compareTo(value: any): number;
}
```

Метод `compareTo()` может получать в качестве аргумента любой объект, а класс, реализующий `Comparator`, должен включать подходящие алгоритмы сравнения (вроде сравнения размеров прямоугольников).

Следующий листинг показывает частичную реализацию классов `Rectangle` и `Triangle`, использующих интерфейс `Comparator`.

Листинг 4.9. Использование интерфейса без обобщенных типов

```
interface Comparator {
    compareTo(value: any): number;  <-- Метод compareTo() получает один параметр типа any
}

class Rectangle implements Comparator {

    compareTo(value: any): number {  <-- Реализация compareTo() в Rectangle
        // Здесь помещается алгоритм для сравнения прямоугольников.
    }
}

class Triangle implements Comparator {

    compareTo(value: any): number {  <-- Реализация compareTo() в Triangle
        // Здесь помещается алгоритм для сравнения треугольников
    }
}
```

Если разработчик хорошенько выспится и выпьет чашку двойного эспрессо, то создаст экземпляры двух прямоугольников, а затем передаст один из них в метод `compareTo()`другого:

```
rectangle1.compareTo(rectangle2);
```

Но что, если кофемашина в то утро не работала? Наш разработчик мог допустить ошибку и попытаться сравнить прямоугольник с треугольником.

```
rectangle1.compareTo(triangle1);
```

`Triangle` подходит типу `any` параметра `compareTo()`, в связи с чем предыдущий код может привести к ошибке компиляции. Чтобы перехватить подобную ошибку процесса компиляции, мы можем использовать обобщенный тип (вместо `any`), чтобы установить ограничение на допустимые для передачи в метод `compareTo()` типы.

```
interface Comparator<T> {
    compareTo(value: T): number;
}
```

Важно, чтобы и интерфейс и метод, оба использовали обобщенный тип, представленный одной буквой `T`. Теперь классы `Rectangle` и `Triangle` могут реализовывать `Comparator`, указывая конкретные типы в угловых скобках.

Листинг 4.10. Использование интерфейса с обобщенным типом

```
interface Comparator <T> {
    compareTo(value: T): number;
}

class Rectangle implements Comparator<Rectangle> {

    compareTo(value: Rectangle): number {
        // Здесь помещается алгоритм сравнения прямоугольников
    }
}

class Triangle implements Comparator<Triangle> {

    compareTo(value: Triangle): number {
        // Здесь помещается алгоритм сравнения треугольников
    }
}
```

Объявляет обобщенный интерфейс, получающий в качестве параметра один тип

Метод compareTo() получает один параметр обобщенного типа

В классе Rectangle метод compareTo() имеет параметр типа Rectangle

В классе Triangle метод compareTo() имеет параметр типа Triangle

Теперь предположим, что наш разработчик пытается повторить ту же ошибку:

```
rectangle1.compareTo(triangle1);
```

Анализатор кода TypeScript подчеркнет `triangle` красной волнистой линией, сообщая об ошибке: «Argument of type 'Triangle' is not assignable to parameter of type 'Rectangle'» (Аргумент типа 'Triangle' не может быть присвоен параметру типа 'Rectangle'). Как вы видите, использование обобщенных типов снижает зависимость качества кода от кофемашины.

Листинг 4.11 показывает еще один рабочий пример, объявляющий интерфейс `Comparator<T>`, который объявляет метод `CompareTo()`. Этот код показывает, как этот интерфейс может использоваться для сравнения прямоугольников, а также программистов. Наши алгоритмы просты:

- Если площадь первого прямоугольника (ширина, умноженная на высоту) больше, чем второго, то первый прямоугольник больше. Они также могут иметь одинаковую площадь.
- Если зарплата первого программиста выше, чем зарплата второго, то первый богаче. Они также могут иметь одинаковую зарплату.

Листинг 4.11. Рабочий пример, использующий обобщенный интерфейс

```
interface Comparator<T> {
    compareTo(value: T): number;
}

class Rectangle implements Comparator<Rectangle> {
```

Объявляет обобщенный интерфейс Comparator

Создает класс, реализующий Comparator для типа Rectangle

```
    constructor(private width: number, private height: number){};

    compareTo(value: Rectangle): number {
        return this.width * this.height - value.width * value.height;
    }
}

const rect1:Rectangle = new Rectangle(2,5);
const rect2: Rectangle = new Rectangle(2,3);

rect1.compareTo(rect2) > 0 ? console.log("rect1 is bigger"):
    rect1.compareTo(rect2) == 0 ? console.log("rectangles are equal") :
➥ console.log("rect1 is smaller");

class Programmer implements Comparator<Programmer> {

    constructor(public name: string, private salary: number){};

    compareTo(value: Programmer): number{
        return this.salary - value.salary;
    }
}

const prog1:Programmer = new Programmer("John",20000);
const prog2: Programmer = new Programmer("Alex",30000);

prog1.compareTo(prog2) > 0 ? console.log(`${prog1.name} is richer`):
    prog1.compareTo(prog2) == 0?
        console.log(`${prog1.name} and ${prog1.name} earn the
➥ same amounts`) :
        console.log(`${prog1.name} is poorer`);
```

Реализует метод для сравнения прямоугольников

Сравнивает прямоугольники (тип T удаляется и замещается Rectangle)

Создает класс, реализующий Comparator для типа Programmer

Реализует метод для сравнения программистов

Сравнивает программистов (тип T удаляется и замещается Programmer)

Выполнение скрипта из листинга 4.11 выводит в консоль следующее:

```
rect1 is bigger
John is poorer
```

Посмотреть эту программу в действии на CodePen можно здесь: http://mng.bz/7zqe.

ПРЕДУСТАНОВЛЕННЫЕ ЗНАЧЕНИЯ ОБОБЩЕННЫХ ТИПОВ

Для использования обобщенного типа вам необходимо передать конкретный тип. Следующий код не скомпилируется, поскольку при использовании типа A мы не указали конкретный тип параметра:

```
class A <T> {
    value: T;
}
```

```
class B extends A { // Ошибка компиляции

}
```

Добавление типа any при использовании класса A исправит эту ошибку:

```
class B extends A <any> {

}
```

Еще один способ сделать это — указать предустановленный параметр при объявлении обобщенного типа. Следующий фрагмент кода скомпилируется без всяких ошибок:

```
class A < T = any > { // объявление предустановленного параметра типа.
    value: T;
}

class B extends A { // Никаких ошибок

}
```

Вместо any вы можете указать другой фиктивный тип:

```
class A < T = {} >
```

В этом случае не потребуется указывать обобщенный параметр при использовании обобщенных классов. На рис. 13.10 видно, как тип React.FC из библиотеки React задействует эти предустановленные типы. Конечно, если вы создаете собственные обобщенные типы и можете обеспечить параметры по умолчанию, имеющие коммерческий смысл (а не просто any), то делайте это всеми доступными способами.

В этом разделе мы создали обобщенный интерфейс Comparator<T>. Теперь давайте посмотрим, как мы можем создать обобщенную функцию.

4.2.3. Создание обобщенных функций

Мы все знаем, как писать функцию, которая может получать аргументы конкретных типов и возвращать значение конкретного типа. Но на сей раз мы напишем обобщенную функцию, которая может получать параметры нескольких типов.

Однако для начала давайте рассмотрим не самое удачное решение, где функция может получать параметр типа any и возвращать значение того же типа. Функция в следующем листинге может регистрировать объекты разных типов и возвращать зарегистрированные данные.

Листинг 4.12. Функция с типом any

```
function printMe(content: any): any {   <-- Объявляет выражение функции с any
    console.log(content);
    return content;
}

const a = printMe("Hello");   <-- Вызывает printMe() со строковым аргументом

class Person{   <-- Объявляет пользовательский тип Person
    constructor(public name: string) { }
}

const b = printMe(new Person("Joe"));   <-- Вызывает printMe() с аргументом типа Person
```

Эта функция работает для разных типов аргументов, но TypeScript не помнит тип аргумента, с которым была вызвана функция `printMe()`. Если вы наведете курсор на переменные `a` и `b` в своей IDE, статический анализатор сообщит типы обеих переменных как `any`.

Если мы хотим знать, какие типы аргументов использовались в вызове `printMe()`, то нужно переписать ее как обобщенную функцию. Следующий листинг показывает синтаксис для передачи обобщенного типа `<T>` функции, ее параметра и возвращаемого значения.

Листинг 4.13. Обобщенная функция

```
function printMe<T> (content: T): T {   <-- Использует тип T для функции, параметра и возвращаемого значения
    console.log(content);
    return content;
}

const a = printMe("Hello");   <-- Вызывает printMe() со строковым аргументом

class Person{
    constructor(public name: string) { }
}

const b = printMe(new Person("Joe"));   <-- Вызывает printMe() с аргументом типа Person
```

В этой версии функции мы объявляем обобщенный тип для функции как `<T>`, а тип параметра и возвращаемого значения как `T`. Теперь типы зафиксированы, и тип константы `a` — это `string`, а тип `b` — это `Person`. Если позднее в сценарии вам потребуется использовать `a` и `b`, статический анализатор TS (и компилятор) произведут должную проверку типов.

> **ПРИМЕЧАНИЕ** Используя ту же букву `T` для типа аргумента функции и возвращаемого типа, мы устанавливаем ограничение, которое гарантирует, что независимо от конкретного типа, использованного в вызове, возвращаемый тип этой функции будет тем же.

Схожим образом вы можете использовать обобщения в выражениях стрелочных функций. Образец кода из листинга 4.13 можно переписать так:

Листинг 4.14. Использование обобщенных типов в стрелочных функциях

```
const printMe = <T> (content: T): T => {  ◄── Сигнатура стрелочной функции начинается с <T>
    console.log(content);
    return content;
}

const a = printMe("Hello");

class Person{
    constructor(public name: string) { }
}

const b = printMe(new Person("Joe"));
```

Вы также можете вызвать эти функции, указав типы явно в угловых скобках:

```
const a = printMe<string>("Hello");

const b = printMe<Person>(new Person("Joe"));
```

Но использование явных типов здесь не требуется, поскольку компилятор TS выведет тип `a` как `string`, а `b` как `Person`.

Предыдущий фрагмент кода может выглядеть неубедительным — кажется, что излишне использовать `<string>`, если и так ясно, что «Hello» является строкой. Но как вы увидите в листинге 4.17, это не всегда может быть так.

Давайте напишем другой небольшой скрипт, который предоставит бóльшую наглядность использования обобщений в классе и функции. Следующий листинг объявляет класс, который может представлять пару ключ — значение. И ключ и значение, оба могут быть представлены несколькими типами, поэтому здесь предчувствуем обобщения.

Листинг 4.15. Обобщенный класс Pair

```
class Pair<K, V> {  ◄──  Объявляет класс с двумя параметризованными типами
    key: K;  ◄──  Объявляет свойство обобщенного типа K
    value: V;  ◄──  Объявляет свойство обобщенного типа V
}
```

Когда вы пишете часть кода, вводящую параметры обобщенных типов, представленных буквами (вроде `K` и `V`), то можете объявить переменные, используя эти буквы так, как если бы они были встроенными типами TS. Когда вы объявляете (и компилируете) конкретный класс `Pair` с конкретными типами для `K` и `V`, `K` и `V` будут удалены и замещены объявленными типами.

Давайте напишем более сжатую версию класса `Pair`, имеющего конструктор и автоматически созданные свойства `key` и `values`:

```
class Pair<K, V> {
    constructor(public key: K, public value: V) {}
}
```

Теперь давайте напишем обобщенную функцию, которая может сравнивать обобщенные пары. Голова еще не закружилась? Функция в следующем листинге объявляет два обобщенных типа, которые также представлены как `K` и `V`.

Листинг 4.16. Обобщенная функция compare

```
function compare <K,V> (pair1: Pair<K,V>, pair2: Pair<K,V>): boolean {
    return pair1.key === pair2.key &&
        pair1.value === pair2.value;
}
```

Объявляет обобщенную функцию

Сравнивает ключи и значения пар

В процессе вызова функции `compare()` вы можете указать два конкретных типа, которые должны совпадать с типами, переданными для ее параметров — объектов `Pair`.

Листинг 4.17 показывает рабочий сценарий, использующий обобщенный класс `Pair` наряду с функцией `compare()`. Сначала мы создаем и сравниваем два экземпляра `Pair`, использующих тип `number` для ключа и тип `string` для значений. Затем мы сравниваем два других экземпляра `Pair`, использующих тип `string` как для ключа, так и для значения.

Листинг 4.17. Использование compare() и Pair

```
class Pair<K, V> {
    constructor(public key: K, public value: V) {}
}

function compare <K,V> (pair1: Pair<K,V>, pair2: Pair<K,V>): boolean {
    return pair1.key === pair2.key &&
        pair1.value === pair2.value;
}

let p1: Pair<number, string> = new Pair(1, "Apple");

let p2 = new Pair(1, "Orange");

// Сравнивает Apple с Orange.
console.log(compare<number, string>(p1, p2));

let p3 = new Pair("first", "Apple");

let p4 = new Pair("first", "Apple");

// Сравнивает Apple с Apple. console.log(compare(p3, p4));
```

Создает первую пару <number, string>

Создает вторую пару <number, string>, используя вывод типа

Сравнивает пары (выводит в консоль «false»)

Создает первую пару <string, string>

Создает вторую пару <string, string>

Сравнивает пары (prints «true»)

Пожалуйста, обратите внимание, что в первом вызове `compare()` мы явно указали конкретные параметры, а во втором вызове нет:

```
compare<number, string>(p1, p2) compare(p3, p4)
```

Первая строка более понятна, так как мы видим, какие типы имеют пары `p1` и `p2`. Кроме того, если вы допустите ошибку, указав неверные типы, компилятор тут же ее перехватит:

```
compare<string, string>(p1, p2) //ошибка компиляции
```

Посмотреть этот код в действии можно здесь: http://mng.bz/m454.

Следующий скрипт показывает другой пример обобщенной функции. На этот раз мы отобразим члены строчного перечисления в пользовательские роли, возвращаемые функцией. Представьте, что механизм авторизации возвращает одну из следующих ролей пользователя: админ или менеджер. Мы хотим использовать строчное перечисление и отобразить роли пользователя в соответствующие члены этого перечисления.

Во-первых, мы объявляем пользовательский тип `User`:

```
interface User {
    name: string;
    role: UserRole;
}
```

Затем мы создаем строковое `enum`, перечисляющее ограниченный набор, который может использоваться в качестве ролей пользователя:

```
enum UserRole {
    Administrator = 'admin',
    Manager = 'manager'
}
```

Следом мы создаем функцию, загружающую объект с жестко закодированным именем пользователя и его ролью. В реальных приложениях мы бы сделали запрос к серверу авторизации, сообщив ID пользователя, чтобы получить его роль. Но для наших целей будет достаточно возвращения жестко закодированного объекта.

```
function loadUser<T>(): T {
    return JSON.parse('{ "name": "john", "role": "admin" }');
}
```

Следующий код показывает обобщенную функцию, возвращающую `User` и затем отображающую пользовательскую роль в действие, используя строчное перечисление.

Листинг 4.18. Отображение строчных перечислений

```
interface User {  ◄—— Объявляет пользовательский тип User
    name: string;
    role: UserRole;
}

enum UserRole {  ◄—— Объявляет строчное перечисление
    Administrator = 'admin',
    Manager = 'manager'
}

function loadUser<T>(): T {  ◄—— Объявляет обобщенную функцию
    return JSON.parse('{ "name": "john", "role": "admin" }');
}

const user = loadUser<User>();  ◄—— Вызывает обобщенную функцию с конкретным типом User

switch (user.role) {  ◄—— Переключает роль пользователя с помощью строчного enum
    case UserRole.Administrator: console.log('Show control panel'); break;
    case UserRole.Manager: console.log('Hide control panel'); break;
}
```

Сценарий в листинге 4.18 в процессе вызова `loadUser()` использует тип `User`, и обобщенный тип `T`, объявленный в качестве возвращаемого типа этой функции, становится конкретным типом `User`. Обратите внимание, что жестко закодированный объект, возвращаемый этой функцией, имеет ту же структуру, что и интерфейс `User`.

Здесь `user.role` всегда будет `admin`, который соответствует члену перечисления `UserRole.Administrator`, и скрипт выведет в консоль «Show control panel» (Показать панель управления). Вы можете посмотреть этот сценарий в действии на CodePen: http://mng.bz/5Aqa.

> **ПРИМЕЧАНИЕ** В главе 10 (листинг 10.15) вы увидите код класса `MessageServer`, использующий обобщенный тип `<T>` в нескольких методах.

4.2.4. Обеспечение возвращаемого типа функции высшего порядка

Если функция может получать функцию в качестве аргумента или возвращать другую функцию, мы называем ее *функцией высшего порядка*. В текущем разделе мы покажем вам пример, который обеспечивает возвращаемый тип функции высшего порядка, в то же время допуская аргументы разных типов. Предположим, нам нужно написать функцию высшего порядка, возвращающую функцию со следующей сигнатурой:

```
(c: number) => number
```

Эта стрелочная функция получает в качестве аргумента число и возвращает также число. Функция высшего порядка (мы будем использовать нотацию стрелочной функции) может выглядеть так:

```
(someValue: number) => (multiplier: number) => someValue * multiplier;
```

Мы не написали инструкцию `return` после первой стрелочной функции, так как в однострочных стрелочных функциях возвращение неявное. Следующий листинг показывает использование подобной функции.

Листинг 4.19. Использование функции высшего порядка

```
const outerFunc = (someValue: number) =>
    (multiplier: number) => someValue * multiplier;

const innerFunc = outerFunc(10);

let result = innerFunc(5);

console.log(result);
```

Объявляет функцию высшего порядка

innerFunc является замыканием, знающим, что someValue = 10

Вызывает возвращаемую функцию

Выводит в консоль 50

Теперь давайте несколько усложним задачу. Нужно позволить нашей функции высшего порядка быть вызванной с аргументами разных типов, обеспечивая тем самым постоянное возвращение функции с одинаковой сигнатурой:

```
(c: number) => number
```

Давайте начнем с объявления обобщенной функции, которая может получать обобщенный тип `T`, но возвращает функцию `(c: number) ? number`:

```
type numFunc<T> = (arg: T) => (c: number) => number;
```

Теперь мы можем объявить переменные типа `numFunc`, и TS обеспечит, чтобы эти переменные были функциями типа `(c: number) ? number`.

Листинг 4.20. Использование обобщенной функции numFunc<T>

```
const noArgFunc: numFunc<void> = () =>
        (c: number) => c + 5;
const numArgFunc: numFunc<number> = (someValue: number) =>
        (multiplier: number) => someValue * multiplier;
const stringArgFunc: numFunc<string> = (someText: string) =>
        (padding: number) => someText.length + padding;

const createSumString: numFunc<number> = () => (x: number) => 'Hello';
```

Вызывает функцию без аргументов

Вызывает функцию с численным аргументом

Вызывает функцию со строчным аргументом

Ошибка компиляции: numFunc ожидает другую сигнатуру

Последняя строчка не скомпилируется, так как сигнатура возвращенной функции — это `(c: number) ? string`, которая не может быть присвоена переменной типа `numFunc`. Изучить этот пример в песочнице можно здесь: http://mng.bz/6wqA.

ИТОГИ

- TypeScript предлагает ключевое слово `enum`, которое может определять ограниченный набор констант.
- TypeScript поддерживает численные и строчные перечисления.
- Если вы используете ключевое слово `const` в процессе объявления `enum`, то его значения будут встроены и код JavaScript сгенерирован не будет.
- Обобщение — это часть кода, которая может обрабатывать значения нескольких типов, указываемых при использовании кода.
- Вы можете создавать собственные обобщенные классы, интерфейсы и функции.

5 Декораторы и продвинутые типы

В этой главе:

- ✓ Назначение декораторов в TypeScript.
- ✓ Как с помощью отображенных типов создать новый тип, основанный на существующем.
- ✓ Принципы работы условных типов.
- ✓ Совмещение отображенных и условных типов.

В предыдущих главах мы рассмотрели основные типы, которых должно быть достаточно для большинства задач при написании кода. Однако TypeScript идет дальше и предлагает дополнительные производные типы, которые могут быть очень полезны в определенных сценариях.

В заголовке мы использовали слово «продвинутые» по двум причинам. Во-первых, не обязательно знать эти типы, чтобы быть продуктивным членом команды. Во-вторых, их синтаксис может быть не сразу очевиден для разработчика, знакомого с другими языками программирования.

Название этой книги «TypeScript быстро», и содержание первых четырех глав выполняет обещание представить основные синтаксические конструкции языка в ускоренном режиме. Однако эту главу может потребоваться прочесть не один раз.

Хорошие новости состоят в том, что изучение продвинутых типов, представленных в этой главе, не является необходимым для понимания остальной части книги. Можете пропустить их, если хотите изучить язык быстро. Но изучить эту главу стоит, если:

- Настало время готовиться к вашему следующему собеседованию, предполагающему, что вы обладаете редко применяемыми знаниями.
- Вы смотрите на какой-то конкретный код и интуитивно понимаете, что для него есть более изящное решение.
- Вам мало работы с интерфейсами, обобщениями и перечислениями и хочется узнать, что еще доступно для изучения.

ПРИМЕЧАНИЕ В этой главе мы будем часто использовать синтаксис *обобщений*, предполагая, что вы уже прочли и усвоили материал из раздела 4.2. Это важно для понимания отображенных и условных типов, описываемых в текущей главе.

5.1. ДЕКОРАТОРЫ

В документации TS *декоратор* — особый вид объявления, который может быть прикреплен к объявлению класса, метода, аксессора, свойства или параметра. Декораторы используют форму `@expression`, где выражение должно вычисляться в функцию, которая будет вызвана при выполнении с информацией о декорированном объявлении[1].

Предположим, у нас есть `class A {…}`, а также магический декоратор `@injectible()`, умеющий инстанцировать классы и внедрять их экземпляры в другие объекты. Можно декорировать один из наших классов так:

```
@Injectable() class A {}
```

Как можно догадаться, декоратор `@injectible` каким-то образом будет изменять поведение `class A`. Альтернативный способ изменения его поведения без модификации кода подразумевал бы создание подкласса `class A` и добавление или переназначение поведения в нем. Но простое добавление декоратора в определение класса выглядит более изящно.

Мы можем также сказать, что декоратор добавляет *метаданные* конкретной цели, которой в нашем примере является `class A`. В общем смысле метаданные — это дополнительные данные о других данных. Возьмем, к примеру, файл mp3, в котором песня — это данные. Но у mp3-файла могут быть и дополнительные

[1] «Decorators» в документации TypeScript. (www.typescriptlang.org/docs/handbook/decorators.html).

свойства, такие как имя артиста, название альбома, картинка и пр., — это уже метаданные файла. Подобным образом вы можете использовать декораторы для аннотирования TypeScript классов метаданными, описывающими дополнительные возможности, которые вы хотите добавить классу.

Имена декораторов начинаются со знака `@`, например `@Component`. Вы можете написать собственные декораторы, но скорее всего, будете использовать те, что доступны в библиотеке или фреймворке.

Рассмотрите следующий простой класс:

```
class OrderComponent {
    quantity: number;
}
```

Представьте, что вы хотите преобразовать этот класс в компонент UI. Более того, вы хотите объявить, что значение для свойства `quantity` будет передано родительским компонентом. Если вам доводилось использовать с TypeScript фреймворк Angular, то вы могли применять встроенный декоратор `@Component` для класса и `@Input()` для свойства.

Листинг 5.1. Пример компонента Angular

В Angular декоратор `@Component()` может применяться только к классу. При этом он поддерживает различные свойства, такие как `selector` и `template`. Декоратор `@input()`, в свою очередь, может применяться только к свойствам класса. Мы можем также сказать, что эти декораторы предоставили метаданные о классе `OrderComponent` и свойстве `quantity` соответственно.

Чтобы от декораторов была польза, должен также присутствовать код, знающий, как их считывать и делать то, что они предписывают. В примере из листинга 5.1 фреймворк Angular считывает эти декораторы и генерирует дополнительный код для преобразования класса `OrderComponent` в отображаемый компонент UI.

TypeScript не имеет встроенных декораторов, но вы можете создавать свои собственные или использовать присутствующие в выбранном вами фреймворке или библиотеке.

ПРИМЕЧАНИЕ В главах 11 и 12 мы будем работать с фреймворком Angular, и вы увидите множество примеров использования декораторов. Они активно используются серверной частью фреймворка Nest.js (https://docs.nestjs.com/custom-decorators), библиотекой управления состояниями MobX (https://mobjx.org/refguide/modifiers.html) и UI библиотекой Stencil.js (https://stenciljs.com/docs/decorators).

Декораторы в листинге 5.1 позволили вам указать дополнительное поведение для класса и его свойства в краткой и объявляющей форме. Конечно же, использование декораторов — это не единственный способ добавления поведения объекту. Например, создатели Angular могли бы сделать абстрактный класс `UIComponent` с определенным конструктором, вынудив разработчиков расширять его каждый раз, когда им нужно превратить класс в компонент UI. Но использование компонентов декораторов — это более лаконичное, читаемое и декларативное решение.

Декораторы (в отличие от наследования) разделяют задачи и облегчают обслуживание кода, поскольку фреймворк волен интерпретировать их по своему усмотрению. В противоположность этому, если бы компонент был подклассом, то либо переопределял, либо ожидал, либо опирался на конкретное поведение методов суперкласса.

ПРИМЕЧАНИЕ Существует предложение добавить декораторы в JavaScript. Сейчас оно находится на второй стадии рассмотрения (https://tc39.github.io/proposal-decorators).

Несмотря на то что декораторы были представлены еще в 2015 году, они до сих пор считаются экспериментальной функцией, и вам приходится компилировать приложение с tsc опцией `--experimentalDecorators`. Если вы используете tsconfig.json, добавьте в него следующую опцию компиляции:

```
"experimentalDecorators": true
```

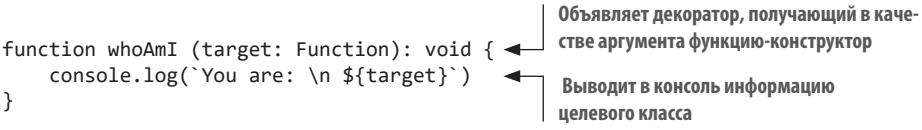
Декоратор можно использовать для просмотра или изменения определения цели (класса, метода и т. п.). В зависимости от выбранной цели также будут отличаться сигнатуры этих специальных функций (декораторов). В текущей главе мы покажем вам, как создавать декораторы классов и методов.

5.1.1. Создание декораторов классов

Этот вид декораторов применяется к классу, а функция декоратора выполняется при выполнении конструктора. Для их реализации необходим один параметр — функция — конструктор класса. Другими словами, декоратор класса будет получать функцию — конструктор декорируемого класса.

Следующий листинг объявляет декоратор класса, который только выводит в консоль информацию о классе.

Листинг 5.2. Объявление пользовательского декоратора whoAmI




```
function whoAmI (target: Function): void {
    console.log(`You are: \n ${target}`)
}
```

> **ПРИМЕЧАНИЕ** Если возвращаемый тип декоратора класса — `void`, тогда эта функция не возвращает никакого значения (см. раздел 2.1.1). Подобный декоратор не заместит объявление класса — он только просмотрит его, как показано в листинге 5.3. Но декоратор также может и изменять объявление класса, возвращая при этом измененную версию этого класса (функции-конструктора). В конце раздела мы покажем, как именно это делается.

Чтобы использовать декоратор `whoAmI`, просто поместите перед именем класса выражение `@whoAmI`. В итоге функция — конструктор класса будет автоматически передана декоратору. В нашем примере у конструктора есть два аргумента: строка и число.

Листинг 5.3. Применение декоратора whoAmI к классу

```
@whoAmI
class Friend {

    constructor(private name: string, private age: number){}
}
```

Когда TS-код транспилируется в JS, tsc проверяет, использовались ли декораторы. Если да, то он генерирует дополнительный JS-код, который будет задействован при выполнении.

Вы можете запустить приведенный пример кода в песочнице здесь: http://mng.bz/omop. Для этого вам потребуется активировать опцию `experimentalDecorators` в конфигурации и нажать кнопку `Run`. Будет запущена JavaScript-версия кода, которая произведет в консоль браузера следующий вывод:

```
You are:
function Friend(name, age) {
    this.name = name;
    this.age = age;
}
```

> **СОВЕТ** Чтобы увидеть эффект декоратора `@whoAmI` в сгенерированном коде в песочнице, удалите декоратор из объявления класса `Friend` и обратите внимание на отличия в сгенерированном JS-коде.

Вам может показаться, что декоратор `@whoAmI` не слишком полезен, поэтому давайте создадим еще один пример. Предположим, вы разрабатываете

UI-фреймворк и хотите позволить классам преобразование в компоненты UI посредством объявления. Вы хотите создать функцию, получающую произвольный аргумент, вроде HTML-строки для отображения.

Листинг 5.4. Объявление пользовательского декоратора UIcomponent

```
function UIcomponent (html: string) {
        console.log(`The decorator received ${html} \n`);
    return function(target: Function) {
          console.log(`Someone wants to create a UI component from
                      \n ${target} `);
    }
}
```

У этой фабрики декораторов есть аргумент

Это функция-декоратор

Выводит строку, полученную декоратором

Здесь вы видите функцию-декоратор внутри другой функции. Мы можем назвать внешнюю функцию *фабрикой декораторов*. Она может получать любые аргументы и применять логику приложения, чтобы решать, какой декоратор возвращать. В данном случае в листинге 5.4 есть только одна инструкция `return`, и этот код всегда возвращает одну функцию-декоратор, но ничто не мешает вам по условию возвращать нужный декоратор, основываясь на параметрах, переданных фабрике декораторов.

> **ПРИМЕЧАНИЕ** Во врезке под названием «Формальные объявления сигнатур декораторов» вы увидите, что требования для сигнатур декораторов зависят от того, что они декорируют. Итак, как нам удалось использовать произвольный аргумент в функции `UIComponent()`? Причина в том, что `UIComponent()` является не декоратором, а фабрикой декораторов, возвращающей декоратор с правильной сигнатурой `function (target: Function)`.

Следующий листинг показывает класс `Shopper`, декорированный как `UIcomponent`:

Листинг 5.5. Применение пользовательского декоратора UIcomponent

```
@UIcomponent('<h1>Hello Shopper!</h1>')
class Shopper {

    constructor(private name: string) {}

}
```

Передает декоратору HTML

Конструктор класса, получающий имя покупателя

Выполнение этого кода произведет следующий вывод:

```
The decorator received <h1>Hello Shopper!</h1>

Someone wants to create a UI component from
function Shopper(name) {
        this.name = name;
}
```

Посмотрите этот код на CodePen: http://mng.bz/nv62.

До сих пор все наши примеры декораторов только просматривали классы, не изменяя сами объявления этих классов. В следующем примере мы продемонстрируем вам декоратор, который это делает. Но начнем со знакомства с примесью конструктора.

В JS *примесь* (миксин) — это часть кода, реализующая конкретное поведение. Примеси не используются отдельно, но их поведение может быть добавлено другим классам. Хотя JS не поддерживает множественное наследование, вы можете совместить поведения из нескольких классов, используя примеси.

Если примесь не имеет конструктора, смешивание ее кода с другими классами состоит в копировании ее свойств и методов в целевой класс. Но если примесь содержит собственный конструктор, то он должен быть способен получать любое число параметров любого типа; в противном случае он не будет «смешиваться» с произвольными конструкторами целевых классов.

TypeScript поддерживает *примесь-конструктор*, имеющую следующую сигнатуру:

```
{ new(...args: any[]): {} }
```

Она использует один оставшийся аргумент (три точки) типа `any[]` и может быть смешана с другими классами, имеющими конструкторы. Давайте объявим для этой примеси псевдоним типа:

```
type constructorMixin = { new(...args: any[]): {} };
```

Соответственно, следующая сигнатура представляет обобщенный тип `T`, расширяющий `constructorMixin`; в TypeScript это также означает, что тип `T` может быть присвоен типу `constructorMixin`:

```
<T extends constructorMixin>
```

Вы будете использовать эту сигнатуру для создания декоратора класса, изменяющего оригинальный конструктор класса. Сигнатура декоратора класса будет выглядеть так:

```
function <T extends constructorMixin> (target: T) {
    // Здесь реализуется декоратор
}
```

Теперь мы подготовились к написанию декоратора, изменяющего объявление (и конструктор) целевого класса. Давайте предположим, что у нас есть следующий класс `Greeter`:

Листинг 5.6. Недекорированный класс Greeter

```
class Greeter {
    constructor(public name: string) { }
    sayHello() { console.log(`Hello ${this.name} `) };
}
```

Мы можем инстанцировать и использовать его так:

```
const grt = new Greeter('John');
grt.sayHello(); // выводит "Hello John"
```

Мы хотим создать декоратор, который может принимать параметр `salutation` (приветствие) и добавлять классу новое свойство `message`, конкатенируя заданное приветствие и имя. Мы также хотим изменить код метода `sayHello()`, чтобы он выводил в консоль `message`.

Следующий листинг показывает *функцию высшего порядка* (ту, которая возвращает другую функцию), реализующую наш декоратор. Мы также можем назвать ее *фабричной функцией*, поскольку она создает и возвращает функцию.

Листинг 5.7. Объявление декоратора useSalutation

```
function useSalutation(salutation: string) {

    return function <T extends constructorMixin> (target: T) {
      return class extends target {
        name: string;
        private message = 'Hello ' + salutation + this.name;

        sayHello() { console.log(`${this.message}`); }
      }
    }
}
```

Фабричная функция, получающая один параметр — приветствие

Тело декоратора

Повторно объявляет декорированный класс

Добавляет в новый класс приватное свойство

Повторно объявляет метод

Начиная со строки `return class extends target`, мы предоставляем другое объявление декорированного класса. В частности, мы добавили оригинальному классу свойство `message` и заместили тело метода `sayHello()`, чтобы оно использовало приветствие, предоставленное декоратором.

В следующем листинге мы используем декоратор `@useSalutation` с классом `Greeter`. Вызов `grt.sayHello()` выведет `Hello Mr. Smith`.

Листинг 5.8. Использование декорируемого класса Greeter

```
@useSalutation("Mr. ")
class Greeter {
```

Применяет к классу декоратор с аргументом

```
    constructor(public name: string) { }
    sayHello() { console.log(`Hello ${this.name} `) }
}

const grt = new Greeter('Smith');
grt.sayHello();
```

Запустите этот код в песочнице здесь: http://mng.bz/vlM4. Нажмите кнопку Run и откройте консоль браузера, чтобы увидеть вывод.

Очень хорошо, что у нас есть такой мощный механизм для замещения объявления класса, но использовать его следует с осторожностью. Избегайте изменения публичного API класса, потому что статический анализатор кода не предложит автоподстановку для публичных свойств и методов, добавленных декораторами.

Предположим, вы изменили декоратор `useSalutation()`, чтобы он добавлял целевому классу публичный метод `sayGoodbye()`. После набора `grt.`, как в листинге 5.8, ваша IDE по-прежнему подскажет вам только метод `sayHell()` и свойство `name`. Она не предложит `sayGoodbye()`, даже несмотря на то что в итоге `grt.sayGoodbye()` будет отлично работать.

ФОРМАЛЬНЫЕ ОБЪЯВЛЕНИЯ СИГНАТУР ДЕКОРАТОРОВ

Декоратор — это функция, и ее сигнатура зависит от цели. Сигнатуры для декораторов классов и методов будут отличаться. После того как вы установите TS, он будет содержать несколько файлов с объявлениями типов. Один из них называется lib.es5.d.ts и включает именно объявления типов для декораторов, имеющих разные цели.

```
declare type ClassDecorator =                    <-- Сигнатура для декоратора класса
    <TFunction extends Function>(target: TFunction) =>
➥ TFunction | void;
declare type PropertyDecorator =                 <-- Для декоратора свойства
    (target: Object, propertyKey: string | symbol) => void;
declare type MethodDecorator =                   <-- Для декоратора метода
    <T>(target: Object, propertyKey: string | symbol,
    descriptor: TypedPropertyDescriptor<T>) =>
➥ TypedPropertyDescriptor<T> | void;
declare type ParameterDecorator =                <-- Подпись для декоратора параметров
    (target: Object, propertyKey: string | symbol,
➥ parameterIndex: number) => void;
```

В разделе 4.2.3 мы объяснили обобщенные функции, а также синтаксис именованных и стрелочных функций. Содержание упомянутого раздела должно помочь в понимании сигнатур, приведенных в предыдущем коде. Посмотрите на следующую строку:

```
<T>(someParam: T) => T | void
```

Все верно — она объявляет, что стрелочная функция может получать параметр обобщенного типа T и возвращать либо значение типа T, либо значение типа void. Теперь давайте попытаемся прочитать объявление сигнатуры ClassDecorator:

```
<TFunction extends Function>(target: TFunction) => TFunction | void
```

Она объявляет, что стрелочная функция может получать параметр обобщенного типа TFunction, имеющего дополнительное ограничение: конкретный тип должен быть подтипом Function. Любой TS-класс является подтипом Function, который представляет функцию-конструктор. Другими словами, целью этого декоратора должен быть класс, и декоратор может либо возвращать значение, имеющее тип этого класса, либо не возвращать значение вообще.

Взгляните еще раз на декоратор класса @whoAmI, показанный в листинге 5.2. Мы не использовали в нем выражение стрелочной функции, но эта функция имела следующую сигнатуру, допустимую для декораторов классов:

```
function whoAmI (target: Function): void
```

Так как сигнатура функции whoAmI() не возвращает значение, мы можем сказать, что этот декоратор только просматривает цель. Если бы вы хотели изменить оригинальную цель в декораторе, то вам бы потребовалось вернуть измененный класс, но его тип был бы таким же, как изначально переданный вместо Tfunction (подтипа Function).

5.1.2. Создание декораторов методов

Теперь давайте создадим декоратор, который можно применить к методу класса. Например, вам может понадобиться создать декоратор @deprecated, чтобы помечать методы, которые будут вскоре удалены. Как видно из врезки «Формальные объявления сигнатур декораторов», функция MethodDecorator требует три параметра:

- target — объект, ссылающийся на созданный экземпляр класса, определяющего метод;
- propertyKey — имя декорируемого метода;
- descriptor — дескриптор декорируемого метода.

Параметр descriptor содержит объект, описывающий метод, декорируемый вашим кодом. Говоря конкретнее, TypedPropertyDescriptor имеет свойство value, хранящее оригинальный код декорированного метода. Изменяя значение этого свойства внутри декоратора метода, вы можете модифицировать оригинальный код декорируемого метода.

Рассмотрим класс `Trader`, имеющий метод `placeOrder()`:

```
class Trade {

    placeOrder(stockName: string, quantity: number, operation:
➥ string, traderID: number) {

        // Здесь помещается реализация метода
     }
  // Здесь помещаются другие методы
}
```

Предположим, есть трейдер с ID 123, и он может разместить заявку на покупку 100 акций IBM следующим образом:

```
const trade = new Trade();
trade.placeOrder('IBM', 100, 'Buy', 123);
```

Этот код отлично работал годами, но было введено новое положение: «В целях аудита все сделки должны быть зарегистрированы». Один из способов сделать это — пройти по всем методам, относящимся к покупке/продаже финансовых продуктов, которые могут иметь различные параметры, и добавить код, регистрирующий их вызовы. Но создание декоратора метода `@logTrade`, работающего с любым методом и регистрирующего параметры, будет более изящным решением. Следующий листинг показывает код декоратора метода `@logTrade`.

Листинг 5.9. Декоратор метода @logTrade

```
function logTrade(target, key, descriptor) {   <-- Декоратор метода должен иметь три аргумента

    const originalCode = descriptor.value;   <-- Хранит код оригинального метода

    descriptor.value = function () {   <-- Изменяет код декорируемого метода

        console.log(`Invoked ${key} providing:`, arguments);
        return originalCode.apply(this, arguments);   <-- Вызывает целевой метод
    };
  return descriptor;   <-- Возвращает измененный метод
}
```

Мы сохранили код оригинального метода, а затем изменили полученный дескриптор, добавив инструкцию `console.log()`. После этого мы использовали JS-функцию `apply()`, чтобы вызвать декоратор метода. В завершение мы вернули измененный дескриптор метода.

В следующем листинге мы применили декоратор `@logTrade` к методу `placeOrder()`. Независимо от того, как реализован `placeOrder()`, его декорированная версия начнет выводить следующее сообщение: «`Invoked placeOrder providing:`» (Вызван `placeOrder`, в который передано:).

Листинг 5.10. Использование декоратора @logTrade

```
class Trade {

    @logTrade  <-- Декорирует метод placeOrder()
    placeOrder(stockName: string, quantity: number,
                          operation: string, tradedID: number) {
      // Здесь помещается реализация метода
    }
}

const trade = new Trade();
trade.placeOrder('IBM', 100, 'Buy', 123);  <-- Вызывает placeOrder()
```

Запустите реализацию примера предыдущего кода в песочнице TS: http://mng.bz/4e7j.

После вызова декорированного метода `placeholder()` вывод консоли будет выглядеть подобным образом:

```
Invoked placeOrder providing:
Arguments(4) 0: "IBM"
1: 100
2: "Buy"
3: 123
```

Создав декоратор метода, мы избавились от необходимости обновления кода потенциально многих связанных с куплей-продажей методов, требующих аудита. Кроме того, декоратор `@logTrade` может работать с методами, которые даже еще не были написаны.

Мы показали вам, как писать декораторы классов и методов, которые должны дать вам хорошую основу для дальнейшего самостоятельного освоения декораторов свойств и параметров.

5.2. ОТОБРАЖЕННЫЕ ТИПЫ

Отображенные типы позволяют вам создавать новые типы из уже имеющихся. Это делается с помощью применения функции преобразования к существующему типу. Давайте посмотрим, как они работают.

5.2.1. Отображенный тип Readonly

Представьте, что вам нужно передать объекты типа `Person` (показан далее) в функцию `doStuff()` для обработки:

```
interface Person {
    name: string;
    age: number;
}
```

Тип `Person` используется в нескольких местах, но вы не хотите позволить функции `doStuff()` случайно изменить какие-нибудь свойства `Person`, например `age` в следующем листинге:

Листинг 5.11. Неправомерное изменение age (возраста)

```
const worker: Person = {name: "John", age: 22};

function doStuff(person: Person) {

    person.age = 25;  <-- Мы не хотим допустить такое
}
```

Ни одно из свойств типа `Person` не было объявлено с модификатором `readonly`. Стоит ли нам объявить еще один тип, просто чтобы использовать его с `doStuff()`, как показано ниже?

```
interface ReadonlyPerson {
    readonly name: string;
    readonly age: number;
}
```

Нужно ли вам объявлять (и обслуживать) новый тип каждый раз, когда вам требуется создать версию уже существующего, но только для чтения? Здесь есть решение получше. Мы можем использовать встроенный отображенный тип `Readonly`, чтобы преобразовать все свойства ранее объявленного типа в `readonly`. Нам всего лишь нужно изменить сигнатуру функции `doStuff()`, чтобы вместо `Person` она получала аргумент типа `Readonly<Person>`.

Листинг 5.12. Использование отображенного типа Readonly

```
const worker: Person = {name: "John", age: 22};

function doStuff(person: Readonly<Person>) {  <-- Изменяет существующий тип на отображенный тип Readonly

    person.age = 25;  <-- Эта строка генерирует ошибку компиляции
}
```

Чтобы понять, почему попытка изменить значение свойства age вызывает ошибку компиляции, вам нужно посмотреть, как объявлен тип `Readonly`, что, в свою очередь, требует понимания оператора `keyof` и поиска типа (`lookup type`).

Keyof и поиск типов

Прочтение объявлений встроенных отображенных типов в файле typescript/lib/lib.es5.d.ts поможет понять их внутренние процессы, но требует некоторого

знакомства с *запросом индексов типа* с помощью `keyof`, а также *поиском типа*.

В lib.es5.d.ts вы можете найти следующее объявление отображающей функции `Readonly`:

Листинг 5.13. Объявление отображенного типа Readonly

```
type Readonly<T> = {
    readonly [P in keyof T]: T[P];
};
```

Мы предполагаем, что вы ознакомились с обобщениями в главе 4 и знаете, что значит `<T>` в угловых скобках. Обычно буква `T` в обобщениях представляет тип: `K` для ключа, `V` для значения, `P` для свойства и т. д.

`Keyof` также называется *запросом индекса типа* и предоставляет объединение допустимых имен свойств (ключей) заданного типа. Если тип `Person` — это наш `T`, тогда `keyof T` предоставит объединение `name` и `age`. Рисунок 5.1 показывает скриншот, сделанный в момент наведения указателя на пользовательский тип `propNames`. Как вы видите, тип `propNames` — это объединение `name` и `age`.

В листинге 5.13 фрагмент `[P in keyof T]` означает: «Предоставьте мне объединение всех свойств заданного типа `T`». Этот фрагмент выглядит так, как будто мы обращаемся к элементам какого-либо объекта, но фактически это делается для объявления типов. Запрос `keyof` для типа может использоваться только в объявлениях типов.

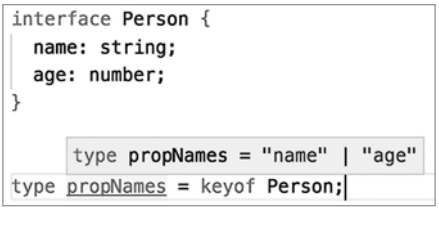


Рис. 5.1. Применение keyof к типу Person

Мы знаем, как получить доступ к именам свойств заданного типа, но для создания отображенного типа из существующего нам также нужно знать типы свойств. В случае с типом `Person` нужна возможность выяснить программно, что типы свойств — это `string` и `number`.

Именно для этого используется *поиск типов*. Часть `T[P]` (из листинга 5.13) как раз является примером поиска типа и означает «Предоставьте мне тип свойства `P`». Рисунок 5.2 показывает скриншот, сделанный в момент наведения курсора на тип `propTypes`. Свойства имеют типы `string` и `number`.

Теперь давайте прочитаем код в листинге 5.13 еще раз. Объявление типа `Readonly<T>` означает «Найти имена и типы свойств конкретного переданного типа и применить к каждому свойству квалификатор `readonly`».

В нашем примере `Readonly<Person>` создаст отображенный тип, который будет выглядеть так:

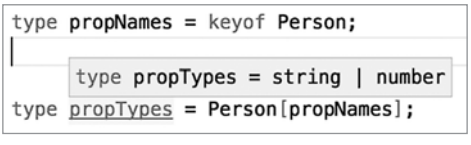


Рис. 5.2. Получение типов свойств Person

Листинг 5.14. Применение отображенного типа Readonly к типу Person

```
interface Person {
    readonly name: string;
    readonly age: number;
}
```

Теперь вы можете видеть, почему попытка изменить возраст приводит к ошибке компиляции «Cannot assign to age because it's a read-only property» (Невозможно выполнить присвоение к age, так как это свойство только для чтения). По сути, мы взяли существующий тип `Person` и отобразили его в схожий тип, но со свойствами только для чтения. Вы можете увидеть, что этот код не компилируется, здесь: http://mng.bz/Q05v.

Вы можете сказать: «Хорошо, я понял, как применять отображенный тип `Readonly`, но в чем здесь практическая польза?» Позже, в листинге 10.16, вы увидите два метода, использующие тип `Readonly` с их аргументом сообщения, схожим с приведенным ниже:

```
replyTo(client: WebSocket, message: Readonly<T>): void
```

Этот метод может отправлять сообщения узлам блокчейна через протокол WebSocket. Сервер сообщений не знает, какие типы сообщений будут переданы, и тип сообщений обобщен. Для предотвращения случайного изменения сообщения внутри `replyTo()`мы используем там отображенный тип `Readonly`.

Давайте рассмотрим еще один пример кода, показывающий выгоды от использования `keyof` и `T[P]`. Представьте, что нужно написать функцию для фильтрации обобщенного массива объектов, сохраняя только те, что имеют указанное значение в указанном свойстве. В первой версии мы не будем использовать проверку типов и напишем функцию так:

Листинг 5.15. Ущербная версия функции filterBy()

```
function filterBy<T>(
    property: any,
    value: any,
    array: T[]) {

    return array.filter(item => item[property] === value);
}
```

Оставляет только те объекты, которые имеют предоставленное значение в указанном свойстве

Вызов этой функции с несуществующим именем свойства или неверным типом значения приведет к неочевидным багам.

Следующий листинг объявляет тип `Person` и функцию. Последние две строки вызывают функцию, передавая либо несуществующее свойство `lastName`, либо неверный тип для `age`.

Листинг 5.16. Версия filterBy() с ошибками

```
interface Person {
    name: string;
    age: number;
}

const persons: Person[] = [
    { name: 'John', age: 32 },
    { name: 'Mary', age: 33 },
];

function filterBy<T>(
    property: any,
    value: any,
    array: T[]) {

return array.filter(item => item[property] === value);
}

console.log(filterBy('name', 'John', persons));

console.log(filterBy('lastName', 'John', persons));

console.log(filterBy('age', 'twenty', persons));
```

Эта функция не проверяет типы

Фильтрует данные на основе свойства/значения

Верный вызов функции

Неверный вызов функции

Неверный вызов функции

Две последние строки кода будут возвращать нулевые объекты без каких-либо жалоб, даже несмотря на то что тип `Person` не имеет свойство `lastName` и тип свойства `age` не является строкой. Другими словами, код в листинге 5.16 содержит ошибки.

Давайте изменим сигнатуру функции `filterBy()`, чтобы она перехватывала эти ошибки в процессе компиляции. Новая версия `filterBy()` показана в следующем листинге.

Листинг 5.17. Улучшенная версия filterBy()

```
function filterBy<T, P extends keyof T>(
    property: P,
    value: T[P],
    array: T[]) {

    return array.filter(item => item[property] === value);
}
```

Проверяет, чтобы переданное свойство P принадлежало объединению [keyof T]

Свойство для фильтра

Значение для фильтра должно иметь тип переданного свойства P

В первую очередь фрагмент `<T, P extends keyof T>` сообщает, что наша функция принимает два обобщенных значения: `T` и `P`. Мы также добавили ограничение, что `P extends keyof T`. Другими словами, `P` должен быть одним из свойств переданного типа `T`. Если конкретный тип `T` будет `Person`, тогда `P` может быть либо `name`, либо `age`.

Сигнатура функции в листинге 5.17 имеет и другое ограничение — `value: T[P]`, которое означает, что переданное значение должно быть того же типа, что объявлен для `P` в типе `T`. Вот почему следующие строки будут вызывать ошибки компиляции.

Листинг 5.18. Эти строки не скомпилируются

```
filterBy('lastName', 'John', persons)
filterBy('age', 'twenty', persons)
```

Несуществующее свойство lastName

Значение age должно быть числом

Как вы видите, введение `keyof` и поиска типа в сигнатуру функции позволяет вам перехватить возможные ошибки во время компиляции. Этот код в действии можно посмотреть здесь: http://mng.bz/XpXa.

5.2.2. Объявление собственных отображенных типов

В листинге 5.13 показана функция преобразования для встроенного отображенного типа `Readonly`. Вы можете определять свои собственные функции преобразования, используя аналогичный синтаксис.

Попробуем определить тип `Modifiable` — противоположный `Readonly`. В предыдущем разделе мы взяли тип `Person` и обозначили все его свойства только для чтения, применив отображенный тип `Readonly: Readonly<Person>`. Теперь предположим, что свойства типа `Person` изначально были объявлены с модификатором `readonly` так:

```
interface Person {
    readonly name: string;
    readonly age: number;
}
```

Как бы вы могли удалить квалификаторы `readonly` из объявления `Person`, если бы вдруг это понадобилось? Встроенного отображенного типа для этого не существует, поэтому давайте его объявим.

Листинг 5.19. Объявление пользовательского отображенного типа Modifiable

```
type Modifiable<T> = {
    -readonly[P in keyof T]: T[P];
};
```

Знак «минус» перед квалификатором `readonly` удаляет его из всех свойств заданного типа. Теперь вы можете удалить ограничение `readonly` из всех свойств, применив отображенный тип `Modifiable`.

Листинг 5.20. Применение отображенного типа Modifiable

```
interface Person {
    readonly name: string;
    readonly age: number;
}

const worker1: Person = {name: "John", age: 25};

worker1.age = 27;  <-- Приводит к ошибке компиляции

const worker2: Modifiable<Person> = {name: "John", age: 25};

worker2.age = 27;  <-- Здесь никаких ошибок
```

Вы можете найти этот код в песочнице здесь: http://mng.bz/yzed.

5.2.3. Другие встроенные отображенные типы

Если имя свойства в объявлении типа оканчивается модификатором `?`, это свойство является опциональным. Предположим, у нас есть следующее объявление типа `Person`:

```
interface Person {
    name: string;
    age: number;
}
```

Поскольку ни одно из имен свойств не оканчивается вопросительным знаком, передача значений для `name` и `age` является обязательной. А что, если вам нужен тип, имеющий те же свойства, что и `Person`, но все они при этом должны быть опциональными? Для этого используется отображенный тип `Partial<T>`. Его функция отображения объявлена в lib.es5.d.ts так:

Листинг 5.21. Объявление отображенного типа Partial

```
type Partial<T> = {
    [P in keyof T]?: T[P];
};
```

Заметили вопросительный знак? По сути, мы создаем новый тип, добавляя вопросительный знак каждому имени свойства данного типа. При этом отображенный тип `Partial` делает все свойства данного типа опциональными.

На рис. 5.3 приведен скриншот, сделанный в момент наведения курсора на объявление переменной `worker1`. Он показывает сообщение об ошибке, так как переменная `worker1` имеет тип `Person`, где все свойства являются обязательными, но при этом для `age` значение передано не было. В то же время при инициализации `worker2` с помощью того же объекта ошибок не возникает, потому что тип этой переменной — `Partial<Person>`, следовательно, все его свойства опциональны.

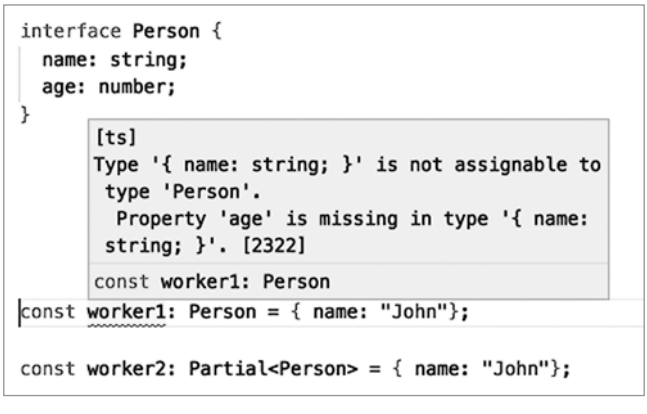


Рис. 5.3. Применение типа Partial

Теперь вы можете сделать все свойства типа опциональными, но возможно ли обратное? Взять тип, в котором ряд свойств были объявлены опциональными, и сделать их обязательными? Конечно! Поможет отображенный тип `Required`, который объявляется так:

```
type Required<T> = {
    [P in keyof T]-?: T[P];
};
```

`-?` означает удаление модификатора `?`.

На рис. 5.4 приведен скриншот, сделанный в момент наведения курсора на объявление переменной `worker2`. В основном типе `Person` свойства `age` и `name`

являются опциональными, но в отображенном типе `Required<Person>` они уже обязательны. Отсюда и появляется ошибка об упущенном свойстве `age`.

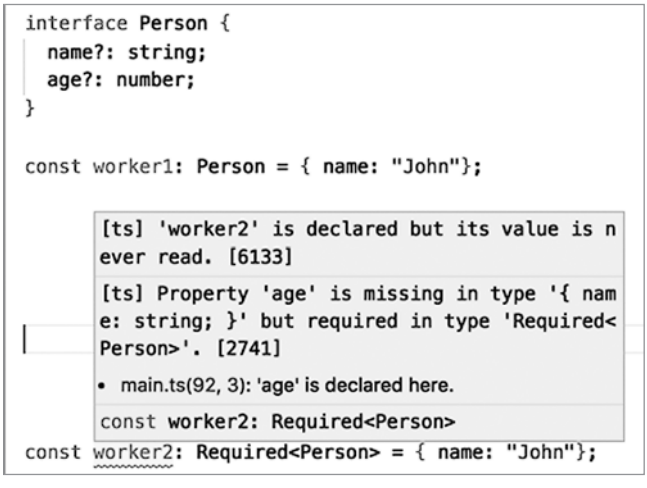


Рис. 5.4. Применение типа Required

СОВЕТ Тип `Required` был введен в TypeScript 2.8. Если ваша IDE не распознает его, убедитесь, что она использует правильную версию языковой службы TS. В Visual Studio Code увидеть версию можно в нижнем правом углу. Щелкните по ней для изменения на более новую, если таковая у вас установлена.

К заданному типу вы можете применить более одного отображенного типа. В следующем листинге мы применяем `Readonly` и `Partial` к типу `Person`. Первый будет делать каждое свойство только для чтения, а второй — опциональным.

Листинг 5.22. Применение более одного отображенного типа

```
interface Person {
    name: string;
    age: number;
}

const worker1: Readonly<Partial<Person>>
                        = {name: "John"};

worker1.name = "Mary"; // Ошибка компиляции
```

worker1 по-прежнему Person, но его свойства только для чтения и опциональны

Инициализирует имя свойства, но не опциональное age

name только для чтения и может инициализироваться только один раз

TypeScript предлагает еще один полезный отображенный тип — `Pick`. Он позволяет вам объявлять новый тип, выбрав подмножество свойств заданного типа. Его функция преобразования выглядит так:

```
type Pick<T, K extends keyof T> = {
    [P in K]: T[P];
};
```

Первый аргумент ожидает произвольный тип `T`, а второй — подмножество `K` со свойствами из `T`. Вы можете прочитать это выражение так: «Из `T` взять набор свойств, чьи ключи находятся в объединении `K`». Следующий листинг показывает тип `Person`, имеющий три свойства. При помощи `Pick` мы объявляем отображенный тип `PersonNameAddress`, который имеет два строковых свойства: `name` и `address`.

Листинг 5.23. Использование отображенного типа Pick

```
interface Person {
    name: string;
    age: number;
    address: string;
}

type PersonNameAddress<T, K> = Pick<Person, 'name' | 'address' >;
```

Объявляет тип Person с тремя свойствами

Объявляет отображенный тип PersonNameAddress с двумя свойствами

Вы можете подумать: «Обсуждать отображенные типы хорошо, и они выглядят весьма полезными, но разве мне нужно знать, как реализовывать свои собственные?» Да. И вы увидите примеры использования отображенного типа `Pick` для определения пользовательского отображенного типа вскоре на рис. 5.5, а также позже в главе 10 во врезке «Примеры условных и отображенных типов».

Отображенные типы позволяют модифицировать существующие, но TS предлагает и другой способ изменения типа, уже на основе некоего условия. Этот способ мы рассмотрим дальше.

5.3. УСЛОВНЫЕ ТИПЫ

При использовании отображенных типов функция преобразования всегда одинакова, но в случае с условными преобразование зависит от конкретного условия.

Многие языки программирования, включая JavaScript и TypeScript, поддерживают условные (тернарные) выражения:

```
a < b ? doSomething() : doSomethingElse()
```

Если значение `a` меньше, чем `b`, эта строка кода вызывает функцию `doSomething()`, в противном случае она вызывает `doSomethingElse()`. Это выражение *проверяет значения* и выполняет соответствующий условию код. Условный тип аналогичным образом использует условное выражение, но *проверяет тип этого выражения*.

Условный тип всегда объявляется по следующей форме:

```
T extends U ? X : Y
```

Здесь `extends U` означает «наследует от `U`» или «является `U`». Как и в случае с обобщениями, эти буквы могут представлять любые типы.

В ООП объявление `class Cat extends Animal` означает, что `Cat` является `Animal` и `Cat` имеет те же (плюс, возможно, дополнительные) возможности `Animal`.

Другими словами, можно сказать, что кошка (`cat`) — это более конкретная версия животного (`Animal`).

Но может ли объект `Animal` быть присвоен переменной типа `Cat`? Нет, не может. Каждая кошка относится к животным, но не каждое животное является кошкой.

Схожим образом выражение `T extends U ?` проверяет, может ли `T` быть присвоен `U`. Если это верно, мы используем тип `X`, в противном же случае тип `Y`. Выражение `T extends U` означает, что значение типа `T` может быть присвоено переменной типа `U`.

> **СОВЕТ** Мы уже упоминали *совместимые* типы при обсуждении листингов 2.20 и 3.6. Вернитесь к этим обсуждениям, чтобы освежить в памяти подробности.

Рассмотрим функцию, которая может иметь разные возвращаемые типы в зависимости от некоего условия. Например, мы хотим написать функцию `getProducts()`, которая должна возвращать тип `Product`, если в качестве аргумента был передан численный `ID` продукта. В противном случае эта функция будет возвращать массив `Product()`. При использовании условных типов сигнатура этой функции может выглядеть так:

```
function getProducts<T>(id?: T):
    T extends number ? Product : Product[]
```

Если тип аргумента будет `number`, то возвращаемый тип этой функции будет `Product`, в противном случае `Product[]`.

Следующий листинг включает пример реализации такой функции. Если переданный опциональный `id` будет числом, мы вернем один продукт; если же нет, мы вернем массив из двух продуктов.

Листинг 5.24. Функция с условным возвращаемым типом

```
class Product {
    id: number;
  }
```

```
const getProducts = function<T>(id?: T):
            T extends number ? Product : Product[] {

        if (typeof id === 'number') {
          return { id: 123 } as any;
        } else {
          return [{ id: 123 }, {id: 567}] as any;
        }
}

const result1 = getProducts(123);

const result2 = getProducts();
```

Объявляет условный возвращаемый тип

Проверяет тип переданного аргумента

Вызывает функцию с численным аргументом

Вызывает функцию без аргументов

Тип переменной `result` — это `Product`, а тип `result2` — это `Product[]`. Вы можете увидеть это сами, наведя курсор на эти переменные в уже хорошо знакомой песочнице: http://mng.bz/MOqB.

В листинге 5.24 мы использовали утверждение типа `as`, которое сообщает TS, что он не должен выводить тип, так как вы знаете о нем лучше. `as any` означает: «TypeScript, не ругайся на этот тип». Проблема в том, что суженный тип `id` не выбирается условным типом, поэтому `функция` не может вычислить условие и сузить возвращаемый тип до `Product`.

В разделе 3.1.6 был аналогичный пример, где мы реализовывали функцию `getProducts()`, которая могла быть вызвана с аргументом или без него (см. листинг 3.9). Там `getProducts()` была реализована с использованием перегрузки метода.

Условные типы могут использоваться во многих сценариях. Рассмотрим другой случай использования.

В TS есть встроенные условные типы вроде `Exclude`, который позволяет вам исключать указанные типы. Объявлен `Exclude` в файле lib.es5.d.ts следующим образом:

```
type Exclude<T, U> = T extends U ? never : T;
```

Этот тип исключает типы, совместимые с `U`. Обратите внимание на использование типа `never`, который означает «Это не должно существовать никогда; отфильтровать». Если тип `T` не может быть присвоен `U`, тогда сохраните его.

Предположим, у нас есть класс `Person` и мы используем его в нескольких местах приложения для популярного ТВ-шоу *The Voice*:

```
class Person {
    id: number;
    name: string;
    age: number;
}
```

Все вокалисты должны пройти слепое прослушивание, где жюри не может их видеть и ничего о них не знает. Для таких прослушиваний мы хотим создать другой тип, который будет таким же, как `Person`, но не будет иметь имен (`name`) или возрастов (`age`). Иначе говоря, мы хотим исключить свойства `name` и `age` из типа `Person`.

Из предыдущего раздела вы помните, что оператор `keyof` может предоставить список всех свойств типа. Таким образом, следующий тип будет содержать свойства `T`, за исключением тех, которые принадлежат заданному типу `K`:

```
type RemoveProps<T, K> = Exclude<keyof T, K>;
```

Создадим новый тип, который будет таким же, как `Person`, но без `name` и `age`:

```
type RemainingProps = RemoveProps<Person, 'name' | 'age'>;
```

В этом примере тип `K` представлен объединением `'name' | 'age'`, а тип `RemainingProps` представляет объединение оставшихся свойств, к которым в нашем примере относится лишь `id`.

Теперь можно построить новый тип, содержащий только `RemainingProperties`, используя отображенный тип `Pick` (ранее продемонстрированный в листинге 5.24).

```
type RemainingProps = RemoveProps<Person, 'name' | 'age'>;
type PersonBlindAuditions = Pick<Person, RemainingProps>;
```

На рис. 5.5 приведен скриншот, сделанный в момент наведения курсора на тип `PersonBlindAudition`. Вы можете найти его в песочнице здесь: http://mng.bz/adGm.

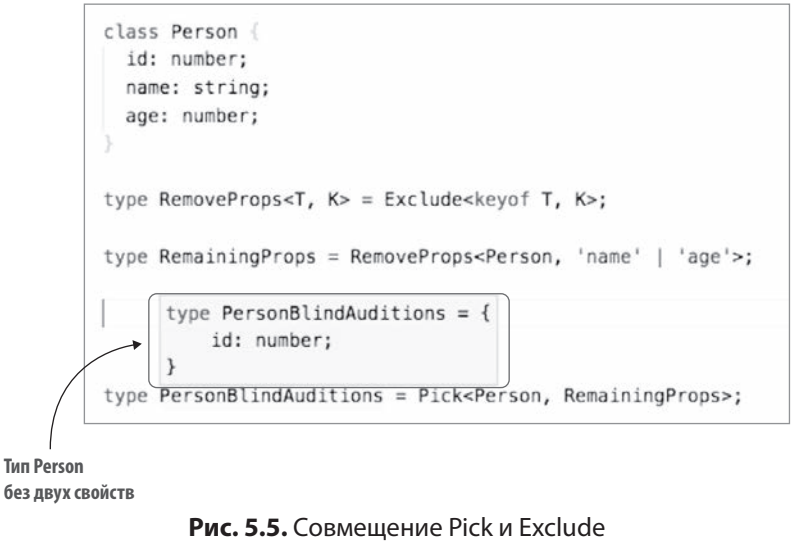


Рис. 5.5. Совмещение Pick и Exclude

Возникает вопрос: не будет ли легче создать отдельный тип `PersonBlindAudition`, имеющий только свойство `id`? Это предположение верно для текущего простого случая, где тип `Person` имеет всего три свойства. Но `Person` может быть описан, к примеру, тридцатью свойствами, и может возникнуть необходимость использовать его как основной класс, создавая на его основе более описательные условные типы.

Даже с нашим классом, имеющим всего три свойства, использование условных типов может принести свои выгоды. Что, если в недалеком будущем разработчик решит заменить в классе `Person` свойство `name` на `firstName` и `lastName`? Если использовать условный тип `PersonBlindAuditions`, его объявление начнет вызывать ошибку компиляции, которую вы сразу исправите. Но если вы не объявите `PersonBlindAuditions` как условный тип и просто создадите независимый класс `PersonBlindAuditionsIndie`, тогда разработчику, переименовывающему свойства `Person`, придется *не забыть* продублировать изменения в классе `PersonBlindAuditionsIndie`.

Кроме того, тип `RemoveProperties` является обобщенным. Можно использовать его для удаления любых свойств из любых типов.

5.3.1. Ключевое слово infer

Следующей задачей будет попытка найти возвращаемый тип функции и заместить его другим. Предположим, есть интерфейс, объявляющий некие свойства и методы, и нужно обернуть каждый из этих методов в `Promise`, чтобы они выполнялись асинхронно. Для простоты мы рассмотрим интерфейс `SyncService`, объявляющий одно свойство — `baseUrl` и один метод — `getA()`:

```
interface SyncService {
baseUrl: string;
getA(): string;
}
```

Нам нечего делать с самим свойством `baseUrl`, но мы хотим обернуть в промис-метод `getA()`. Вот наши задачи:

- Как разграничить свойства и методы?
- Как обнаружить изначальный возвращаемый тип метода, прежде чем оборачивать его в `Promise`?
- Как сохранить существующие типы аргументов методов?

Поскольку нам нужно разграничить свойства и методы, мы используем условные типы; отображенные же типы помогут нам при изменении сигнатур методов. Наша цель создать тип `Promisify` и применить его к `SyncService`. В таком случае

реализация `getA()` будет вынуждена вернуть `Promise`. Должна быть возможность написать код как в следующем листинге:

Листинг 5.25. Оборачивание в промис синхронных методов

```
class AsyncService  ◄──  Отображает SyncService в Promisify
  implements Promisify<SyncService> {
      baseUrl: string;  ◄──  Нет нужды изменять свойства из SyncService

      getA(): Promise<string> {      ◄──┐  Оригинальный возвращаемый тип
          return Promise.resolve('');   │  должен быть обернут в Promise
      }
}
```

Мы хотим объявить новый отображенный тип `Promisify`, который будет перебирать все свойства заданного типа `T` и преобразовывать их сигнатуры в асинхронные. Преобразование будет производить условный тип, чье условие будет гласить, что тип `U` (супертип `T`) должен быть функцией, которая может получать любое число аргументов любых типов и возвращать любое значение:

```
T extends (...args: any[]) => any ?
```

После знака вопроса вы передаете тип для использования на случай, если `T` окажется функцией; другой тип вы передаете после запятой на случай, если `T` функцией не окажется (не показано здесь).

Тип `T` должен быть совместим с типом, который выглядит как сигнатура функции. Если переданный тип является функцией, мы хотим обернуть возвращение этой функции в `Promise`. Проблема в том, что если мы используем тип `any`, то утратим информацию типа для аргументов функции, так же как и возвращаемый тип.

Давайте предположим, что обобщенный тип `R` представляет возвращаемый тип функции. Затем мы сможем использовать ключевое слово `infer` с переменной `R`:

```
T extends (...args: any[]) => infer R ?
```

Написав `infer R`, мы проинструктировали TS проверять переданный конкретный возвращаемый тип (например, `string` для метода `getA()`) и замещать `infer R` этим конкретным типом. Аналогично мы можем заменить тип `any[]` в аргументах функции на `infer A`:

```
T extends (...args: infer A) => infer R ?
```

Теперь мы можем объявить наш условный тип так:

```
type ReturnPromise<T> =
    T extends (...args: infer A) => infer R ? (...args: A) => Promise<R> : T;
```

Это сообщит TS следующее: «Если конкретный тип для `T` окажется функцией, оберни его возвращаемый тип: `Promise<R>`. В противном случае просто сохрани его тип `T`». Условный тип `ReturnPromise<T>` может быть применен к любому типу, и если мы захотим перечислить все свойства класса, интерфейса и т. д., то просто можем использовать оператор `keyof` для получения всех этих свойств.

Если вы прочитали раздел 5.2, посвященный отображенным типам, то синтаксис следующего фрагмента кода должен быть вам знаком:

```
type Promisify<T> = {
    [P in keyof T]: ReturnPromise<T[P]>;
};
```

Отображенный тип `Promisify<T>` будет итерировать свойства `T` и применять к ним условный тип `ReturnPromise`. В нашем примере мы сделаем `Promisify<SyncService>`, который не будет ничего делать со свойством `baseUrl`, но будет изменять возвращаемый тип `getA()` на `Promise<string>`.

На рис. 5.6 показан весь сценарий, который можно найти здесь: http://mng.bz/gVWv.

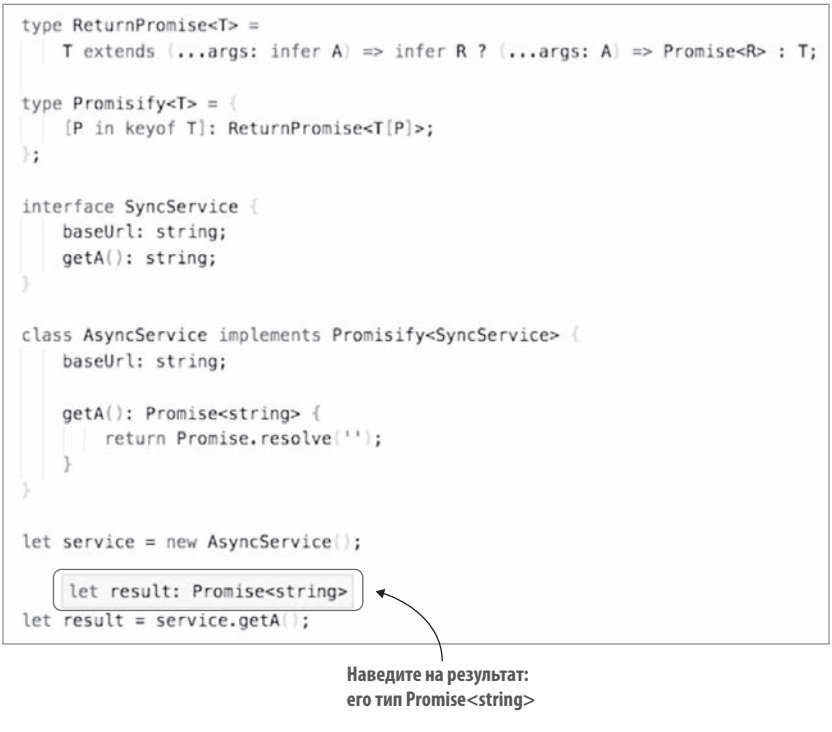


Рис. 5.6. Совмещение условных и отображенных типов

ИТОГИ

- Используя декораторы, вы можете добавлять метаданные в класс, функцию, свойство или параметр.
- Декораторы позволяют изменять объявление типа, а также поведение класса, метода, свойства или параметра. Даже если вы не будете писать свои декораторы, важно понимать их применение, если один из фреймворков их использует.
- Вы можете создавать тип на основе другого типа.
- Отображенные типы позволяют вам создавать приложения, имеющие ограниченное число основных типов, но множество производных, созданных на их основе.
- Условные типы позволяют вам отложить принятие решения о том, какой тип использовать; решение принимается при выполнении, в зависимости от заданного условия.
- Эти возможности языка непросты для понимания, но они показывают его мощь. При рассмотрении кода блокчейн-приложения в главе 10 вы увидите практическую пользу отображенных и условных типов.

6 Инструменты

В этой главе:

- ✓ Отладка TypeScript кода с помощью карт кода.
- ✓ Роль линтеров.
- ✓ Компиляция и связка приложений TypeScript с помощью Webpack.
- ✓ Компиляция приложений TypeScript с помощью Babel.
- ✓ Как компилировать TypeScript с помощью Babel и связывать с помощью Webpack.

TypeScript является одним из любимых языков среди программистов. Да, разработчики обожают его синтаксис, но главная причина популярности — сопутствующий инструментарий. Люди, работающие с ним, ценят автоподстановку — эти волнистые линии, указывающие на ошибки в процессе печати, а также рефакторинг, предлагаемый IDE.

Самое замечательное, что большая часть этих возможностей реализована командой TS, а не разработчиками IDE. Вы увидите все ту же автоподстановку и сообщения об ошибках в онлайн-песочнице TS, в VS Code или в WebStorm. Когда вы устанавливаете TypeScript, его директория bin включает в себя два файла: tsc и tsserver. Последний является языковой службой TS, которую IDE использует для поддержки всех этих продуктивных возможностей. Когда вы печатаете TS-код, IDE связывается со службой tsserver, которая компилирует код в памяти.

С помощью файлов карт кода можно отлаживать TS-код прямо в браузере. Также существуют инструменты, называемые линтерами. Они позволяют вам обеспечивать соблюдение стилей кода в организации.

Файлы объявлений типов (.d.ts) позволяют tsserver предлагать контекстную помощь, показывая сигнатуры доступных функций или свойств объектов. В открытом доступе находятся тысячи файлов объявлений типов для популярных JS-библиотек. С их помощью вы можете быть более продуктивными, даже работая с кодом, написанным не в TypeScript.

Если сложить все эти удобства воедино, то становится ясно, почему людям нравится TS. Но наличия одного лишь крутого компилятора недостаточно для реальных проектов, которые состоят из разнообразного набора компонентов вроде JS-кода, CSS, изображений и т. д. Рассмотрим важные инструменты для современной веб-разработки: WebPack бандлер и Babel. Помимо этого, мы вкратце затронем новейшие инструменты ncc и Deno.

6.1. КАРТЫ КОДА

Код, написанный в TypeScript, компилируется в JavaScript, который выполняется в браузере или отдельном движке JavaScript. Для отладки программы вам нужно предоставить отладчику исходный код, но у нас есть две версии исходного кода: выполняемый код в виде JavaScript и оригинальный в TypeScript. Нас же интересует отладка именно TS-версии, в чем нам помогут файлы карт кода.

Файлы карт кода имеют расширение .map и содержат данные в формате JSON, отображающие соответствующий код сгенерированного JavaScript в оригинальный язык, которым в нашем случае является TypeScript. Если вы решите отладить JS-программу, написанную в TS, просто попросите браузер загрузить файлы карт кода, сгенерированные в процессе компиляции, и вы сможете расставить точки останова в TS-коде, даже несмотря на то что движок выполняет код JS.

Давайте возьмем простую программу TypeScript, показанную в следующем листинге, и скомпилируем ее, включив опцию `--sourceMap`.

Листинг 6.1. Файл greeter.ts

```
class Greeter {

    static sayHello(name: string) {
        console.log (`Hello ${name}`);  <-- Печатает имя в консоли
    }
}

Greeter.sayHello('John');  <-- Вызывает метод sayHello()
```

Давайте скомпилируем этот файл, сгенерировав карту кода:

```
tsc greeter.ts --sourceMap true
```

По завершении компиляции вы увидите файлы greeter.js и greeter.js.map. Последний и является картой кода, фрагмент которой показан в следующем листинге:

Листинг 6.2. Сгенерированный файл карты кода, greeter.js.map

```
{"file":"greeter.js",
    "sources":["greeter.ts"],
    "mappings":"AAAA;IAAA;IAMA,CAAC;IAJU,gBAAQ,..."
}
```

Этот файл не предполагает чтения человеком, но вы можете видеть, что он имеет свойство `file` с именем сгенерированного файла JS и свойство `sources` с именем исходного файла TS. Свойство `mappings` содержит отображения фрагментов кода в JS- и TS-файлах.

Как движок JavaScript догадался, что имя файла, содержащего отображение, — это greeter.js.map? А догадываться ему и не пришлось. Компилятор TS просто добавляет в конец сгенерированного файла greeter.js следующую строку:

```
//# sourceMappingURL=greeter.js.map
```

Запустим наше приложение greeter в браузере и посмотрим, сможем ли мы произвести отладку его TS-кода. Для начала мы создадим HTML-файл, загружающий greeter.js

Листинг 6.3. Файл index.html, загружающий greeter.js

```
<!DOCTYPE html>
<html>
    <body>
        <script src="greeter.js"/>  ◄——  Загружает здесь файл JavaScript, а не TypeScript
    </body>
</html>
```

Далее нужен веб-сервер, который будет обслуживать предыдущий HTML-документ в браузере. Вы можете загрузить и установить удобный пакет live-server так:

```
npm install -g live-server
```

В завершение запустите этот сервер в окне терминала из директории, в которой расположены файлы greeter.

```
live-server
```

Он откроет браузер Chrome на localhost:8080 и загрузит код из index.html (листинг 6.3). Вы увидите пустую страницу. Откройте Chrome Dev Tools (Инструменты разработчика) во вкладке Sources (Источники) и выберите файл greeter.ts. В исходном коде щелкните слева от строки 5, чтобы установить на ней точку останова. Панель Sources должна быть схожа с изображенной на рис. 6.1.

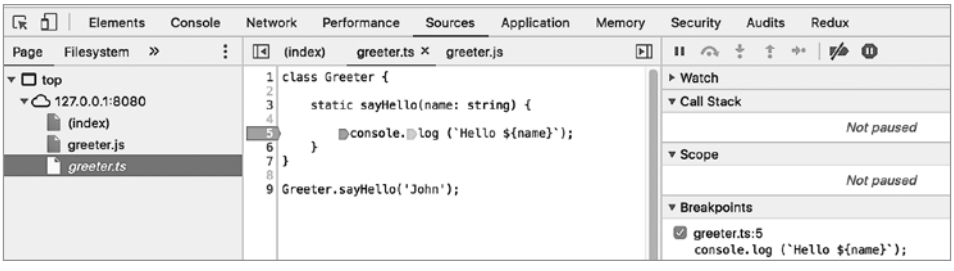


Рис. 6.1. Установка точки останова в TypeScript-коде

Обновите страницу, и выполнение этого кода прервется на обозначенной точке останова, как показано на рис. 6.2. Теперь вы можете использовать знакомое управление отладкой вроде шага вперед, шага внутрь и т. д.

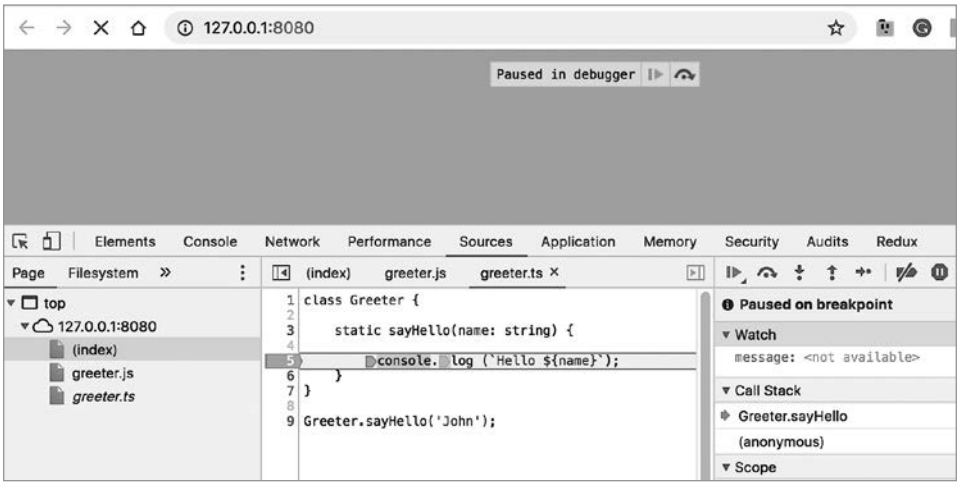


Рис. 6.2. Выполнение остановлено в отладчике

Теперь покажем вам опцию компиляции `--inlineSources`, которая влияет на процесс генерации карт кода. С этой опцией файл .js.map будет дополнительно

включать исходный код TS вашего приложения. Попробуйте скомпилировать файл greeter.js следующим образом:

```
tsc greeter.ts --sourceMap true --inlineSources true
```

> **ПРИМЕЧАНИЕ** Хотя каждая IDE и имеет свой собственный отладчик, мы предпочитаем отлаживать исходный код в Chrome Dev Tools. Здесь даже можно производить отладку кода, выполняемого как самостоятельное приложение Node.js. Как это делается, мы расскажем во врезке «Отладка Node.js кода в браузере» в конце раздела 10.6.1. (Мы рассматриваем отладку Node.js-приложений в главе 10, где вы сможете увидеть использование Node.js сервера в блокчейн-приложении.)

Он по-прежнему производит файлы greeter.js и greeter.js.map, но последний теперь также включает код из greeter.js. Эта опция устраняет необходимость развертывания отдельных файлов .ts на вашем веб-сервере, но при этом вы по-прежнему можете производить отладку TS-кода.

> **ПРИМЕЧАНИЕ** Развертывание файлов с расширением .js.map на продакшен-серверах не увеличивает размер кода, загружаемого браузером. Браузер загружает карты кода, только если пользователь открывает инструменты разработчика. Единственная причина не развертывать карты кода на продакшен-сервере — это желание оградить исходный код от чтения пользователями.

6.2. ЛИНТЕР TSLINT

Линтеры — это инструменты, проверяющие и обеспечивающие следование стилю написания кода. Например, вы можете обеспечить, чтобы все строковые значения указывались в одинарных кавычках, или запретить необязательные скобки. Подобные ограничения являются правилами, которые вы можете конфигурировать в текстовых файлах.

JavaScript-разработчики используют несколько линтеров: JsLint, JSHint и ESLint. TypeScript-разработчики используют TSLint, являющийся открытым проектом, поддерживаемым Palantir (https://palantir.github.io/tslint). Также существует план слияния TSLint и ESLint, чтобы обеспечить комплексный процесс соблюдения стандартов кода, — вы можете прочитать об этом подробнее в блоге Palantir («TSLint in 2019» по адресу http://mng.bz/eD9V). Как только это слияние будет завершено, Javascript- и TypeScript-разработчики будут использовать ESLint (https://eslint.org). Так как многие TS-команды продолжают использовать TSLint, мы познакомим вас с базовыми особенностями этого инструмента. Чтобы начать использовать TSLint, нужно установить его в свой проект. Давайте начнем с нуля. Создайте новый каталог, откройте в нем терминал и инициализируйте новый проект npm, используя следующую команду:

```
npm init -y
```

Опция `-y` примет все предустановленные настройки в процессе создания пакетного файла .json.

Затем установите TypeScript и ts-lint:

```
npm install typescript tslint
```

Эта команда создаст каталог node_modules и установит в него TS и tslint. Исполняемый файл tslint будет размещен в директории node_modules/.bin.

Теперь создайте файл конфигурации tslint.json, используя следующую команду:

```
./node_modules/.bin/tslint --init
```

> **СОВЕТ** Начиная с версии 5.2, npm предоставляет командную строку npx, которая может запускать исполняемые файлы из node_modules/.bin. Например, так: `npx tslint --init`. Подробнее об этой полезной команде вы можете прочитать на сайте npm: www.npmjs.com/package/npx.

Эта команда создаст файл tslint.json со следующим содержимым:

Листинг 6.4. Сгенерированный файл tslint.json

```
{
    "defaultSeverity": "error",
    "extends": [
        "tslint:recommended"   <-- Использует рекомендованные правила
    ],
    "jsRules": {},
    "rules": {},   <-- Здесь помещаются правила пользователя
    "rulesDirectory": []   <-- Опциональная директория с пользовательскими правилами
}
```

Этот файл конфигурации утверждает, что tslint должен расширить предустановленные рекомендуемые правила, которые вы можете найти в файле node_modules/tslint/lib/configs/recommended.js. На рис. 6.3 показан снимок части файла recommended.js.

Правило в строках 126–128 выглядит так:

```
quotemark: {
    options: ["double", "avoid-escape"],
},
```

Оно обязывает заключать строки в двойные кавычки. Правило `"avoid-escape"` позволяет использовать «другие» кавычки в случаях, где обычно потребовалось бы экранирование. Для двойных кавычек «другие» будет означать одинарные кавычки.

```
117     "ordered-imports": {
118         options: {
119             "import-sources-order": "case-insensitive",
120             "module-source-path": "full",
121             "named-imports-order": "case-insensitive",
122         },
123     },
124     "prefer-const": true,
125     "prefer-for-of": true,
126     quotemark: {
127         options: ["double", "avoid-escape"],
128     },
129     radix: true,
130     semicolon: { options: ["always"] },
131     "space-before-function-paren": {
132         options: {
133             anonymous: "never",
134             asyncArrow: "always",
135             constructor: "never",
136             method: "never",
137             named: "never",
138         },
139     },
140     "trailing-comma": {
141         options: {
142             esSpecCompliant: true,
143             multiline: "always",
144             singleline: "never",
145         },
```

Рис. 6.3. Фрагмент из recommended.js

На рис. 6.4 приведен скриншот IDE WebStorm, показывающий ошибку листинга в первой строке. Несмотря на то что использование одинарных кавычек вокруг строк абсолютно нормально для компилятора TS, правило quotemark утверждает, что в этом проекте вы должны использовать двойные.

```
1   const customerName = 'Mary';
2
3   ~                    TSLint: ' should be " (quotemark)
4
5   const greeting = 'Hello "World"';
```

Рис. 6.4. TSLint сообщает об ошибках

СОВЕТ Наведите на ошибку TSLint, и IDE предложит вам исправление.

Чтобы избежать использования символов экранирования для строки внутри другой строки на рис. 6.4 (строка 5), мы заключили слово `"World"` в двойные кавычки, и линтер не ругался, так как включена опция avoid-escape.

СОВЕТ Чтобы активировать TSLint в VS Code, установите его расширение. Для этого щелкните по иконке расширения на боковой панели и найдите TSLint в маркетплейсе. Убедитесь, что VS Code использует текущую версию TypeScript (номер версии показан в правом нижнем углу строки состояния). Изменение версии TS описано в документации VS Code (http://mng.bz.pvnK).

На рис. 6.4 вы можете заметить короткую волнистую линию в пустой строке 3. Если вы наведете указатель мыши на эту волнистую линию, то увидите еще одну ошибку линтера: «TSLint: Consecutive blank lines are forbidden (no consecutive blank lines)». (Последовательные пустые строки запрещены). Имя правила показано в скобках, в файле recommended.js оно представлено следующей строкой:

```
"no-consecutive-blank-lines": true,
```

ПРИМЕЧАНИЕ Основные правила TSLint доступны на официальном сайте по ссылке https://palantir.github.io/tslint/rules/.

Давайте переопределим правило no-consecutive-blank-lines в файле tslint.json, как показано в листинге 6.5. Для этого мы добавим в него правило, разрешающее последовательные пустые строки.

Листинг 6.5. Переопределение правила в tslint.json

```
{
    "defaultSeverity": "error",
    "extends": [
        "tslint:recommended"
    ],
    "jsRules": {},
    "rules": {"no-consecutive-blank-lines": false},   <-- Мы добавили эту строку
    "rulesDirectory": []
}
```

Присвоив значение `false` правилу no-consecutive-blank-lines, мы переопределили его значение из файла recommended.js. Теперь та маленькая волнистая линия на рис. 6.4 исчезнет.

Можете переопределять рекомендованные правила или добавлять новые, соответствующие вашему стилю написания кода или тому, который принят в вашей команде.

6.3. СВЯЗЫВАНИЕ КОДА С ПОМОЩЬЮ WEBPACK

Когда браузер делает запрос на сервер, он получает HTML-документы, которые могут включать дополнительные файлы вроде CSS, изображений, видео и т. д.

В части 2 этой книги мы будем работать с примерами блокчейн-приложения, имеющего несколько файлов. Часто приходится использовать один из популярных JavaScript-фреймворков, который может добавлять в приложение сотни файлов. Если все эти файлы будут развертываться по отдельности, браузеру потребуется сделать сотни запросов для их загрузки. Размер вашего приложения может достигать нескольких мегабайтов.

Реальные приложения состоят из сотен и даже тысяч файлов. Мы же в процессе развертывания хотим их минимизировать, оптимизировать, а также связать вместе. На рис. 6.5 показано, как различные файлы передаются в WebPack, который на выходе производит меньшее их число для развертывания.

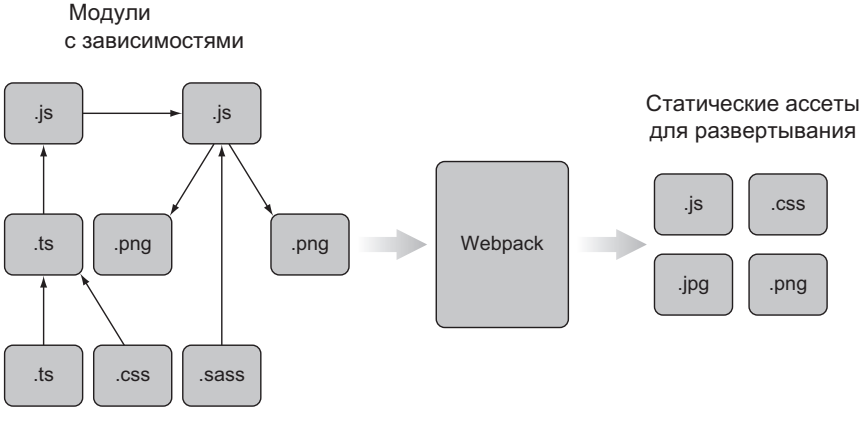


Рис. 6.5. Связывание исходников с помощью Webpack

Связывание нескольких файлов в один может повысить производительность и ускорить загрузки. Технически вы можете указать один файл вывода в tsconfig.json, и tsc сможет поместить сгенерированный код нескольких файлов .ts в один итоговый файл. Но это сработает, только если опция компилятора module определена как System или AMD; выводимый файл, сгенерированный бандлером, не будет включать сторонние зависимости вроде библиотек JavaScript.

Помимо этого, нужна возможность по-разному настроить сборки разработки и продакшена. Для продакшена вы добавите оптимизацию и минимизацию, в то время как для разработки будете просто связывать файлы вместе.

Несколько лет назад для управления процессом сборки JS-разработчики чаще всего использовали менеджеры задач вроде Grunt и Gulp. Сегодня же они используют такие бандлеры, как WebPack, Rollup и Broweserify. С помощью плагинов бандлеры могут также поддерживать возможности компиляторов, если потребуется.

В этом разделе мы представим бандлер Webpack (https://github.com/webpack), который был создан для веб-приложений, выполняющихся в браузере. Webpack поддерживает многие типичные задачи, необходимые для подготовки сборки приложений, и требует при этом минимума настройки.

Вы можете установить Webpack либо глобально, либо локально (в поддиректорию node_modules вашего проекта). Он также имеет интерфейс командной строки (CLI) — webpack-cli. (https://github.com/webpack/webpack-cli). До версии Webpack 4 настройка проекта Webpack была непростой, но инструмент подготовки webpack-cli существенно упрощает настройку. Чтобы установить Webpack и его CLI глобально на компьютер, выполните следующую команду (`-g` означает глобально):

```
npm install webpack webpack-cli -g
```

> **ПРИМЕЧАНИЕ** Установка Webpack (или другого инструмента) глобально позволяет вам использовать его в нескольких проектах. Это круто, но в вашей компании продакшен-сборка может выполняться на выделенном компьютере с ограничениями относительно глобально устанавливаемого софта. Поэтому будем использовать Webpack и Webpack CLI, установленные локально.

Webpack, вероятно, наиболее популярный бандлер в экосистеме JavaScript, и в следующем разделе мы свяжем простое JS-приложение.

6.3.1. Связывание JavaScript с помощью Webpack

Цель этого раздела — показать, как настраивать и запускать Webpack для простых JS-приложений, чтобы вы могли увидеть, как этот бандлер работает и чего ожидать в качестве его вывода.

Исходный код, используемый в этой главе, имеет несколько проектов, и мы начнем с проекта под названием webpack-javascript. Это npm-проект, поэтому нужно установить его зависимости, выполнив следующую команду:

```
npm install
```

В этом проекте у нас небольшой JS-файл, использующий стороннюю библиотеку Chalk (www.npmjs.com/package/chalk), которая перечислена в разделе зависимостей файла package.json проекта.

Листинг 6.6. Файл package.json проекта webpack-javascript

```
{
    "name": "webpack-javascript",
    "description": "Code sample for chapter 6",
    "homepage": "https://www.manning.com/books/typescript-quickly",
```

```
    "license": "MIT",
    "scripts": {
        "bundleup": "webpack-cli"  <-- Определяет команду для запуска Webpack
    },
    "dependencies": {
        "chalk": "^2.4.1"  <-- Библиотека chalk
    },
    "devDependencies": {
        "webpack": "^4.28.3",  <-- Бандлер Webpack
        "webpack-cli": "^3.1.2"  <-- Интерфейс командной строки Webpack
    }
}
```

Пакеты Webpack расположены в разделе devDependencies, так как нам они нужны только на компьютере разработчика. Будем запускать это приложение, вводя команду `npm run bundleup`, запускающую исполняемый файл webpack-cli в директории node_modules/.bin.

Пакет Chalk закрашивает текст окна терминала в разные цвета, но его действия здесь для нас неважны. Наша цель — связать JS-код (index.js) с библиотекой. Код из index.js показан в следующем листинге.

Листинг 6.7. index.js: исходный код приложения webpack-javascript

```
const chalk = require('chalk');  <-- Загружает библиотеку
const message = 'Bundled by the Webpack';
console.log(chalk.black.bgGreenBright(message));  <-- Использует библиотеку
```

Как правило, разработчики используют файл webpack.config.js для создания пользовательских конфигураций, даже несмотря на то что это стало необязательным, начиная с версии Webpack 4. Это то место, где вы настраиваете сборку для вашего проекта. Следующий листинг показывает файл webpack.config.js для текущего проекта.

Листинг 6.8. webpack.config.js: файл конфигурации Webpack

```
const { resolve } = require('path');

module.exports = {
    entry: './src/index.js',  <-- Имя исходного файла для связки
    output: {
        filename: 'index.bundle.js',  <-- Имя итоговой связки
        path: resolve( dirname, 'dist')  <-- Путь для вывода
    },
    target: 'node',  <-- Мы запустим это приложение под Node.js; не используйте встроенный модуль Node.js
    mode: 'production'  <-- Оптимизирует размер файла итоговой связки
};
```

Чтобы создать связку Webpack, требуется знать главный модуль (точку входа) приложения, который может иметь зависимости от других модулей или сторонних библиотек. Webpack загружает модуль точки входа и создает дерево памяти из всех зависимых модулей, если таковые имеются.

В этом файле конфигурации мы используем модуль path Node.js, который может определить абсолютный путь файла с помощью переменной окружения `_dirname`. Webpack выполняется под Node, и переменная `_dirname` будет хранить директорию, где размещен исполняемый JavaScript-модуль (webpack.config.js). Фрагмент `resolve(_dirname, 'dist')` тем самым дает команду Webpack создать поддиректорию с именем dist в корне проекта, и связанное приложение будет размещено именно в этой директории dist.

> **СОВЕТ** Хранение итоговых файлов в отдельной директории позволит вам настроить систему контроля версий на исключение сгенерированных файлов. Если вы используете Git, просто добавьте директорию dist в файл .gitignore.

Значение свойства mode мы установили как `production`, чтобы Webpack минимизировал размер связки.

Начиная с Webpack 4, настройка проекта стала намного легче, поскольку были введены предустановленные режимы production и development. В режиме production размер сгенерированных связок мал, код оптимизирован для выполнения, а код, нужный только для разработки, удален из исходников. В режиме development компиляция поэтапная, и вы можете отлаживать код в браузере.

Инструмент webpack-cli позволяет связывать файлы приложения, сообщая entry, output и другие параметры прямо в командной строке. Он также может генерировать файл конфигурации для вашего проекта.

Давайте запустим Webpack, чтобы увидеть, как он будет связывать наше приложение. Вы будете запускать версию Webpack, установленную локально в директорию node_modules. В листинге 6.6 мы определили npm-команду bundleup и теперь можем запускать локально установленный webpack-cli:

```
npm run bundleup
```

Связывание будет выполнено в течение нескольких секунд, и консоль будет выглядеть аналогично изображенной на рис. 6.6.

Из этого вывода видно, что Webpack создал связку index.bundle.js и ее размер около 22 Кб. Это основной кусок (связка). В этом простом примере у нас всего одна связка, но более крупные приложения обычно разделяются на несколько модулей, и Webpack может быть настроен на сборку нескольких связок.

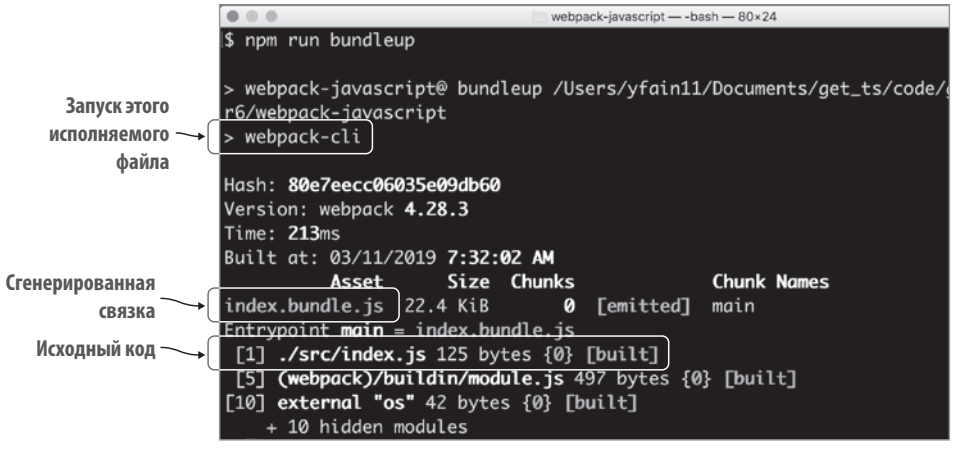


Рис. 6.6. Вывод консоли при выполнении npm run bundleup

Обратите внимание, что размер оригинального файла src/index.js был всего 125 байт. Размер связки гораздо больше, поскольку она включает не только три строки нашего файла index.js, но также библиотеку Chalk и все свои переходные зависимости. Webpack также добавляет свой код, чтобы отслеживать содержимое связки.

Измените значение свойства mode в webpack.config.js на development и перезапустите связывание. Webpack за кадром применит различные настройки конфигурации, и размер сгенерированного файла в итоге получится более 56 Кб (в противоположность 22 Кб в режиме продакшен).

Откройте файл dist/index.bundle.js в любом простом текстовом редакторе. В продакшен-версии вы просто увидите очень длинную оптимизированную строку, в то время как в связке режима разработки будет присутствовать читаемое содержимое с комментариями.

Сгенерированный файл index.bundle.js является обычным JS-файлом, и вы можете использовать его в HTML-теге `<script>` или любом другом месте, где допускаются имена файлов JS. Это конкретное приложение не предусматривает выполнения в браузере — оно для Node.js, и вы можете запустить его, используя следующую команду:

```
node dist/index.bundle.js
```

Рисунок 6.7 показывает вывод консоли, где библиотека Chalk отобразила сообщение «Bundled by the Webpack» (Связана с помощью Webpack) на ярко-зеленом фоне.

Мы запустили этот пример, используя среду выполнения Node.js, но для веб-приложений; Webpack предлагает dev-сервер, который может обслуживать ваши веб-страницы. Его потребуется установить отдельно.

```
$ node dist/index.bundle.js
Bundled by the Webpack
```

Рис. 6.7. Запуск вашей первой связки

```
npm install webpack-dev-server -D
```

После этого добавьте npm-команду start в раздел scripts файла package.json. Он будет выглядеть так:

```
"scripts": {
    "bundleup": "webpack-cli --watch",
    "start": "webpack-dev-server"
}
```

Запустите Webpack в режиме просмотра, чтобы он перестраивал связку в процессе изменения кода

Теперь давайте создадим суперпростой JS-файл index.js:

```
document.write('Hello World!');
```

Мы попросим Webpack сгенерировать связку для этого файла и сохранить ее в dist/bundle.js. Файл webpack.config.js будет схож с показанным в листинге 6.8, имея при этом два изменения: мы добавим свойство devServer и изменим режим на development.

Листинг 6.9. Добавление свойства devServer

```
const { resolve } = require('path');

module.exports = {
    entry: './src/index.js',
    output: {
        filename: 'index.bundle.js',
        path: resolve( dirname, 'dist')
    },
    target: 'node',
    mode: 'development',
    devServer: {
      contentBase: '.'
    }
};
```

Настройка Webpack для dev-режима

Добавление раздела для webpack-dev-server

В devServer вы можете настроить опции, которые позволяет webpack-dev-server, в командной строке (см. документацию Webpack на https://webpack.js.org/configuration/

dev-server/). Мы используем `contentBase`, чтобы указать, что должны обслуживаться файлы из текущей директории.

Соответственно, наш HTML-файл index.html может обратиться к index.bundle.js.

Листинг 6.10. Файл index.html, загружающий связанный JavaScript

```
<!DOCTYPE html>
<html>
<body>
    <script src="dist/index.bundle.js"></script>
</body>
</html>
```

Теперь мы готовы создать связку и запустить веб-сервер. Первым идет создание связки:

```
npm run bundleup
```

Чтобы запустить сервер, используя webpack-dev-server, просто выполните следующую команду:

```
npm start
```

Вы можете найти это приложение в директории webpack-devserver. Откройте браузер на localhost:8080, и он поприветствует мир, как показано на рис. 6.8.

Когда вы обслуживаете приложение с помощью webpack-dev-server, он будет запускаться через порт по умолчанию 8080. Так как мы запустили Webpack в режиме просмотра, он будет перекомпилировать связку при каждом изменении кода.

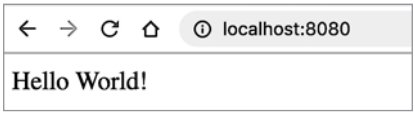


Рис. 6.8. Браузер, отображающий документ index.html

Теперь, когда вы увидели, как связывать чистый JavaScript-проект, давайте посмотрим, как использовать Webpack с TS-кодом.

6.3.2. Связывание TypeScript с помощью Webpack

Исходный код, использующийся в этой главе, имеет несколько проектов, и в этом разделе мы будем работать с одним из них в директории webpack-typescript. Этот проект почти такой же, как и предыдущий. Он включает трехстрочный

TS-файл index.ts (в предыдущем разделе это был index.js), использующий ту же JS-библиотеку Chalk.

Давайте подчеркнем отличия, начиная с файла index.ts.

Листинг 6.11. index.ts: исходный код приложения webpack-typescript

Библиотека Chalk явно предоставляет экспорт по умолчанию. Поэтому вместо написания `import * as chalk from 'chalk'` мы написали `import chalk from chalk`. Нам не пришлось явно объявлять тип `message`, но мы хотели сделать очевидным тот факт, что это код TypeScript.

В этом проекте файл package.json имеет две дополнительные строки в разделе `devDependencies` (по сравнению с листингом 6.6). Мы добавили `ts-loader` и `typescript`.

Листинг 6.12. Раздел devDependencies в package.json

```
"devDependencies": {
    "ts-loader": "^5.3.2",      <-- Добавлен загрузчик TypeScript
    "typescript": "^3.2.2",     <-- Добавлен компилятор TypeScript
    "webpack": "^4.28.3",
    "webpack-cli": "^3.1.2"
}
```

Как правило, инструменты автоматизации сборки предоставляют разработчикам способ указывать дополнительные задания, которые требуется выполнить в процессе этой сборки, а Webpack предлагает *загрузчики* и *плагины*, позволяющие сборку настраивать. Загрузчики одновременно могут предварительно обрабатывать только один файл, в то время как плагины могут работать сразу с группой файлов.

> **СОВЕТ** Список загрузчиков можно найти в документации Webpack на Github по адресу http://mng.bz/U0Yv.

Загрузчики — это преобразователи, которые в качестве ввода получают исходный файл, а на выходе производят другой файл (в память или на диск). Например, json-loader получает файл и считывает его как JSON. Для компиляции TS в JS ts-loader использует компилятор TS, что объясняет, почему мы добавили его в раздел devDependencies файла package.json. Компилятор использует tsconfig.json, содержащий следующее:

Листинг 6.13. tsconfig.json: настройки конфигурации компилятора

```
{
    "compilerOptions": {
        "target": "es2018",      <-- Устанавливает компиляцию в JS синтаксис ECMAScript 2018
        "moduleResolution": "node"   <-- Использует разрешение модулей Node.js
    }
}
```

Опция `moduleResolution` сообщает TS, как разрешать модули, если код включает инструкции импорта. Если ваше приложение включает инструкцию `import { a } from "moduleA"`, tsc должен знать, где искать `moduleA`.

Существует два подхода для разрешения модулей: `Classic` и `Node`. В подходе Classic tsc будет искать определение `moduleA` в файле moduleA.ts, а также в файле определений типов moduleA.d.ts. В подходе Node разрешение модулей будет происходить через поиск модуля в файлах, размещенных в директории node_modules, что как раз нам и нужно, поскольку сторонняя библиотека `chalk` установлена именно в node_modules.

> **СОВЕТ** Подробнее прочитать о разрешении модулей можно в документации к TypeScript по адресу www.typescriptlang.org/docs/handbook/module-resolution.html.

Следующий листинг показывает, как мы добавили ts-loader в раздел rules файла webpack.config.js:

Листинг 6.14. webpack.config.js: файл конфигурации Webpack

```
const { resolve } = require('path');

module.exports = {
    entry: './src/index.ts',
    output: {
        filename: 'index.bundle.js',
        path: resolve( dirname, 'dist')
    },
    module: {
        rules: [   <-- Правила для модулей (конфигурация загрузчиков, опции считывания и пр.)
            {
                test: /\.ts$/,   <-- Применяется к файлам с расширением .ts
                exclude: /node_modules/,
                use: 'ts-loader'   <-- Компилирует TypeScript, используя опции из tsconfig.json
            }
        ]
    },
    resolve: {
        extensions: [ '.ts', '.js' ]   <-- Добавляет расширение .ts к свойству resolve, чтобы оно могло импортировать файлы TS
    },
    target: 'node',
    mode: 'production'
};
```

Правила для модулей (конфигурация загрузчиков, опции считывания и пр.)

Коротко говоря, мы обращаемся к Webpack: «Если ты видишь файл с расширением .ts, используй для его обработки ts-loader. Игнорируй файлы .ts, размещенные под директорией node_modules, — их компилировать не нужно».

В этом простом примере массив `rules` имеет всего один загрузчик — ts-loader. В реальном проекте он, как правило, включает несколько загрузчиков. Например, css-loader используется для обработки CSS-файлов; `file-loader` разрешает `import/require()` исходного файла в URL и отправляет его в генерируемую связку (используется для обработки изображений или других файлов, имеющих указанные расширения).

> **СОВЕТ** Существует альтернативный загрузчик Webpack для TS, называемый `awesome-typescript-loader` (см. https://github.com/s-panferov/awesome-typescript-loader), который может показать лучшую производительность в крупных проектах.

Теперь вы можете создать связку index.bundle.js так же, как мы делали это в предыдущем разделе, выполнив следующую команду:

```
npm run bundleup
```

Чтобы убедиться, что связанный код работает, просто запустите его:

```
node dist/index.bundle.js
```

Файл конфигурации, представленный в листинге 6.14, достаточно мал и прост; в реальных проектах файл webpack.config.js может быть сложен и включать несколько загрузчиков и плагинов. Мы в нашем малюсеньком приложении использовали только загрузчик TS, но вы наверняка будете использовать загрузчики для HTML, CSS, изображений и пр.

Ваш проект может иметь несколько `entry`-файлов, и вы можете захотеть использовать специальные плагины для создания связок особым способом. Например, если ваше приложение будет развернуто как десять связок, Webpack может извлечь общий код (из используемого вами фреймворка) в отдельную связку, чтобы остальные девять его не повторяли.

По мере роста сложности процесса связывания файл webpack.config.js также увеличивается, в связи с чем усложняется его написание и обслуживание. Передача значения неверного типа может привести к ошибкам в процессе создания связки, чьи описания иногда не так уж легко понять. Хорошие новости здесь в том, что вы можете написать файл конфигурации Webpack в TypeScript (webpack.config.ts), получая помощь от статического анализатора типов, как и при работе с любым другим TS-кодом. Прочитать об использовании TS для настройки Webpack можно в документации: https://webpack.js.org/configuration/configuration-languages.

НАЗНАЧЕНИЕ ПЛАГИНОВ WEBPACK

Загрузчики Webpack преобразуют файлы по одному за раз, но плагины имеют доступ сразу к нескольким файлам и могут обрабатывать их до или после подключения к работе загрузчиков. Список доступных Webpack-плагинов ищите здесь: https://webpack.js.org/plugins.

Например, плагин SplitChunksPlugin позволяет вам разбивать связку на раздельные части. Предположим, код вашего приложения разбит на два модуля, `main` и `admin`, и вы хотите создать две соответствующие связки. Каждый из этих модулей использует фреймворк вроде Angular. Если вы просто укажете две точки входа (`main` и `admin`), каждая связка будет включать код приложения наряду с ее собственной копией кода Angular.

Чтобы такого не случилось, вы можете обработать код с помощью SplitChunkPlugin. При использовании этого плагина Webpack не будет включать никакой Angular-код в связки `main` и `admin`; он создаст отдельную совместно используемую связку, содержащую только код Angular. Это уменьшит итоговый размер вашего приложения, так как одна копия Angular будет использоваться двумя модулями приложения. В этом случае ваш HTML-файл должен включать файл этой общей связки (например, код фреймворка Angular), сопровождаемый связкой приложения.

Плагин UglifyJSPlugin уменьшает код всех скомпилированных файлов. Он является оберткой для популярного минификатора UglifyJS, который получает JS-код и производит различные оптимизации. Например, он сжимает код, объединяя последовательные инструкции `var`, удаляет неиспользуемые переменные и недоступный код, а также оптимизирует инструкции `if`. Его минифицирующий инструмент-манглер переименовывает локальные переменные в однобуквенные.

Плагин TerserWebpackPlugin также производит уменьшение кода, используя terser (специальный JS-анализатор), тот же манглер для имен переменных, оптимизатор и средство изящного форматирования для ES6.

Используя опцию `mode: "production"` в файле webpack.config.js, вы можете неявно задействовать несколько Webpack-плагинов, которые будут оптимизировать и уменьшать связки кода. Если вам интересно, какие конкретные связки используются в режиме production, посмотрите документацию Webpack по следующему адресу: https://webpack.js.org/concepts/mode/ #mode-production.

В этом разделе мы представили проект, где TypeScript код использует JS-библиотеку Chalk. В главе 7 мы предоставим более подробное рассмотрение совместных TS-JS-проектов. Ну а пока давайте посмотрим, как мы можем использовать TypeScript с другим популярным инструментом — Babel.

6.4. ИСПОЛЬЗОВАНИЕ КОМПИЛЯТОРА BABEL

Babel — это популярный JS-компилятор, предлагающий решение для широко известной проблемы: не все браузеры поддерживают весь набор возможностей, объявленных в ECMAScript. Мы даже не говорим о полной реализации конкретной версии ECMAScript. В любой момент времени один из браузеров может реализовать конкретную выборку возможностей из ECMAScript 2019, в то время как другой по-прежнему будет понимать только ECMAScript 5. Посетите сайт caniuse.com и поищите arrow functions (стрелочные функции). Вы увидите, что Internet Explorer 11, OperaMini и некоторые другие браузеры их не поддерживают.

Если вы разрабатываете новое веб-приложение, то захотите протестировать его для всех браузеров, которые могут использоваться вашей целевой аудиторией. Babel позволяет вам писать современный JS и компилировать его в более старые версии. Несмотря на то что tsc дает возможность указать для компиляции конкретную целевую спецификацию ECMAScript (например, ES2019), Babel более точен. Он позволяет выборочно указать возможности языка, которые требуется трансформировать в JavaScript, поддерживаемый старыми браузерами.

На рис. 6.9 показан фрагмент таблицы совместимости с браузерами (из http://mng.bz/O9qw). Сверху вы видите названия браузеров и компиляторов, а слева расположен список возможностей. Браузер, компилятор или среда выполнения сервера могут полностью либо частично поддерживать некоторые из перечисленных возможностей, а плагины Babel позволяют указывать только конкретные из них, которые требуется трансформировать в более старый код. Полный список плагинов можно посмотреть в документации Babel: https://babeljs.io/docs/en/plugins.

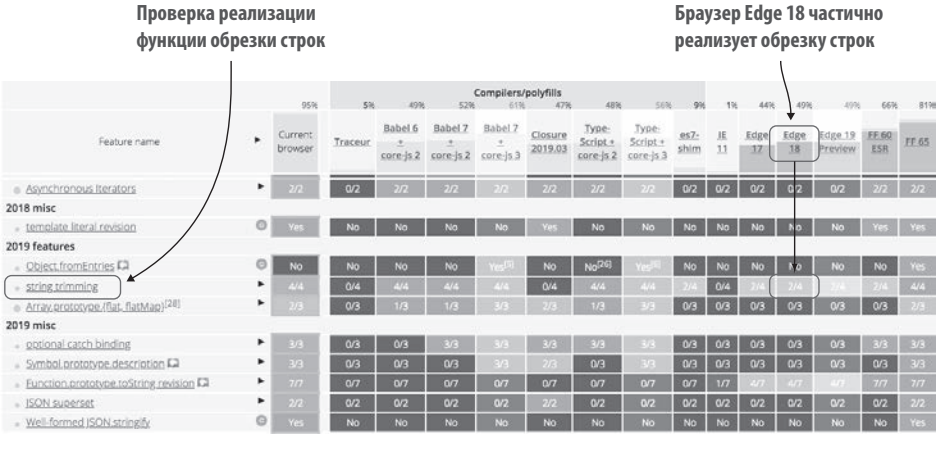


Рис. 6.9. Фрагмент таблицы совместимости с браузерами

В целях этого рассмотрения мы выбрали функцию «Обрезка строк» из ES2019 (см. черную стрелку слева от рис. 6.9). Давайте предположим, что наше приложение должно работать в браузере Edge. Проследуйте по вертикальной стрелке, и вы увидите, что Edge 18 на данный момент реализует обрезку строк только частично (2/4).

Мы можем использовать функцию обрезки строк в нашем коде, но нужно попросить Babel скомпилировать эту возможность в более старый синтаксис. Когда Edge все-таки начнет поддерживать эту функцию полноценно и компиляции не потребуется, гибкости Babel окажется достаточно.

Babel состоит из множества плагинов, каждый из которых компилирует конкретную функцию языка, но попытка найти и отобразить функции в плагины может потребовать много времени. Именно поэтому плагины Babel скомбинированы в *предустановки*, являющиеся списками именно тех из них, которые вы хотите применить для компиляции. В частности, preset-env позволяет указывать возможности ECMAScript и браузеры, которые должно поддерживать ваше приложение.

В разделе A.12 приложения мы включили скриншот из http://babeljs.io, иллюстрирующий Babel-инструмент REPL. Взгляните на меню Babel Try it now, показанное на рис. 6.10, где вы увидите навигационную панель, позволяющую конфигурировать предустановки.

Каждая предустановка просто является группой плагинов, и если вы хотите скомпилировать код в синтаксис ES2015, достаточно отметить галочкой es2015. Вместо указания имен спецификаций ECMAScript вы можете настраивать конкретные версии браузеров или других сред выполнения, используя опцию ENV PRESET. Белая стрелка на рис. 6.10 показывает редактируемое окошко с предполагаемыми значениями для предустановки ENV: >2%, ie11, safari>9. Это означает, что вы хотите, чтобы Babel скомпилировал код для запуска во всех браузерах, имеющих рыночный охват не менее 2%, а также в Internet Explorer 11 и Safari.

Ни IE11, ни Safari 9 не поддерживают стрелочные функции, и если вы введете `(a, b) ?a+b;`, Babel преобразует это в JS, который перечисленные браузеры понимают, как показано в правой части рис. 6.11.

> **СОВЕТ** Если вы видите ошибки после ввода имен браузеров, уберите галочку Enabled после введения браузеров и версий. Это похоже на баг, но к моменту прочтения вами этой книги он может быть уже устранен.

Теперь давайте изменим предустановку на «last 2 chrome versions» (две последние версии Chrome), как показано на рис. 6.12. Babel достаточно сообразителен, чтобы понять, что последние две версии Chrome поддерживают стрелочные функции и нет нужды производить преобразование.

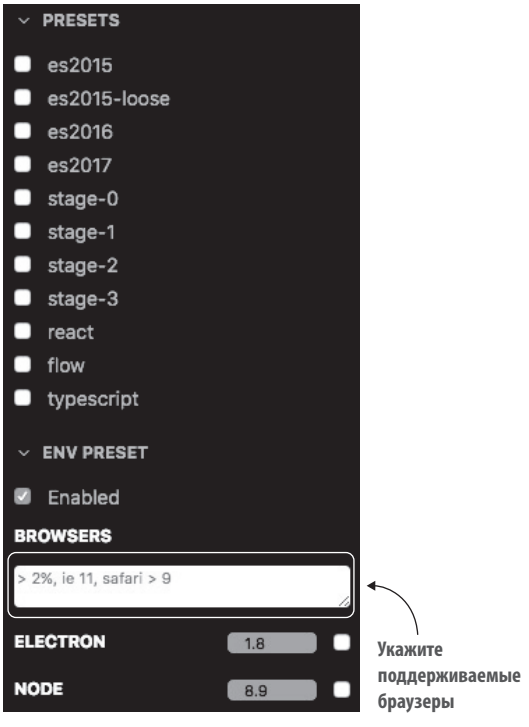


Рис. 6.10. Конфигурирование предустановки ENV

Рис. 6.11. Применение предустановок ie и safari

Предустановка ENV идет со списком браузеров, и вам нужно использовать верные имена и фразы, чтобы указать ограничения (например, last2majorversions, Firefox>=20 или >5% in US). Эти фразы перечислены в проекте browserslist, который доступен здесь: https://github.com/browserslist/browserslist.

ПРИМЕЧАНИЕ Мы использовали предустановку ENV в Babel REPL, чтобы поиграть с целевыми средами, но эти настройки могут быть также сконфигурированы и использованы из командной строки. В листинге 6.15 мы добавим в файл конфигурации .babelrc следующее: @babel/preset-env. В листинге 6.17 вы увидите файл .browserlistrc, в котором вы можете настроить конкретные браузеры и версии, как мы делали в Babel REPL. Подробнее о preset-env вы можете прочитать в документации Babel здесь: https://babeljs.io/docs/en/next/babel-preset-env.html.

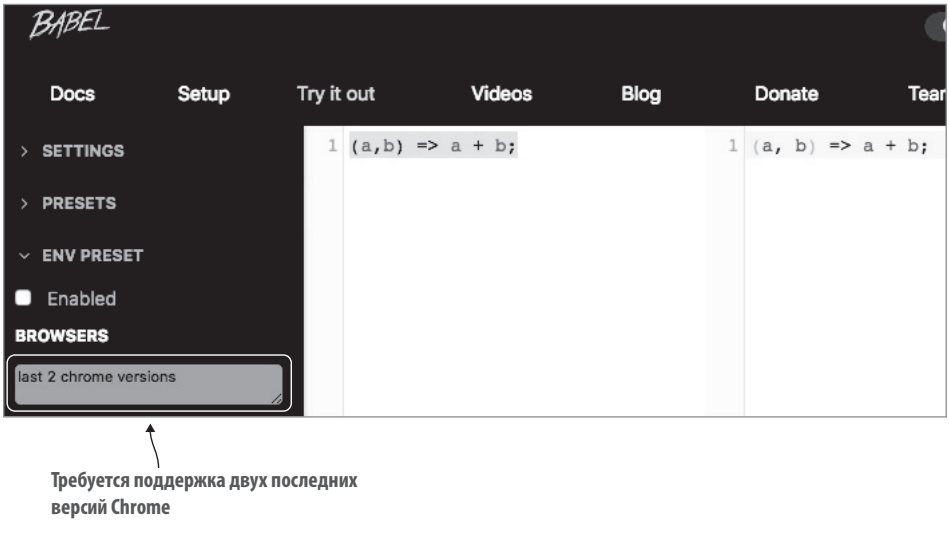


Рис. 6.12. Применение предустановки Chrome

Babel может использоваться для компиляции таких языков, как JavaScript, TypeScript, CoffeeScript, Flow и др. Например, фреймворк React использует синтаксис JSX, который не относится к стандарту JS, и Babel это понимает. В главе 12 мы используем Babel с приложением React.

Когда Babel компилирует TS, он не выполняет проверку типов в отличие от tsc. Создатели Babel не реализовали полноценный компилятор TS. Babel просто считывает TS-код и генерирует соответствующий синтаксис JS.

Должно быть, вы подумали: «Я вполне доволен компилятором TS. Зачем вообще в эту книгу о TypeScript включать раздел о компиляторе JS-в-JS?» Причина в том,

что вы можете подключиться к проекту, где часть модулей написаны в JS, а часть в TS. В таких проектах Babel может уже быть частью потока разработки-развертывания. Например, Babel популярен среди разработчиков, использующих фреймворк React, который лишь недавно начал поддерживать TS.

Подобно любому npm-пакету, вы можете установить Babel локально или глобально (с опцией `-g`). Локальная установка внутри директории проекта делает этот проект самодостаточным, так как после запуска `npm install` вы можете использовать Babel, не ожидая, что он установлен где-то в другом месте (кто-нибудь может работать над вашим проектом с другого компьютера).

```
npm install @babel/core @babel/cli @babel/preset-env
```

Здесь @babel/core является компилятором Babel, @babel/cli — интерфейсом командной строки, а @babel/preset-env — это предустановка ENV, которую мы недавно рассматривали.

> **ПРИМЕЧАНИЕ** В реестре npmjs.org пакеты JavaScript могут быть организованы как *ветки*. Например, `@babel` — это ветка для пакетов, относящихся к Babel. `@angular` — это ветка для пакетов, принадлежащих фреймворку Angular. `@types` — это место для файлов определений типов TS для различных популярных JS-библиотек.

В последующих разделах мы представим вам три небольших проекта. Первый использует Babel с JS, второй — Babel с TS, а третий — Babel, TS и Webpack.

6.4.1. Использование Babel с JavaScript

В этом разделе мы рассмотрим простой проект, использующий Babel с JavaScript и расположенный в директории babel-javascript. Мы продолжим работать с трехстрочным скриптом index.js, представленным в листинге 6.7 и использующем JS-библиотеку Chalk. Единственное изменение в следующем листинге заключается в том, что сообщение теперь гласит «Compiled with Babel» (Скомпилировано с помощью Babel).

Листинг 6.15. index.js: исходный код приложения babel-javascript

```
const chalk = require('chalk');
const message = 'Compiled with Babel';
console.log(chalk.black.bgGreenBright(message));
```

Следующий листинг показывает npm-скрипт, который мы будем использовать для запуска Babel и зависимостей, которые должны быть установлены на машине разработчика.

Листинг 6.16. Фрагмент из babel-javascript/package.json

```
"scripts": {
    "babel": "babel src -d dist"  <-- npm-скрипт для компиляции кода из src в dist
},
"dependencies": {
    "chalk": "^2.4.1"
},
"devDependencies": {  <-- Локально установленные dev-зависимости
    "@babel/cli": "^7.2.3",
    "@babel/core": "^7.2.2",
    "@babel/preset-env": "^7.2.3"
}
```

Babel настраивается в файле .babelrc, и наш файл конфигурации будет очень прост. Нам нужно только использовать `preset-env` для компиляции.

Листинг 6.17. Файл .babelrc

```
{
    "presets": [
        "@babel/preset-env"
    ]
}
```

Мы не настраивали здесь никакие конкретные версии браузеров, и без каких бы то ни было опций конфигурации @babel/preset-env ведет себя в точности так же, как @babel/preset-es2015, @babel/preset-es2016 и @babel/preset-es2017. Другими словами, все возможности языка, представленные в ECMAScript2015, 2016 и 2017, будут скомпилированы в ES5.

> **СОВЕТ** Мы сконфигурировали Babel в файле .babelrc, который отлично подходит для статических конфигураций вроде нашей. Если ваш проект требует программного создания конфигураций Babel, вам понадобится использовать файл babel.config.js (подробнее в документации Babel здесь: https://babeljs.io/docs/en/config-files#project-wide-configuration). Если вы хотите увидеть, как Babel компилирует наш файл src/index.js, установите зависимости этого проекта, выполнив `npm install`, а затем запустите npm-скрипт из `package.json: npm run babel`.

Следующий листинг показывает скомпилированную версию index.js, созданную в директории dist. Она будет иметь следующее содержимое (сравните с листингом 6.15):

Листинг 6.18. dist/index.js: скомпилированная версия src/index.js

```
"use strict";  <-- Babel добавил эту строку
var chalk = require('chalk');
var message = 'Compiled with Babel';   | Babel заменил const на var
console.log(chalk.black.bgGreenBright(message));
```

ПРИМЕЧАНИЕ Скомпилированный файл по-прежнему вызывает `require('chalk')`, и эта библиотека расположена в отдельном файле. Помните, что Babel — это не бандлер. Мы используем Webpack с Babel в разделе 6.4.3.

Можете запустить скомпилированную версию так:

```
node dist/index.js
```

Вывод консоли будет выглядеть схожим с изображенным на рис. 6.13.

Если бы мы захотели, чтобы Babel генерировал код, работающий в конкретных версиях браузеров, потребовалось бы добавить дополнительный файл конфигурации, .browserslistrc. Например, нам нужно, чтобы код работал только в двух последних версиях Chrome и Firefox. Мы можем создать следующий файл в корне проекта:

Рис. 6.13. Запуск программы, скомпилированной с помощью Babel

Листинг 6.19. Пример файла .browserslistrc

```
last 2 chrome versions
last 2 firefox versions
```

Теперь при запуске Babel не будет преобразовывать `const` в `var`, как в листинге 6.18, потому что и Firefox, и Chrome уже поддерживают ключевое слово `const`. Попробуйте сами, чтобы убедиться.

6.4.2. Использование Babel с TypeScript

В этом разделе мы рассмотрим простой проект, использующий Babel с TypeScript; он размещен в директории babel-typescript. Мы продолжим работать с трехстрочным скриптом, представленным в листинге 6.11 и использующим JS-библиотеку Chalk. Единственное изменение будет в том, что теперь сообщение гласит «Compiled with Babel» (Скомпилировано с помощью Babel).

Листинг 6.20. index.ts: исходный код приложения babel-typescript

```
import chalk from 'chalk';
const message: string = 'Compiled with Babel';
console.log(chalk.black.bgGreenBright(message));
```

В сравнении с package.json из чистого JS-проекта (см. листинг 6.16) наш TS-проект добавляет dev-зависимость `preset-typescript`, отделяющую типы TS от кода,

чтобы Babel мог воспринимать его как чистый JS. Мы также добавим опцию `--extensions '.ts'` в npm-скрипт, запускающий Babel, как в листинге 6.21. Теперь Babel будет считывать файлы .ts.

Листинг 6.21. Фрагмент из package.json

```
"scripts": {
    "babel": "babel src -d dist --extensions '.ts'"
},
"dependencies": {
    "chalk": "^2.4.1"
},
"devDependencies": {
    "@babel/cli": "^7.2.3",
    "@babel/core": "^7.2.2",
    "@babel/preset-env": "^7.2.3",
    "@babel/preset-typescript": "^7.1.0"
}
```

Инструктирует Babel обрабатывать файлы с расширением .ts

Добавляет зависимость preset-typescript

Как правило, предустановки включают набор плагинов, но preset-typescript содержит только один, @babel/plugin-transform-typescript. Этот плагин внутренне использует @babel/plugin-syntax-typescript, чтобы считывать TypeScript, и @babel/helper-plugin-utils для основных утилит плагинов.

Несмотря на то что @babel/plugin-transform-typescript преобразует TS-код в синтаксис ES.Next, это не компилятор TS. Как бы странно это ни звучало, Babel просто стирает TypeScript. Например, он преобразует `const x: number = 0` в `const x = 0`. @babel/plugin-transform-typescript намного быстрее, чем компилятор TS, так как не производит проверку типов для вводных файлов.

> **ПРИМЕЧАНИЕ** @babel/plugin-transform-typescript имеет несколько небольших ограничений, перечисленных в документации на https://babeljs.io/docs/en/babel-plugin-transform-typescript (например, он не поддерживает `const enum`). Для лучшей поддержки TS рассмотрите использование плагинов @babel/plugin-proposal-class-properties и @babel/plugin-proposal-object-rest-spread.

Прочитав первые пять глав этой книги, вы наверняка уже начали ценить проверку типов и обнаружение ошибок во время компиляции, осуществляемое реальным компилятором TS. Неужели теперь мы действительно предлагаем вам использовать Babel, чтобы стереть связанный с TS синтаксис? Не совсем так. В процессе разработки вы можете продолжать использовать tsc (с помощью tsconfig.json) и IDE с полной поддержкой TypeScript. Тем не менее на стадии развертывания вы можете все же ввести Babel- и ENV-предустановки. (Скорее всего, вы уже оценили гибкость, предлагаемую ENV-предустановками, при конфигурировании целевых браузеров, не так ли?)

В вашем процессе сборки вы можете даже добавить npm-скрипт (в package.json), запускающий tsc:

```
"check_types": "tsc --noEmit src/index.ts"
```

Теперь вы можете последовательно запустить `check_types` и Babel при наличии локально установленного tsc:

```
npm run check_types && npm run babel
```

Опция `--noEmit` гарантирует, что tsc не сгенерирует никаких файлов (вроде index.js), так как это будет сделано командой `babel`, выполняемой сразу после `check_types`. Если в index.js присутствуют ошибки компиляции, процесс сборки провалится и команда `babel` даже не запустится.

> **СОВЕТ** Если вы используете `&&` (двойной амперсанд) между двумя npm-скриптами, они будут выполняться последовательно. Для параллельного выполнения используйте `&` (одинарный амперсанд). Подробности вы можете найти во врезке «Использование амперсандов в npm-скриптах в Windows» в главе 10.

В этом проекте файл конфигурации .babelrc включает @babel/preset-typescript.

Листинг 6.22. Файл .babelrc

```
{
    "presets": [
        "@babel/preset-env",
        "@babel/preset-typescript"
    ]
}
```

В сравнении с проектом babel-javascript мы сделали следующие относящиеся к TypeScript изменения:

- Добавили опцию `--extensions '.ts'` в команду, запускающую Babel.
- Добавили в package.json связанные с TypeScript dev-зависимости.
- Добавили @babel/preset-typescript в файл конфигурации .babelrc.

Чтобы скомпилировать наш простой скрипт index.ts, запустите следующий npm-скрипт из package.json:

```
npm run babel
```

Вы найдете скомпилированную версию index.js в директории dist. Вы можете запустить скомпилированный код так же, как мы это делали в предыдущем разделе:

```
node dist/index.js
```

Теперь давайте добавим в наш рабочий поток Webpack, чтобы связать скрипт index.js и JS библиотеку Chalk.

6.4.3. Использование Babel с TypeScript и Webpack

Babel — это компилятор, но не бандлер, который необходим для любого реального приложения. Вы вольны выбирать из ряда доступных бандлеров (вроде Webpack, Rollup и Browserify), но мы будем придерживаться Webpack. В этом разделе мы рассмотрим простой проект, использующий Babel с TypeScript и Webpack. Расположен он в директории webpack-babel-typescript.

В разделе 6.3.2 мы рассмотрели настройку TypeScript-Webpack, и далее мы продолжим использовать наш трехстрочный исходный код из того проекта.

Листинг 6.23. index.ts: исходный код приложения webpack-babel-typescript

```
import chalk from 'chalk';
const message: string = 'Built with Babel bundled with Webpack';
console.log(chalk.black.bgGreenBright(message));
```

В следующем листинге показан раздел devDependency из package.json.

Листинг 6.24. Раздел devDependencies в package.json

```
"devDependencies": {
    "@babel/core": "^7.2.2",
    "@babel/preset-env": "^7.2.3",
    "@babel/preset-typescript": "^7.1.0",
    "babel-loader": "^8.0.5",  <-- Добавление Webpack-загрузчика для Babel
    "webpack": "^4.28.3",
    "webpack-cli": "^3.1.2"
}
```

Сравните зависимости Babel в листингах 6.24 и 6.21. В листинге 6.24 присутствуют три изменения:

- Мы добавили babel-loader, являющийся Webpack-загрузчиком для Babel.
- Мы удалили babel-cli, потому что не будем запускать Babel из командной строки.
- Вместо babel-cli Webpack будет использовать babel-loader как часть процесса связывания.

Как вы помните из раздела 6.3, Webpack использует файл конфигурации webpack.config.js. Для настройки TS с помощью Webpack мы использовали ts-loader (см. листинг 6.14). В данном же случае мы хотим, чтобы файлы с расширением .ts

обрабатывал babel-loader. Следующий листинг показывает раздел Babel из файла webpack.config.js.

Листинг 6.25. Фрагмент из webpack-babel-typescript/webpack.config.js

```
module: {
    rules: [
        {
            test: /\.ts$/,   <-- Применяет это правило для файлов, оканчивающихся на .ts
            exclude: /node_modules/,
            use: 'babel-loader'   <-- Обрабатывает файлы .ts с помощью babel-loader
        }
    ]
},
```

Файл .babelrc будет выглядеть в точности так же, как и в предыдущем разделе (см. листинг 6.20).

После установки зависимостей командой `npm install` мы готовы к созданию связки через выполнение команды bundleup из package.json:

```
npm run bundleup
```

Эта команда создаст index.bundle.js в директории dist. Этот файл будет содержать скомпилированную (при помощи Babel) версию файла index.js, а также код из JS-библиотеки Chalk. Запустить эту связку вы можете как обычно:

```
node dist/index.bundle.js
```

Вывод, показанный на рис. 6.14, будет знакомым.

Для генерации JavaScript не нужно выбирать между Babel и tsc. Они могут успешно сосуществовать в одном проекте.

```
$ node dist/index.bundle.js
Built with Babel bundled with Webpack
```

Рис. 6.14. Запуск программы, скомпилированной Babel

ПРИМЕЧАНИЕ Противники TypeScript зачастую используют такой аргумент: «Если я буду писать на чистом JS, мне не потребуется использовать компилятор. Я смогу запустить JS-программу сразу после ее написания». Это абсолютно ошибочно, так как если вы не хотите игнорировать новейший синтаксис JS, представленный, начиная с версии 2015, потребуется процесс, который сможет компилировать код, написанный в современном JS, в код, который смогут понять все браузеры. Скорее всего, вы так или иначе будете использовать в своем проекте компилятор, будь то Babel, TypeScript или какой-либо другой.

6.5. ИНСТРУМЕНТЫ ДЛЯ РАССМОТРЕНИЯ

В этом разделе мы бы хотели отметить пару инструментов, которые еще не были выпущены официально на момент написания книги, но могут стать полезным дополнением в арсенале TypeScript разработчика.

6.5.1. Знакомство с Deno

Каждый JavaScript разработчик знает о среде выполнения Node.js. Мы также используем ее в этой книге для выполнения приложений вне браузеров. Все любят Node.js..., за исключением его создателя Райана Даля (Ryan Dahl). В 2018-м Райан провел презентацию под названием «10 нюансов Node.js, о которых я сожалею» (на YouTube она доступна по ссылке http://mng.bz/Yeyz), после чего он начал работать над Deno — безопасной средой выполнения, созданной на основе движка V8 (как и Node) и имеющей встроенный компилятор TS.

> **ПРИМЕЧАНИЕ** Во время написания этой книги Deno по-прежнему является экспериментальным элементом ПО. Вы можете проверить ее актуальный статус на https://deno.land.

Некоторые из сожалений Райана касались того, что приложения Node нуждаются в package.json, node-modules.npm для разрешения модулей и центральном репозитории для распространения пакетов. Deno ничего из этого не требуется. Если вашему приложению нужен пакет, оно должно иметь возможность получить его непосредственно из репозитория исходного кода этого пакета. Мы покажем, как это работает, в небольшом проекте под названием deno, который идет с кодом из этой главы.

Deno может выполнять как JavaScript-, так и TypeScript-код, а наш проект имеет всего один сценарий index.ts, как показано в следующем листинге:

Листинг 6.26. index.ts: исходный код приложения для запуска под Deno

```
import { bgGreen, black } from 'https://deno.land/std/colors/mod.ts';
const message: string = 'Ran with deno!';
console.log(black(bgGreen(message)));
```

Включает библиотеку colors

Использует строковый тип TS

Использует API из библиотеки colors

Обратите внимание, что мы импортируем библиотеку colors прямо из источника. Нет никакого package.json, который бы мог перечислять эту библиотеку в качестве зависимости, и не нужно использовать `npm install` для получения пакета из центрального репозитория. Она использует только модули ES6, и вам для ее импорта в приложение нужно только знать URL.

Вы можете поинтересоваться: «Разве не опасно использовать прямую ссылку на стороннюю библиотеку? Что, если ее код изменится, нарушив работу приложения?» Этого не произойдет, так как при первой загрузке библиотеки Deno кэширует ее локально. Каждый последующий запуск приложения будет переиспользовать одну и ту же версию каждой библиотеки до тех пор, пока вы не укажете опцию `--reload`.

> **ПРИМЕЧАНИЕ** Мы не смогли использовать библиотеку Chalk для этого примера, так как она изначально не упакована для использования Deno.

Все, что нужно для запуска этого сценария, — это исполняемый файл Deno, который можно загрузить с https://github.com/denoland/deno/releases. Просто выберите последний релиз и получите zip-файл для вашей платформы. Например, для MacOS загрузите и распакуйте файл deno_osx_x64.gz.

Для простоты загрузите приложение в директорию deno. Как только вы загрузите Deno, вы можете использовать следующую команду для запуска приложения:

```
./deno_osx_x64 index.ts
```

> **ПРИМЕЧАНИЕ** В macOS вам может понадобиться разрешение на выполнение этого файла: `chmod+x ./deno_osx_x64`.

> **СОВЕТ** Если вы запустите его в Windows, убедитесь, что у вас установлена минимум шестая версия PowerShell и Windows Management Framework. В противном случае вы можете столкнуться с такой ошибкой: «TS5009: Cannot find the common subdirectory path for input files» (Невозможно найти общий путь поддиректории для файлов ввода).

Deno запускает программы TypeScript прямо «из коробки», не требуя при этом npm или package.json. При первом запуске этого приложения вы увидите следующий вывод:

```
Compiling file: ...chapter6/deno/index.ts
Downloading https://deno.land/std/colors/mod.ts...
Compiling https://deno.land/std/colors/mod.ts
```

Библиотека Deno скомпилировала index.ts, а затем загрузила и скомпилировала библиотеку colors. После этого она запустила наше приложение, произведя вывод, показанный на рис. 6.15.

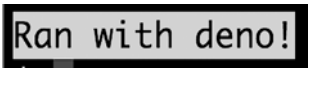


Рис. 6.15. Запуск приложения под Deno

Deno кэшировала скомпилированную библиотеку colors, поэтому при следующем запуске приложения уже не потребуется ее загружать и компилировать. Как вы видите, конфигурировать зависимости проекта не потребовалось и перед запуском приложения ничего не пришлось устанавливать или настраивать.

Deno не понимает формат пакетов npm, но если эта среда наберет популярность, то разработчикам, поддерживающим популярные JS-библиотеки, придется пакетировать свои продукты в формат, приемлемый Deno. Советуем следить за развитием этого инструмента.

6.5.2. Знакомство с ncc

Второй инструмент для рассмотрения — это ncc (https://github.com/zeit/ncc). Он является интерфейсом командной строки для компиляции модуля Node.js в один файл вместе со всеми его зависимостями. Этот инструмент может использоваться TS-разработчиками, которые пишут приложения, выполняемые на стороне сервера.

В качестве бэкграунда ncc можно отметить, что это продукт компании Zeit, являющейся бессерверным облачным провайдером. Вы могли слышать об их продукте Now (https://zeit.co/now), который предлагает суперлегкое бессерверное развертывание веб-приложений.

Zeit также разрабатывает ПО, позволяющее разделять любое приложение на максимально возможное количество мелких частей. Например, если вы пишете приложение, использующее фреймворк Express, то придется представить каждую конечную точку отдельной связкой, содержащей только код, необходимый для функционирования этой конечной точки.

Это позволяет конечным точкам избегать запуска активных серверов. Если клиент достигает конечной точки, она возвращает бессерверную связку в миниатюрном контейнере, и время отклика составляет всего 100 мс, что весьма впечатляет. Инструмент ncc также способен пакетировать серверные приложения в небольшие связки.

В качестве ввода ncc может получать любой код TS или JS и на выходе производить связку. Настроек для его работы требуется минимум, а иногда и вовсе не требуется. Единственное требование — использование вашим кодом ES6-модулей или `require()`.

Мы рассмотрим небольшое приложение — ncc-typescript, которое имеет несколько настроек, поскольку мы используем TS. Если бы мы писали его в JS, настройки бы вообще не потребовались. Это приложение размещено в директории ncc-typescript,

содержащей файлы package.json, указанный в следующем листинге, tsconfig.json и index.ts, использующий библиотеку Chalk.

Листинг 6.27. Фрагмент из ncc/package.json

```
{
    "name": "ncc-typescript",
    "description": "A code sample for the TypeScript Quickly book",
    "homepage": "https://www.manning.com/books/typescript-quickly",
    "license": "MIT",
    "scripts": {
        "start": "ncc run src/index.ts",
        "build": "ncc build src/index.ts -o dist -m"
    },
    "dependencies": {
        "chalk": "^2.4.1"
    },
    "devDependencies": {
        "@zeit/ncc": "^0.16.1"
    }
}
```

Использует ncc в режиме выполнения

Компилирует TypeScript с помощью ncc (-m указывается для продакшен-оптимизации)

Инструмент ncc

Вы не увидите tsc как зависимость в файле package.json, так как компилятор TS является внутренней зависимостью ncc. Тем не менее при необходимости вы можете перечислить опции компилятора в tsconfig.json. С точки зрения разработки ncc позволяет вам компилировать и запускать TS-код в одном процессе.

Обратите внимание на раздел scripts, в котором мы определили две команды: `start` и `build`. Это позволяет TS-разработчикам использовать два режима ncc:

- *Режим выполнения* — ncc запускает TS-код без явной компиляции (он компилирует его внутренне).
- *Режим сборки* — TypeScript компилируется в JavaScript.

Хорошо здесь то, что вам не нужно использовать бандлер вроде Webpack, так как ncc создаст связку за вас. Попробуйте сами, выполнив `npm install` и запустив образец приложения, размещенный в index.ts:

```
npm run start
```

Как определено в package.json, показанном в листинге 6.27, команда `start` запустит ncc для компиляции и выполнения index.ts, а вывод консоли показан на рисунке 6.16.

Наша команда `start` скомпилировала index.ts с генерацией карт кода (по умолчанию). В режиме выполнения скомпилированный файл не сгенерировался. Если вы запустите команду `build`, ncc сгенерирует связку index.js в директории dist, но приложение не запустится:

```
npm run build
```

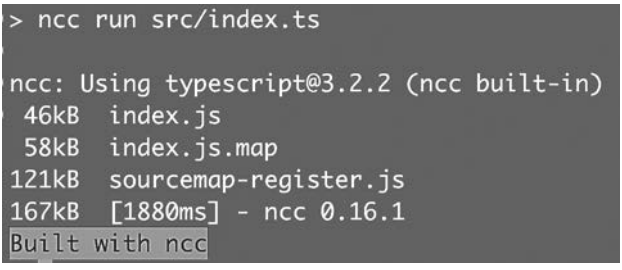


Рис. 6.16. Запуск приложения с помощью ncc

Вывод консоли после команды `build` будет похож на следующий:

```
ncc: Using typescript@3.2.2 (ncc built-in)
24kB dist/index.js
24kB [1313ms] - ncc 0.16.1
```

Размер оптимизированной связки составляет 24 Кб (ncc использует Webpack внутренне). Сгенерированная ncc-связка содержит код, написанный нами, а также код библиотеки Chalk, и вы можете запустить приложение как обычно:

```
node dist/index.js
```

Вывод, показанный на рис. 6.17, выглядит, как и ожидалось.

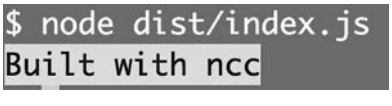


Рис. 6.17. Запуск приложения с помощью ncc

Подытоживая, можно назвать несколько выгод использования ncc:

- Для сборки и запуска приложений не требуется настройка.
- Вы можете использовать режим выполнения или сборки. Режим выполнения избавляет вас от явного компилирования TS-кода.
- ncc поддерживает гибридные проекты, в которых часть кода написана в JavaScript, а часть в TypeScript.

В этом разделе мы рассмотрели два интересных инструмента — Deno и ncc, но экосистема TypeScript развивается очень быстро, и вам следует обращать внимание на другие появляющиеся инструменты, которые помогут вам повысить продуктивность, сделают приложения более отзывчивыми, а процесс сборки и развертывания более простым.

ПРИМЕЧАНИЕ Мы не упомянули еще один полезный пакет под названием ts-node, который может запускать tsc и Node.js среды как один процесс. Мы используем его в разделе 10.4.2 главы 10 во время запуска сервера, написанного на TypeScript.

ИТОГИ

- Карты кода позволяют вам отлаживать TS-код, даже несмотря на то что браузер выполняет JS-версию.
- Линтеры используются для проверки и обеспечения соблюдения стилей написания кода. Мы также представили TSLint, но он вскоре сольется с ESLint.
- Для развертывания приложения мы обычно связываем исходные файлы, чтобы снизить общее число файлов для загрузки. Webpack является одним из наиболее популярных бандлеров.
- JavaScript-разработчики используют Babel для компиляции кода, использующего новейший синтаксис ECMAScript, в код, поддерживаемый конкретными версиями браузеров. В некоторых случаях имеет смысл использовать компилятор TS совместно с компилятором Babel.
- Знание синтаксиса любого языка программирования очень важно, но не менее важно понимание процесса, благодаря которому ваша программа может быть преобразована в рабочее запускаемое приложение.

7 *Использование TypeScript и JavaScript в одном проекте*

В этой главе:

- ✓ Преимущества TypeScript при работе с JavaScript-библиотекой.
- ✓ Роль файлов определений типов.
- ✓ Апгрейд существующего JavaScript-приложения в TypeScript.

В этой главе мы покажем, как воспользоваться преимуществами TS вроде обнаружения ошибок при компиляции и автоподстановке даже при использовании сторонних библиотек, написанных в JS. Начнем с объяснения роли файлов определений типов, а затем обсудим конкретный случай их применения, в котором приложение, написанное в TS, использует JS-библиотеку. В завершение мы обсудим нюансы, которые вам стоит рассмотреть, прежде чем реализовывать постепенный апгрейд приложения JavaScript в TypeScript.

7.1. ФАЙЛЫ ОПРЕДЕЛЕНИЙ ТИПОВ

JavaScript был создан в 1995 году. С тех пор были написаны мириады строк кода. Разработчики по всему миру выпустили тысячи библиотек, написанных в JS, и вы наверняка сможете использовать преимущество каких-либо из них в своем TS-приложении.

Было бы наивным ожидать, что создатели JS-библиотек станут тратить время на переписывание своих библиотек или фреймворков на TypeScript, но мы хотим

использовать наследие JavaScript в наших TS-приложениях. Более того, мы уже избалованы статическим анализатором типов, автоподстановкой и мгновенными отчетами об ошибках компиляции. Можем ли мы продолжить наслаждаться этими возможностями при работе с API JS-библиотек? Да, такая возможность доступна нам благодаря *файлам определений типов*.

ПРИМЕЧАНИЕ В разделе 6.3.2 вы уже видели проект, в котором TS-код использовал JS-библиотеку Chalk. Целью того примера было показать, как *связывать* TypeScript и JavaScript вместе, поэтому мы не обсуждали, как именно код взаимодействует и способен ли анализатор кода TS помочь нам корректно использовать библиотеку Chalk. Мы не использовали файлы определений типов в главе 6, но прибегнем к ним в этой.

7.1.1. Знакомство

Задача файлов определений типов позволить компилятору TS узнать, какие типы ожидают API конкретных JS-библиотек или сред выполнения. Эти файлы просто включают имена переменных (с типами) и сигнатуры функций (с типами), используемые конкретной JS-библиотекой.

В 2012 году Борис Янков (Boris Yankov) создал репозиторий на GitHub для файлов определений типов (см. https://github.com/DefinitelyTyped/DefinitelyTyped). Другие разработчики при этом стали принимать участие в его развитии, и на данный момент этот проект имеет более 10 000 участников. Затем был создан сайт DefinitelyTyped.org, а после выпуска TypeScript 2.0 на npmjs.org была создана ветка @types, ставшая еще одним репозиторием для файлов определений типов. На данный момент все файлы объявлений с DefinitelyTypes.org также автоматически публикуются в ветке @types.

Имена этих файлов имеют суффикс d.ts, и вы можете найти их для более чем 7000 JS-библиотек по ссылке www.npmjs.org/~types. Просто проследуйте по ней и выполните поиск нужной вам библиотеки. Например, вы можете найти информацию по определениям типов jQuery здесь: www.npmjs.com/package/@types/jquery — на рис. 7.1 приведен скриншот этой веб-страницы.

В верхней правой части рис. 7.1 вы можете увидеть команду, устанавливающую файлы определений типов для jQuery, но нам нравится использовать опцию `-D`, чтобы npm добавлял @types/jquery в раздел devDependencies файла package.json проекта:

```
npm install @types/jquery -D
```

ПРИМЕЧАНИЕ Предыдущая команда не устанавливает библиотеку jQuery; она устанавливает только определения типов для членов jQuery.

Как правило, вы устанавливаете определения типов с npmjs.org, указывая имя ветки @types, сопровождаемое именем пакета. Сразу после установки @types/jquery вы сможете найти несколько файлов с расширениями d.ts, например, jQuery.d.ts

и jQueryStatic.d.ts. Расположены они будут в директории node_modules/@types/jquery вашего проекта. Компилятор TypeScript (и статический анализатор) будут использовать их, чтобы делать автоподстановку и обнаружение ошибок типов.

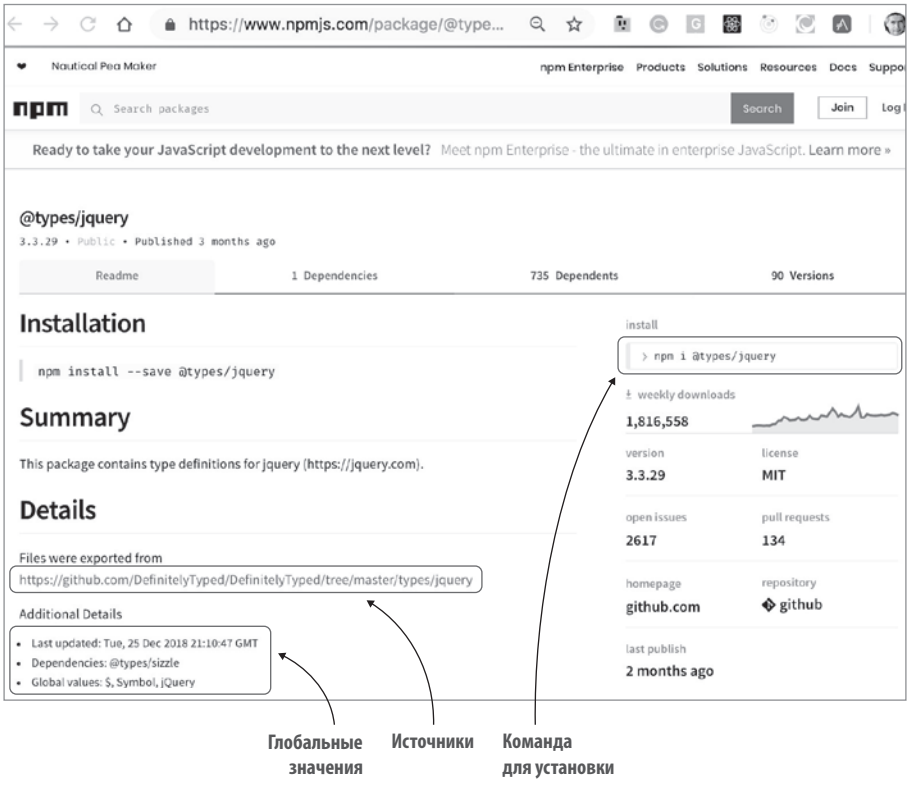


Рис. 7.1. Определения типов jQuery на npmjs.org

В центре рис. 7.1 вы видите URL источников определений типов для jQuery, а в нижней части располагаются имена глобальных значений, предложенных jQuery. Например, вы можете использовать $ для обращения к API jQuery, когда она будет установлена.

Вы можете создать новую директорию и преобразовать ее в npm-проект, выполнив в ней команду `npm init -y`. Эта команда создает файл package.json, после чего вы можете установить определения типов для jQuery:

```
npm install @types/jquery -D
```

Давайте посмотрим, начнут ли статический анализатор и ваша IDE помогать вам с API jQuery после установки файлов определений типов.

7.1.2. Файлы определений типов и IDE

Чтобы увидеть, как IDE использует эти файлы, откройте npm-проект, созданный в предыдущем разделе. В вашей IDE создайте и откройте main.ts, введите $. и нажмите Ctrl-пробел. Если вы используете IDE WebStorm, то увидите доступные API jQuery, как показано на рис. 7.2. VS Code также покажет доступные API jQuery.

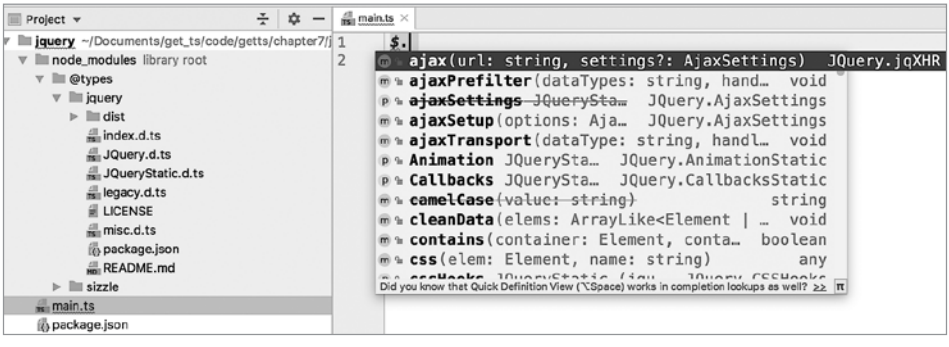


Рис. 7.2. Автоподстановка WebStorm для jQuery

В верхней части рис. 7.2 вы можете видеть jQuery-метод `ajax()` со строго типизированным аргументом, прямо как в любой TS-программе. Имейте в виду, что мы даже не устанавливали jQuery (которая все равно написана в JavaScript); у нас есть только определения типов.

Это здорово, но давайте откроем этот же проект в VS Code. Вы можете не увидеть никакой автоподстановки, как на рис. 7.3. Причина в том, что IDE WebStorm автоматически показывает все определения, какие она может найти в этом проекте, а VS Code предпочитает, чтобы мы явно указывали, какой d.ts-файл использовать.

Давайте сыграем по правилам VS Code и создадим файл tsconfig.json с опцией `types` компилятора. В этом массиве можно указать, какие определения типов использовать для автоподстановки (используя имена директорий под node_modules/@types). Следующий листинг показывает файл tsconfig.json, добавленный нами в проект.

Листинг 7.1. Файл конфигурации TypeScript, tsconfig.json

```
{
    "compilerOptions": {
        "types" : ["jquery"]
    }
}
```

Несмотря на то что `types: [jquery]` работает для этого примера, если понадобится добавить файлы определений типов для нескольких JS-библиотек, то потребуется перечислить их все в опции `types` компилятора (например, `types:`

`[jquery, lodash]`). Но добавление опции `types` это не единственный способ помочь компилятору в поиске определений типов — в разделе 7.14 мы покажем вам директиву `references`.

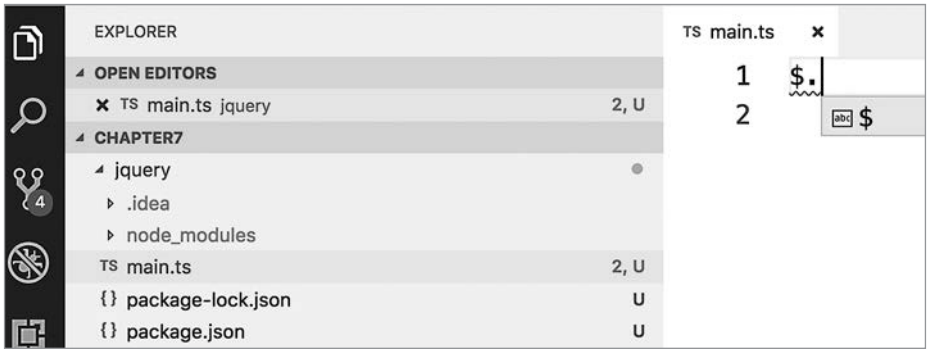


Рис. 7.3. VS Code не может найти определения типов для jQuery

ПРИМЕЧАНИЕ Проекту может потребоваться обратиться к внешним модулям, вроде jquery, lodash и др. Процесс, при котором компилятор выясняет, к чему относится импорт, называется разрешением модуля. Подробнее об этом процессе можно узнать в документации TypeScript: www.typescriptlang.org/docs/handbook/module-resolution.html.

Теперь введите `$.` и нажмите Ctrl-пробел. При этом автоподстановка должна начать исправно работать. Щелкните на функции `ajax()`, и VS Code покажет ее программную документацию, как видно на рис. 7.4. Неважно, какую IDE вы используете, наличие d.ts-файла для JS-кода окажет неоценимую помощь со стороны компилятора и статического анализатора TS.

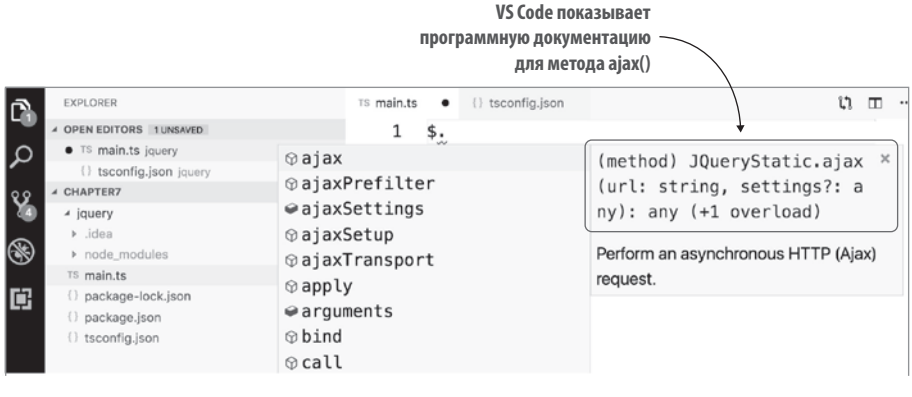


Рис. 7.4. VS Code показывает автоподстановку и документацию для JQuery метода ajax()

ПРИМЕЧАНИЕ В WebStorm, чтобы увидеть программную документацию для элемента, выбранного в списке автоподстановки, выделите его и нажмите Ctrl+J.

Давайте заглянем внутрь файла определений типов. Стрелка на рис. 7.4 указывает на имя интерфейса, где TS нашел определения типов для функции `ajax()`. Так получилось, что она определена в файле JQueryStatic.d.ts, расположенном в директории node_modules/@types/jquery. Наведите указатель мыши на имя функции `ajax()` и щелкните, удерживая Ctrl. И VS Code, и WebStorm откроют следующий фрагмент определений типов `ajax()`:

Листинг 7.2. Фрагмент из JQueryStatic.d.ts

```
/**
 * Выполнить асинхронный HTTP (Ajax) запрос.
 * @param url Строка, содержащая URL, на который отправляется запрос.
 * @param settings Набор пар ключ – значение, конфигурирующих запрос Ajax.
➥ Все установки опциональны.
 * Значение по умолчанию может быть установлено для любой опции
   с $.ajaxSetup(). См. jQuery.ajax(
➥ settings ) ниже
 * для получения полного списка настроек.
 * @see \`{@link https://api.jquery.com/jQuery.ajax/ }\`
 * @since 1.5
 */
ajax(url: string, settings?: JQuery.AjaxSettings): JQuery.jqXHR;
```

Описание параметров ajax()

Описание функции ajax()

Сигнатура ajax() с типами

Файлы определений типов могут содержать только объявления типов. В случае с jQuery ее объявления типов обернуты в интерфейс с несколькими свойствами и объявлениями методов как для `ajax()` в предыдущем листинге.

В некоторых файлах d.ts вы увидите использование слова `declare`:

```
declare const Sizzle: SizzleStatic;

export declare function findNodes(node: ts.Node): ts.Node[];
```

Мы не объявляем здесь `const Sizzle` или функцию `findNodes()`, а просто утверждаем, что собираемся использовать JS-библиотеку, содержащую объявления `const Sizzle` и `findNodes()`. Другими словами, эта строка пытается успокоить tsc: «Не ругайся, если увидишь в моем TS-коде `Sizzle` или `findNodes()`, так как при выполнении приложение включит JS-библиотеку, содержащую эти типы». Подобные объявления известны как `ambient declarations` (внешние объявления) — таким образом вы сообщаете компилятору, что рассматриваемая переменная будет существовать при выполнении. Если у вас нет определений типов для jQuery, вы можете просто написать `declare var $: any` в TS-коде и для обращения к API jQuery использовать переменную `$`. Только не забудьте загрузить jQuery вместе с вашим приложением.

ИСПОЛЬЗОВАНИЕ JAVASCRIPT-БИБЛИОТЕК БЕЗ ФАЙЛОВ ОПРЕДЕЛЕНИЙ ТИПОВ

Несмотря на то что файлы определений типов являются предпочтительным способом использования JS-библиотек в приложениях TS, вы можете использовать эти библиотеки, даже не имея этих файлов. Если вы знаете глобальную переменную выбранного JavaScript-фреймворка (например, `$` в jQuery), то можете использовать ее как есть. Современные JS-библиотеки могут использовать системы модулей, и вместо предложения глобальных переменных они могут требовать, чтобы конкретный член модуля был импортирован в ваш код. За подробностями обратитесь к документации интересующей вас библиотеки.

Возьмем jQuery, которая является набором UI-виджетов и тем, созданных на основе jQuery. Давайте предположим, что файл определений типов для jQueryUI не существует (даже несмотря на то, что это не так).

Стартовое руководство jQuery (http://learn.jquery.com/jquery-ui/getting-started) утверждает, что для использования этой библиотеки в веб-пакете вам требуется установить ее локально и добавить в HTML-документ следующий код:

```
<link rel="stylesheet" href="jquery-ui.min.css">  <-- Добавляет CSS
<script src="external/jquery/jquery.js"></script>  <-- Добавляет jQuery
<script src="jquery-ui.min.js"></script>  <-- Добавляет jQueryUI
```

Когда это сделано, вы можете добавить jQueryUI-виджеты в TS-код.

Чтобы получить доступ к jQuery, вам по-прежнему нужно использовать глобальную переменную `$`. Например, если у вас есть выпадающий список HTML `<select id="customers">`, вы можете преобразовать его в выпадающий список jQuery `selectMenu()` так:

```
$("#customers").selectMenu();
```

Предыдущий код будет работать, но без файла определений типов вы не получите никакой помощи от TypeScript, и ваша IDE выделит API jQueryUI как содержащий ошибки.

Конечно, вы можете «исправить» все ошибки tsc с помощью следующего внешнего объявления типа:

```
declare const $: any;
```

Но всегда лучше использовать файл определения типов, если есть такая возможность.

Как вы видите, файлы определений типов позволяют сразу поймать двух зайцев: использовать существующие JavaScript-библиотеки и наслаждаться преимуществами строго типизированного языка.

ПРИМЕЧАНИЕ Некоторые JS-библиотеки включают файлы d.ts, и их уже не приходится устанавливать отдельно. Хороший пример этого — библиотека moment.js, используемая для проверки, управления и форматирования дат. Посетите ее репозиторий по следующему адресу: https://github.com/moment/moment, и вы увидите файл moment.d.ts.

7.1.3. Shim и определения типов

Shim — это библиотека, которая перехватывает вызовы API и преобразует код, чтобы устаревшие среды (вроде IE 11) могли поддерживать новейшие API (вроде ES6). Например, ES6 представил для массивов метод `find()`, который находит первый элемент, отвечающий заданным критериям. В следующем листинге значение `index` будет равно 4, потому что это первое значение больше 3.

Листинг 7.3. Использование метода find() для массива

```
const data = [1, 2, 3, 4, 5];

const index = data.find(item => item > 3); // index = 4
```

Если ваш код должен выполняться в IE 11, который не поддерживает API ES6, следует добавить в tsconfig.json опцию компиляции `"target": ES5`. В результате ваша IDE подчеркнет метод `find()` волнистой линией как ошибку, поскольку ES5 его не поддерживал. IDE даже не предложит метод `find()` в списке автоподстановки, как показано на рис. 7.5.

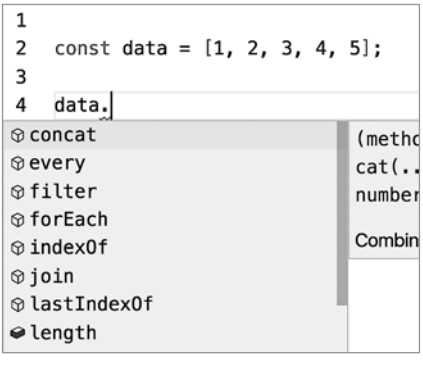


Рис. 7.5. В массивах ES5 нет метода find()

Можете ли вы продолжать использовать новейшие API и видеть их в списке автоподстановки? Да, если установите файл определений es6-shim.d.ts и добавите его в опцию компиляции types файла tsconfig.json:

```
npm install @types/es6-shim -D
```

Добавьте эту прокладку (`shim`) в файл tsconfig.json (`"types": ["jquery", "es6-shim"]`), и ваша IDE перестанет ругаться и начнет отображать метод `find()` в списке автоподстановки, как показано на рис. 7.6.

ПРИМЕЧАНИЕ Есть более новая прокладка под названием core-js (www.npmjs.com/package/core- js), которая может использоваться не только для синтаксиса ES6, но и для более новых версий спецификации ECMAScript.

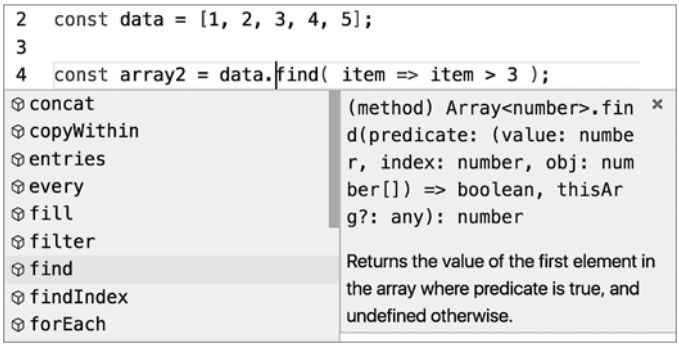


Рис. 7.6. es6-shim помогает с API ES6

7.1.4. Создание собственных файлов определений типов

Предположим, некоторое время назад вы создали JS-функцию `greeting()`, которая расположена в файле hello.js, как показано ниже:

Листинг 7.4. JavaScript файл hello.js

```
function greeting(name) {
    console.log("hello " + name);
}
```

Вы хотите продолжить использовать эту прекрасную функцию (с автоподстановкой и проверкой типов) в вашем TS-проекте. В директории src создайте файл typings.d.ts со следующим содержимым:

Листинг 7.5. Файл ./src/typings.d.ts

```
declare function greeting(name: string): void;
```

В завершение вам нужно дать TypeScript понять, где расположен этот файл определений типов. Так как эта функция `greeting()` не используется широко JavaScript-сообществом, она не опубликована на npmjs.org, и никто не создавал d.ts-файл в ветке @types. В этом случае вы можете использовать специальную TS-директиву reference (*директива с тремя слешами*), которая должна быть заменена в верхней части .ts-файла, использующего `greeting()`. На рис. 7.7 показан скриншот, сделанный в процессе набора greeti в файле main.ts в VS Code.

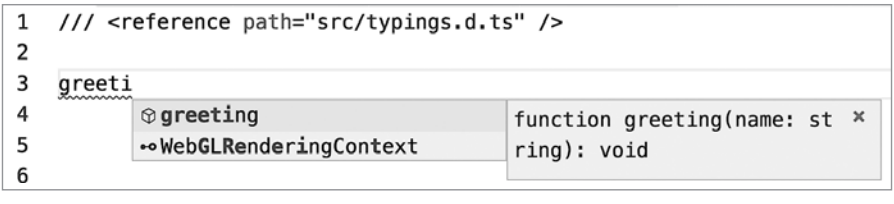


Рис. 7.7. Получение автоподстановки в main.ts

Как вы видите, автоподстановка подсказывает нам аргумент и возвращаемые типы JS-функции `greeting()`. На рис. 7.7 показана директива с тремя слешами, использующая путь к файлу определений типов, но если у вас есть файл определений типов для некоей библиотеки, установленной с помощью npm, вы можете вместо `types` использовать `path`:

```
/// <reference types="some-library" />
```

> **ПРИМЕЧАНИЕ** Если вы хотите написать файл определений типов для JS-библиотеки, прочитайте документацию TS по следующему адресу: http://mng.bz/E1qq. О директивах с тремя слешами можно прочитать более подробно здесь: http://mng.bz/NeqE.

7.2. ПРИМЕР TYPESCRIPT-ПРИЛОЖЕНИЯ, ИСПОЛЬЗУЮЩЕГО JAVASCRIPT-БИБЛИОТЕКИ

В этом разделе мы рассмотрим приложение, написанное в TS и использующее библиотеку jQuery UI. Это простое приложение будет отображать три фигуры: прямоугольник, круг и треугольник (рис. 7.8).

Если вы читаете печатную версию этой книги, знайте, что прямоугольник синий, а круг и треугольник зеленые. Пользователь может ввести допустимый CSS-селектор, и поле ввода отобразит выпадающий список с именами фигур, содержащими переданный селектор.

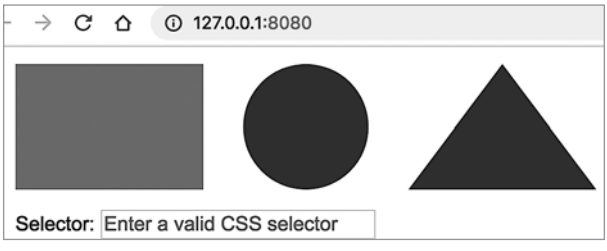


Рис. 7.8. Три фигуры, отображенные jQuery UI

На рис. 7.9 показан скриншот, сделанный после того, как пользователь указал `.green` в поле ввода и выбрал из выпадающего списка треугольник; треугольник был выделен красной рамкой.

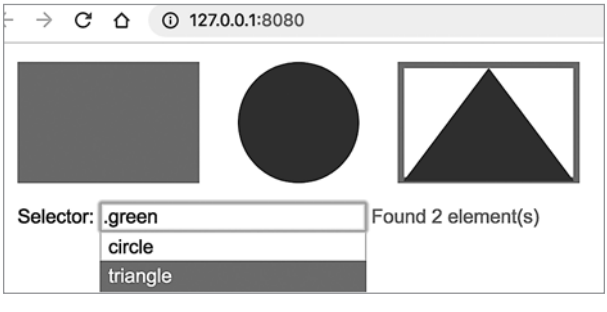


Рис. 7.9. Нахождение элемента, использующего CSS класс .green

В этом приложении мы используем jQuery для поиска HTML-элементов, имеющих один из указанных селекторов; отображение фигур выполняется jQuery UI. Этот образец проекта расположен в директории chapter7/jquery/-ui-example и включает четыре файла: package.json, tsconfig.json, index.ts и index.html. Содержимое файла package.json показано в следующем листинге.

Листинг 7.6. package.json включает файлы определений типов для jQuery и jQuery UI

```
{
    "name": "jquery-ui-example",
    "description": "Code sample for the TypeScript Quickly book",
```

```
    "homepage": "https://www.manning.com/books/typescript-quickly",
    "license": "MIT",
    "devDependencies": {
        "@types/jquery": "^3.3.29",   <-- Определение типов для jQuery
        "@types/jqueryui": "^1.12.7",   <-- Определение типов для jQueryUI
        "typescript": "^3.4.1"   <-- Компилятор TypeScript
    }
}
```

Как вы видите, мы не добавили в package.json библиотеки jQuery и jQuery UI, потому что мы внесли три дополнительные строки, показанные в следующем листинге, в раздел `<head>` файла index.html.

Листинг 7.7. Добавление jQuery и jQueryUI в index.html

```
<link rel="stylesheet"
➥ href="//code.jquery.com/ui/1.12.1/themes/base/jquery-ui.css">
<script src="//code.jquery.com/jquery-3.3.1.min.js"></script>
<script src="//code.jquery.com/ui/1.12.1/jquery-ui.min.js"> </script>
```

Добавляет стили jQuery UI

Добавляет библиотеку jQuery

Добавляет библиотеку jQuery UI

Но почему мы не добавили раздел dependencies в package.json, как делаем со всеми остальными npm-пакетами? Локально установленный jQuery UI не включал связанную версию этой библиотеки, и мы не хотели компилировать приложение, добавляя Webpack или другой бандлер. Поэтому мы решили найти URL-сети, предоставляющей содержимое (CDN) для этих библиотек.

На главной странице jQuery (jQuery.com) есть кнопка Download, которая переносит вас на страницу загрузки (http://jquery.com/download), включающую необходимые URL CDN. Если вам нужно добавить JS-библиотеку в свой проект, вам потребуется выполнить аналогичный процесс поиска.

Раздел `<head>` нашего файла index.html также включает стили, показанные в листинге 7.8. В нашем TS-коде мы будем использовать jQuery для получения HTML-элементов с помощью ID `#shapes`, `#error` и `#info`.

> **ПРИМЕЧАНИЕ** В этом демоприложении мы используем селекторы jQuery для поиска элементов на странице, но эти селекторы уже поддерживаются стандартными методами `document.querySelector()` и `document.querySelectorAll()`. Мы используем jQuery, только чтобы показать вам, как TS-код может работать с JS-библиотеками.

Пользователь сможет набрать в поле ввода любой стиль CSS, допустимый внутри элемента DOM с ID `#shapes`. Он увидит результат в списке автоподстановки, как ранее было показано на рис. 7.9.

Листинг 7.8. Фрагмент из тега <style> в index.html

```
<style>
    #shapes {
        display: flex;
        margin-bottom: 16px;
    }

    #shapes > *:not(:last-child) {
        margin-right: 32px;
    }

    @media (max-width: 640px) {
        #shapes {
            flex-direction: column;
            align-items: center;
        }

        #shapes > *:not(:last-child) {
            margin-bottom: 16px;
            margin-right: 0;
        }
    }

    #rectangle {
        background-color: blue;
        height: 100px;
        width: 150px;
    }

    #circle {
        background-color: green;
        border-radius: 50%;
        height: 100px;
        width: 100px;
    }

    #triangle {
        color: green;
        height: 100px;
        width: 150px;
    }
</style>
```

Изменяет макет страницы для устройств уже, чем 640 пикселов

Прямоугольник синий

Круг и треугольник зеленые

Медиа-запрос `@media (max-width: 640px)` сообщает браузеру о необходимости изменения макета страниц (менее 640 пикселов в ширину).

Стиль `flex-direction: column` отобразит фигуры по вертикали, а `align-items: center` центрирует их на странице, как показано на рис. 7.10.

Раздел <body> файла index.html имеет два контейнера, реализованных в тегах `<div>`. Первый, `<div id="shapes">`, имеет дочерние теги `<div>`, представляющие фигуры. Последний контейнер включает поле ввода для указания критерия поиска,

а также две области для отображения ошибки или информационного сообщения (например, «Found 2 element(s)» (Найдено два элемента), как на рис. 7.9).

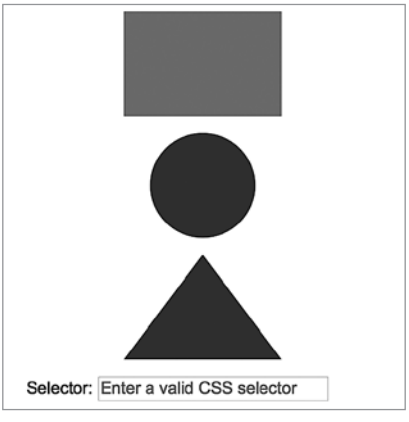


Рис. 7.10. Нахождение элемента DOM, имеющего CSS класс .green

Листинг 7.9. Раздел body файла index.html

```
<body>
    <div id="shapes">
        <div id="rectangle"
            class="blue"
            hasAngles>
        </div>

        <div id="circle"
            class="green"></div>

        <div id="triangle"
            class="green"
            hasAngles>
          <svg viewBox="0 0 150 100">
           <polygon points="75, 0, 150, 100, 0, 100" fill="currentColor"/>
          </svg>
        </div>
    </div>

    <div class="ui-widget">
        <label for="selector">Selector:</label>
        <input id="selector" placeholder="Enter a valid CSS selector">
        <span id="error"></span>
        <span id="info"></span>
    </div>

    <script src="dist/index.js"></script>
</body>
```

Контейнер с фигурами

Любой из этих CSS-атрибутов может использоваться для поиска фигуры без UI

Здесь будет выведено сообщение об ошибке

Здесь отобразится информационное сообщение

Этот скрипт является скомпилированной версией index.ts

В поле для ввода может быть указан любой из допустимых селекторов вроде `div`, `.green`, `hasAngles`. Мы дополнительно добавили прямоугольнику и треугольнику атрибут `hasAngles`, чтобы позволить поиск фигур через указание в поле для ввода селектора `[hasAngles]`.

Главная цель этого приложения — продемонстрировать использование виджета автоподстановки jQuery UI из TypeScript-кода. Он дает пользователям возможность быстро находить и выбирать из предварительно заданного списка значений по мере их ввода, повышая эффективность поиска и фильтрации, как вы видели на рис. 7.9. Если пользователь введет `.green`, приложение отобразит элементы DOM, имеющие этот CSS-селектор, и добавит их в исходный список значений виджета автоподстановки.

Виджет автоподстановки описан в документации к UI jQuery здесь: http://api.jqueryui.com/autocomplete. Он требует наличия объекта options с обязательным свойством `source`, определяющим данные для использования.

Листинг 7.10. Использование виджета автоподстановки

```
$('#selector')   <-- Наше поле <input> имеет id="#selector"
    .autocomplete({   <-- Добавляет виджет автоподстановки jQuery UI
    source: (request,   <-- Функция, получающая данные и возвращающая обратный вызов
        response) => {...}});   <-- Обратный вызов, вызываемый при выборе значения из списка
```

В коде листинга 7.10 нет типов, но поскольку у нас установлен файл определений типов jQuery UI, VS Code предлагает автоподстановку API, как показано на рис. 7.11. Обратите внимание на цифру 9 со стрелками вверх и вниз. jQuery UI предлагает много разных способов вызова `autocomplete()`, и, щелкая по стрелкам, вы можете выбрать желаемый API.

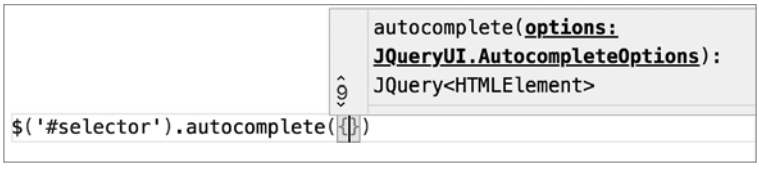


Рис. 7.11. Первая подсказка от VS Code

Подсказка на рис. 7.11 указывает, что объект `option` имеет тип `JQueryUI.AutocompleteOptions`. Вы всегда можете нажать Ctrl-пробел, и IDE продолжит помогать вам. Согласно документации виджета, нам нужно предоставить источник значений для автоподстановки, и VS Code помимо прочих содержит опцию source, как показано на рис. 7.12.

Рис. 7.12. VS Code продолжает подсказывать

Удерживая Cmd (для Mac) или Ctrl (для Windows), щелкните на списке autocomplete, откроется файл index.ts с возможными опциями. Снова удерживайте нажатой кнопку Cmd и щелкните уже по JQueryUI.Autocomplete-options, чтобы увидеть ее определения типов, как показано на рис. 7.13.

Помощь IDE с автоподстановкой не всегда идеальна. Она хороша настолько, насколько хорош предоставленный файл определений типов. Взгляните еще раз на свойство `source` на рис. 7.13. Оно объявлено как `any`, и комментарий утверждает, что оно может быть массивом, строкой или функцией. Это объявление можно улучшить, определив тип объединения, который будет допускать только эти типы:

```
type arrayOrFunction = Array<any> | string | Function;

let source: arrayOrFunction = (request, response) => 123;
```

Введение типа `arrayOrFunction` устранит необходимость написания комментария `// [ ], string, or ( )`. Естественно, вам потребуется заменить `123` на код, обрабатывающий `request` и `response`.

> **ПРИМЕЧАНИЕ** Как вы можете себе представить, код библиотеки и API, перечисленные в файле d.ts, могут выйти из синхронизации. Здесь мы зависим от благоразумия обслуживающих код людей, которые отвечают за актуальность определений типов.

Теперь давайте рассмотрим TS-код в нашем файле index.ts, показанном в листинге 7.11. Если вы занимались разработкой веб-приложений 10 лет назад, то узнаете jQuery-стиль написания кода: мы начинаем с получения ссылок на элементы DOM страницы браузера. Например, `$('#shapes')` означает, что мы хотим найти ссылку на элемент DOM с `id="shapes"`.

```
interface AutocompleteOptions extends AutocompleteEvents {
    appendTo?: any; //Selector;
    autoFocus?: boolean;
    delay?: number;
    disabled?: boolean;
    minLength?: number;
    position?: any; // object
    source?: any; // [], string or ()
    classes?: AutocompleteClasses;
}
```

Объявление типа можно улучшить

Рис. 7.13. Определения типов JQueryUI.AutocompleteOptions

Листинг 7.11. index.ts: исходный код нашего приложения

```
const shapesElement = $('#shapes');
const errorElement = $('#error');
const infoElement = $('#info');

$('#selector')
    .autocomplete({
    source: (request: { term: string },
             response: ([]) => void) => {
        try {
          const elements = $(request.term, shapesElement);
          const ids = elements.map((_index, dom) => ({ label: $(dom).attr('id'),
                      value: request.term })).toArray();
          response(ids);
          infoElement.text(`Found ${elements.length} element(s)`);
          errorElement.text('');
        } catch (e) {
          response([]);
          infoElement.text('');
          errorElement.text('Invalid selector');
          $('*', shapesElement).css({ border: 'none' });
        }
    },
    focus: (_event, ui) => {
      $('*', shapesElement).css({ border: 'none' });
      $(`#${ui.item.label}`, shapesElement).css({ border: '5px solid red' });
    }
});

$('#selector').on('input', (event: JQuery.TriggeredEvent<HTMLInputElement>)
➥ => {
```

Использует jQuery для поиска ссылок на элементы DOM

Первым параметром функции является критерий поиска

Второй параметр — это обратный вызов для изменения DOM

Находит элементы с фигурами, отвечающими критерию поиска

Находит ID фигур, отвечающих критерию

Активирует обратный вызов, передавая значения автоподстановки

Обрабатывает событие, запускаемое при перемещении фокуса на один из ID

Удаляет рамки форм, если они есть

Добавляет красную рамку элементу DOM с выбранным ID

Сбрасывает все предыдущие выборы и сообщения

```
  if (!event.target.value) {
    $('*', shapesElement).css({ border: 'none' });
    errorElement.text('');
    infoElement.text('');
  }
});
```

Мы передаем виджету автоподстановки объект с двумя свойствами: `source` и `focus`. Свойство `source` является функцией, получающей два параметра:

- `request` — объект с поисковым критерием вроде `{term: '.green'}`.
- `response` — обратный вызов, ищущий ID элементов DOM, отвечающих поисковому критерию. В нашем случае мы передаем в обратный вызов функцию, содержащую блок `try/catch`.

Свойство `focus` также является функцией, а именно обработчиком событий, который вызывается при перемещении курсора на один из элементов отображенного списка. Здесь мы удаляем рамку с выделенной ранее фигуры и добавляем ее к уже выбранной в данный момент.

Перед запуском этого приложения нужно скомпилировать TS-код в JavaScript. Компилятор TypeScript будет использовать следующие опции компиляции:

Листинг 7.12. tsconfig.json: конфигурация компилятора

```
{
    "compilerOptions": {
        "outDir": "dist",    <-- Куда разместить скомпилированный JavaScript
        "target": "es2018"   <-- Компилирует в JavaScript, совместимый со спецификацией ES2018
    }
}
```

В предыдущих главах для запуска локально установленной версии нужного нам исполняемого файла мы создавали команду скрипта npm в package.json. Например, добавление команды `"tsc": "tsc"` в раздел scripts файла package.json позволило бы нам запустить локально установленный компилятор следующим образом:

```
npm run tsc
```

На этот раз мы поленились и конфигурировать эту команду не стали. На самом деле мы хотели показать вам инструмент npx (являющийся частью npm). Он запускает локально установленную программу, если она существует, или сперва устанавливает запрашиваемую программу временно и уже потом также ее запускает. Вот как можно скомпилировать файл index.ts, используя локально установленный tsc с инструментом npx:

```
npx tsc
```

После выполнения этой команды вы увидите файл index.js в директории dist. Этот файл используется в index.html следующим образом:

```
<script src="dist/index.js"></script>
```

На этом почти всё. Единственный недостающий участник — это веб-сервер, который сможет предоставить наше приложение браузеру. Одним из простейших подобных серверов является live-server (www.npmjs.com/package/live-server). Давайте его установим:

```
npm i live-server -g
```

> **ПРИМЕЧАНИЕ** Вместо ручной установки live-server вы можете запустить его как npx live-server. Если live-server не будет обнаружен в node_modules проекта, npx загрузит его с npmjs.com, кэширует глобально на компьютере и запустит бинарный файл live-server.

Чтобы запустить сервер, введите следующую команду в терминале в корневой директории вашего проекта:

```
live-server
```

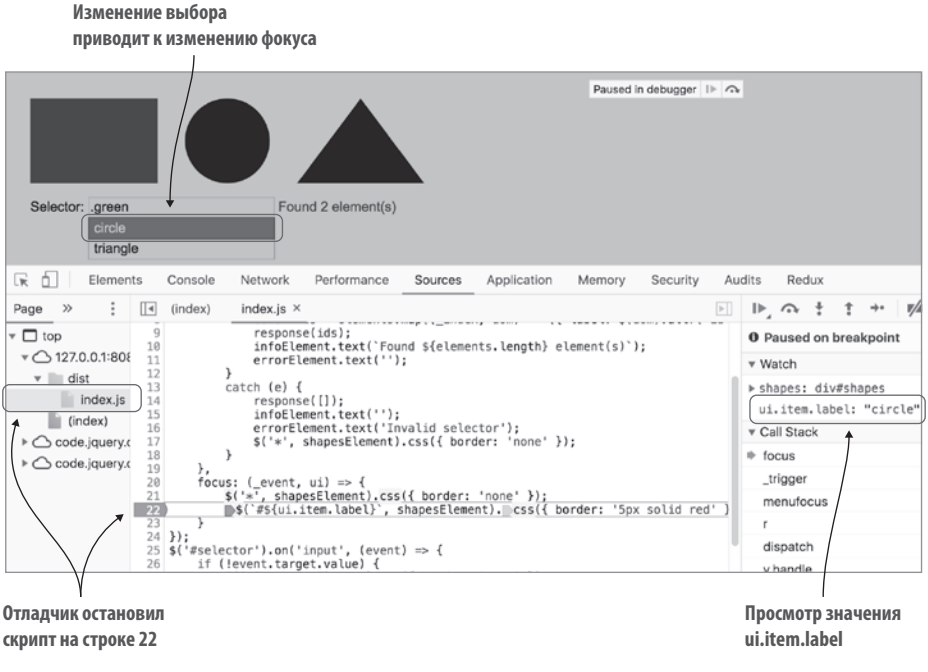


Рис. 7.14. Прерывание у точки останова в dist/index.js

Направьте браузер на localhost:8080, и вы увидите выполняемое приложение. Лучший способ понять, как работает код, — это прогнать его через отладчик; на рис. 7.14 показано приложение, запущенное в Chrome. Оно прервано при выполнении сценария dist/index.js на точке останова, определенной на строке 22, вызываемой при срабатывании события `focus`. В панели просмотра справа мы добавили ui.item.label, и она имеет значение `circle`, совпадающее с выбором в UI.

ПРИМЕЧАНИЕ В разделе 6.1 мы объясняли, что наличие карт кода позволяет вам отлаживать TS-код. Просто добавьте строку `"sourceMap": true` в файл tsconfig.json, и во время выполнения index.js вы также сможете отлаживать index.ts.

Теперь, когда мы обсудили использование сторонних JS-библиотек с TypeScript-кодом, давайте рассмотрим другой сценарий: у вас уже есть приложение, написанное в JavaScript, и вы подумываете о переходе на TS.

7.3. ВВЕДЕНИЕ TYPESCRIPT В JAVASCRIPT-ПРОЕКТ

В идеальном мире вы всегда работаете с новейшими языками и технологиями, но в реальности чаще всего имеете дело с комбинацией старых и новых средств. Предположим, что вы корпоративный разработчик и ваша команда работала над JS-приложением долгие годы, но после прочтения этой книги у вас возникло сильное желание начать использовать TypeScript. Вы всегда можете придумать какой-нибудь домашний проект и разработать его в TypeScript в свободное время, но возможно ли начать использовать TS в основном JS-проекте на работе?

TypeScript поддерживает опциональную типизацию. Это означает, что вам не обязательно модифицировать JS-код для объявления типов каждой переменной или параметра функции, так почему бы не использовать TS-компилятор в вашем JS-приложении? Что ж, ваша база кода JS-приложения может содержать десятки тысяч строк, и использование tsc для компиляции их всех может вскрыть множество притаившихся багов, замедлив тем самым процесс развертывания. Такой вариант будет не самым лучшим началом.

Вместо этого выберите часть приложения, которая реализует некоторую изолированную функциональность (вроде модуля для добавления нового клиента или доставки), и прогоните его через tsc как есть. Вероятнее всего, процесс сборки вашего приложения уже использует такие инструменты, как Grunt, Gulp, Babel, Webpack и пр. Найдите удачное место и внедрите tsc в этот процесс.

Вам даже не потребуется переименовывать JS-файлы, чтобы задать им расширение .ts. Просто используйте опцию TS-компилятора `"allowJs": true`, которая

говорит tsc: «Пожалуйста, скомпилируй не только .ts-файлы, но и файлы .js и не выполняй проверку типов — просто скомпилируй их согласно установленной опции `target`».

> **ПРИМЕЧАНИЕ** Если вы не измените расширение с .js на .ts, ваша IDE по-прежнему будет выделять типы JS-файлов как ошибочные, но если вы используете опцию `"allowJs": true`, tsc будет компилировать файлы.

Зачем просить tsc пропустить проверку типов, если они все равно считаются опциональными? Одна из причин в том, что tsc может оказаться не способен полноценно вывести всю информацию типов из вашего JS-кода и начнет сообщать об ошибках. Вторая причина в том, что ваш существующий код может иметь баги (даже если эти баги не привлекают особого внимания) и полноценный анализ tsc может раскрыть множественные ошибки компиляции, на исправление которых у вас просто нет времени и ресурсов. Если не ломается, значит, и чинить не стоит, верно?

Конечно, вы можете использовать и другой подход: «Если не ломается, улучшайте». Если вы уверены в том, что ваш JS-код написан хорошо, активируйте проверку типов, добавив опцию компилятора `"checkJs": true` в tsconfig.json. Некоторые из ваших JS-файлов могут продолжить генерировать ошибки, и вы можете пропускать их проверку, добавив к ним комментарий `//@ts-nocheck`. И наоборот, вы можете выбрать для проверки только некоторые .js-файлы, добавив комментарий `//@ts-check`, не указывая при этом `"checkJs": true`. Вы можете даже выключить проверку типов для конкретной строки кода, добавив `//@ts-ignore` в предшествующую ей строку.

Чтобы продемонстрировать эффект проверки типов существующего JS-кода, мы случайным образом выбрали файл OpenAjax.js из GitHub репозитория фреймворка Dojo (https://github.com/dojo/dojo/blob/master/OpenAjax.js). Давайте представим, что хотим начать преобразование этого кода в TypeScript, поэтому мы добавили комментарий `//@ts-check` в начало файла. Вы увидите, что в VS Code некоторые строки будут подчеркнуты волнистой линией, как показано на рис. 7.15.

Давайте проигнорируем ошибки в инструкциях импорта сверху; мы бы не увидели их, если бы все эти файлы были в наличии. Похоже, что ошибка в строке 8 тоже не является реальной ошибкой. Скорее всего, объект OpenAjax будет присутствовать при выполнении. Добавление `//@ts-ignore` над строкой 8 устранит эту волнистую линию.

Но для исправления ошибок в строках 19–23 понадобится объявить `type` или `interface` со всеми этими свойствами. (Обратите внимание, что если бы вы изменяли этот код в TypeScript, то скорее предпочли бы присвоить переменной `h` более содержательное имя.)

Рис. 7.15. Добавление комментария //@ts-check в начало файла JavaScript

Давайте рассмотрим другой фрагмент JS-кода (рис. 7.16). Нужно получить цену продукта, и если она будет менее $20, мы его купим. IDE не жалуется, и код выглядит приемлемым.

```
1  const getPrice = () => Math.random()*100;
2
3  if (getPrice < 20) {
4      console.log("Buying!");
5  }
```

Рис. 7.16. JavaScript-код с ошибками

Давайте добавим комментарий `//@ts-check` в первую строку, чтобы статический анализатор типов мог проверить этот код на правильность, как показано на рис. 7.17.

Упс! Наш JS-код содержит баг — мы забыли добавить скобки после `getPrice` в инструкции `if` (мы бы не смогли вызвать эту функцию). Если вам было

интересно, почему этот код никогда не выдавал вам Ok для покупки продукта, то теперь вы знаете причину: выражение `getPrice < 20` никогда не вычислялось как `true`. Простое добавление `@ts-check` в начало кода помогло нам обнаружить баг среды выполнения JavaScript-программы.

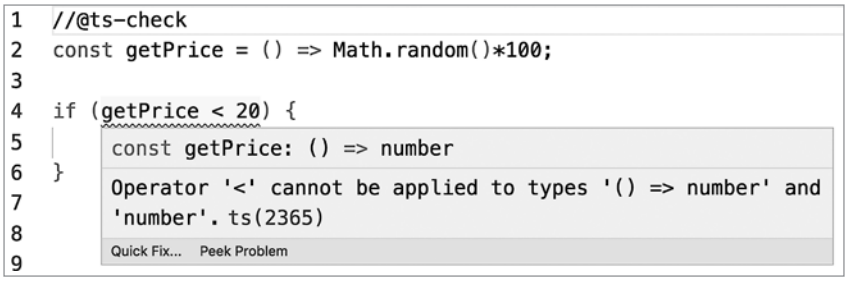


Рис. 7.17. @ts-check обнаружила баг

Есть еще одна опция tsc, `noImplicitAny`. Она поможет вам при переносе JS-проекта в TS. Если вы не собираетесь указывать типы параметров функций и возвращаемые типы, tsc может испытывать затруднения при их выводе, и вы можете временно использовать опцию компилятора `"noImplicitAny": false` (установлена по умолчанию). В этом режиме, если tsc не может вывести тип переменной на основе ее использования, компилятор молча присваивает ей тип any. Именно это имеется в виду под `implicit any` (неявные `any`). Но в идеале `noImplicitAny` должна быть установлена как `true`, так что не забудьте активировать ее по завершении переноса проекта в TypeScript.

> **ПРИМЕЧАНИЕ** После включения `noImplicitAny` добавьте еще одну опцию, `sctrictNullChecks`. Она будет перехватывать все возможные случаи, которые могут содержать `null` или `undefined` там, где ожидаются другие значения.

Компилятор TypeScript будет снисходителен к вашим файлам .js. Он позволит добавлять свойства классу или функции после их объявления. То же касается и объектных литералов в файлах .js: вы можете добавлять свойства объектным литералам, даже если они не были определены изначально. TypeScript поддерживает формат модулей CommoJS и распознает вызовы функции `require()` как импорты модуля. Все параметры функций по умолчанию являются опциональными, и допускаются вызовы с меньшим числом аргументов, чем объявленное число параметров.

Вы также можете помочь tsc с выводом типов, добавив JSDoc-аннотации (вроде `@param` и `@return`) в ваш JavaScript-код. Подробнее по этой теме читайте в документе «JSDoc support in JavaScript» на GitHub по адресу http://mng.bz/DNqy.

ЕЩЕ РАЗ ПОВТОРИМСЯ: ПОЧЕМУ TYPESCRIPT?

TS — это не первая попытка создания альтернативы JavaScript, которая может запускаться как в браузере, так и в обособленном JS-движке. TypeScript всего семь лет, но он уже находится в топ-10 языков программирования по различным рейтингам. Почему этот список топ-10 не включает более старые языки вроде CoffeeScript или Dart, которые, как ожидалось, должны были стать альтернативными способами написания JavaScript?

По нашему мнению, есть три главные особенности, которые выделяют TypeScript:

- TypeScript строго следует стандартам ECMAScript. Если предложенная функция достигает стадии 3 процесса внедрения, она без промедления включается в TypeScript.
- IDE TS работают с таким же статическим анализатором типов, постоянно предлагая вам помощь в процессе написания кода.
- TypeScript с легкостью взаимодействует с JS-кодом, в связи с чем вы можете использовать тысячи существующих JS-библиотек в ваших TS-приложениях (и в этой главе вы как раз научились это делать).

ПРИМЕЧАНИЕ CLI-инструмент для проверки охвата типов (www.npmjs.com/package/type-coverage) позволяет посчитать все идентификаторы в приложении, объявленные с явным типом (кроме `any`), и сообщает охват типов в процентах.

Процесс переноса вашего JS-проекта в TS не особо сложен, и мы предоставили вам высокоуровневый обзор одного подхода для его поэтапного выполнения. Более подробно можете прочитать об этом в документе «Migrating from JavaScript» (Перенос из JavaScript) по ссылке http://mng.bz/lolj. Там вы сможете найти особенности интеграции с Gulp и Webpack, преобразование приложений React.js в TypeScript и др.

На этом завершается первая часть, познакомившая вас с TypeScript. Мы не рассматривали каждую возможность языка, но это и не планировалось. Книга же называется «TypeScript быстро», не так ли?

Если вы поняли весь материал из первой части, то с легкостью сможет пройти собеседование по TypeScript. Но если вы хотите стать продуктивным TS-разработчиком, призываем вас изучить и проработать все примеры приложений, описанные во второй части.

ИТОГИ

- Вы можете использовать тысячи существующих JS-библиотек в вашем TS-проекте.
- Файлы определений типов позволяют вам наслаждаться проверкой типов и функцией автоподстановки в библиотеках, написанных в JavaScript. Эти файлы повышают вашу продуктивность написания кода.
- Вы можете создавать файлы определений типов для любого проприетарного JS-кода.
- Даже если JS-библиотека не имеет файлов определений типов, вы все равно можете использовать ее в своем TS-проекте.
- Вы можете использовать хорошо прописанные шаги, которые позволят вам поэтапно перенести JS-код в TypeScript.

Часть 2

Использование TypeScript в блокчейн-приложении

Часть 2 включает несколько приложений, написанных с помощью популярных фреймворков. Каждое из этих приложений использует TypeScript. Начнем с объяснения принципов блокчейна, который используется для различных версий примера приложения, рассматриваемого в части 2.

Мы представим такие фреймворки и библиотеки, как Angular, React.js и Vue.js, и вы увидите пример разработки блокчейн-приложения с их помощью. Материалы, включенные в эту часть, помогут вам понять, как TypeScript используется в реальных проектах. Несмотря на то что читать все главы в части 2 не обязательно, ознакомление с главами 8 и 10 потребуется для понимания приложений, представленных в главах 12, 14 и 16.

8 Разработка собственного блокчейн-приложения

В этой главе:

- ✓ Принципы работы блокчейн-приложений.
- ✓ Назначение хеш-функций.
- ✓ Объяснение добычи блоков.
- ✓ Разработка простого блокчейн-приложения.

В этой главе мы представим образец приложения, использующего TS совместно с технологией блокчейн. Большинство из последующих глав будут включать различные версии этого блокчейн-приложения, поскольку мы будем применять популярные библиотеки и фреймворки, которые могут использоваться с TS.

Использование нетривиальной технологии для разработки приложения в этой книге может вас удивить, но есть несколько причин, чтобы оценить знакомство с блокчейном:

- Как правило, нас интересует повышение доверия, точности и прозрачности в рабочем процессе с участием нескольких сторон. Блокчейн отлично подходит для перечисленных задач, в то время как TS отлично подойдет для их реализации.
- Мы каждый день слышим о все новых утечках данных. Вы можете на сто процентов быть уверены, что никто не переведет с вашего банковского счета тысячу долларов без вашего ведома? Только при условии, если у вас нет этой

тысячи долларов. Блокчейн исключает малейшие риски подобных инцидентов, так как в этой системе не существует единичных структур (наподобие банка), владеющих вашими данными, а изменение этих данных требует подтверждения большинством участников блокчейна.

- Новые криптовалюты запускаются практически ежедневно, и востребованность разработчиков, понимающих лежащие в их основе технологии, будет только расти.
- Новые высокооплачиваемые вакансии открываются в организациях, реализующих блокчейн в своих операциях, к которым относятся финансовые транзакции, выборы, логистика, установка интернет-идентичности, интернет-реклама, совместные поездки и т. д.

Так как сама идея блокчейна пока еще нова, мы начнем с объяснения основных принципов работы этой технологии и ее назначения.

8.1. БЛОКЧЕЙН 101

Блокчейны могут использоваться для приложений различного рода, но именно благодаря финансовым приложениям термин «блокчейн» закрепился как устоявшийся. Скорее всего, вы уже слышали о криптовалютах в общем и о биткоине в частности. Очень часто вы будете встречать слова «биткоин» и «блокчейн» в одном приложении. При этом блокчейн является децентрализованным способом хранения неизменяемых данных, а биткоин — это криптовалюта, использующая конкретную реализацию блокчейна. Другими словами, биткоин относится к блокчейну так же, как данные вашего приложения к СУБД.

Криптовалюта не имеет физических счетов или монет, но может использоваться для купли/продажи вещей и услуг.

Здесь возникает логичный вопрос: «Если не существует счетов и банки не вовлечены в эту систему, то как одна сторона может быть уверена, что вторая оплатила предоставленные услуги или товары?» В блокчейне транзакции объединяются в блоки, которые затем проверяются и связываются в цепочку. Отсюда и происходит термин *блокчейн* (blockchain).

Представьте цепь велосипеда: на рис. 8.1 изображен блокчейн, состоящий из трех блоков (цепных связей). Если в блокчейн нужно добавить записи новых транзакций, приложение создает новый блок, передаваемый узлам блокчейна (компьютерам) для проверки при помощи одного из алгоритмов (например, вычисления особого хеш-кода), используемого в системах блокчейн. Если блок оказывается действительным, то добавляется в блокчейн; в противном случае он отвергается.

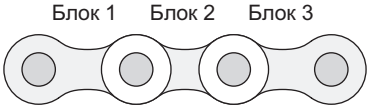


Рис. 8.1. Блокчейн, состоящий из трех блоков

Где хранятся данные о транзакциях и что в этом контексте означает слово *децентрализованный*? Типичный блокчейн децентрализован, так как ни один человек или компания не контролируют данные и не владеют ими. Децентрализованный — значит полное отсутствие потенциальных точек сбоя.

Представьте себе сервер, содержащий информацию о доступных местах на ряде авиарейсов. Множество туристических агентств соединяются с этим сервером, чтобы просмотреть его данные и забронировать билеты.

Одни агентства (узлы) малы и имеют всего один компьютер, подключенный к этому серверу. В некоторых агентствах узлы составляют два, три или более аналогичных компьютера, но при этом все они зависят от данных одного сервера. Это называется *централизованной обработкой данных*, схематично она изображена на рис. 8.2. Если сервер упадет, уже никто не сможет забронировать билеты.

В случае же децентрализованной обработки, которая применяется в большинстве блокчейнов, не существует сервера с данными. Полные копии блокчейна хранятся в узлах *одноранговой* сети, которая может также включать и ваши компьютеры, если вы решите присоединиться к блокчейну. Один узел не означает один компьютер — вы можете владеть компьютерным кластером, представляющим один узел. На рис. 8.3 изображена децентрализованная сеть. Если один из узлов упадет, то система продолжит функционировать до тех пор, пока хотя бы один узел будет активен.

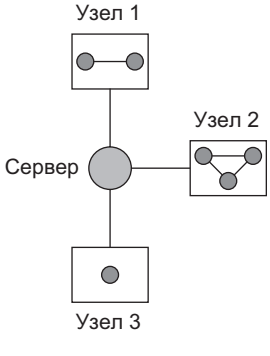


Рис. 8.2. Централизованная обработка данных

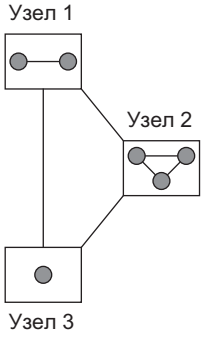


Рис. 8.3. Децентрализованная обработка данных

ПРИМЕЧАНИЕ Возможность передавать цифровую валюту между одноранговыми узлами без участия центрального органа управления стала решающим фактором в популяризации технологии блокчейн. Теперь она используется и в других типах приложений, работающих с логистикой, сделками с недвижимостью, бронированием билетов и т. д.

Безопасно ли хранить копии транзакций на множестве компьютеров, принадлежащих другим людям или организациям? Что, если один из таких владельцев (плохой парень) модифицирует мою транзакцию, изменив сумму оплаты на ноль? Спешим вас обрадовать, что такое невозможно. После добавления блока в цепочку его данные не могут быть изменены, так как данные в блокчейне являются неизменяемыми.

Рассматривайте блокчейн как хранилище, в котором данные «выбиты на камне». Как только элемент данных добавляется в хранилище, вы уже не можете удалить или переместить его. Новый блок может быть вставлен только после того, как один из узлов сети решит математическую задачу. Это может звучать как магия. Поэтому лучшим способом для понимания работы блокчейна будет создание своего собственного приложения, чем мы и займемся в следующем разделе.

В своей основе блокчейн — это децентрализованный неизменяемый журнал учета, представленный коллекцией блоков. Блоки могут хранить любой тип данных, например информацию о денежных переводах, результатах голосования, медицинских записях и т. д. Каждый блок связан с предыдущим посредством хранения его хеш-значения.

В следующем разделе вкратце расскажем о хешировании.

8.1.1. Криптографические хеш-функции

Согласно определению из Википедии, хеш-функция — это математический алгоритм, отображающий данные или произвольный размер (часто называемый «сообщение») в битовую строку фиксированного размера (хеш-значение, хеш или профиль сообщения) и являющийся односторонней функцией, которую невозможно обратить (https://ru.wikipedia.org/wiki/Хеш-функция). Там также утверждается, что хеш-функции подходят для использования в криптографии.

Криптографическая хеш-функция позволяет с легкостью проверить, что вводные данные отображаются в данное хеш-значение, но если при этом вводные данные неизвестны, их намеренно трудно реконструировать (или прибегнуть к любым другим альтернативам) на основе хранящегося хеш-значения.

Шифрование задействует двустороннюю функцию, получающую значение и применяющую секретный ключ, чтобы вернуть зашифрованное значение. С помощью того же ключа зашифрованное значение можно расшифровать.

В противоположность этому *хеширующая* утилита использует необратимую функцию. Этот процесс не может быть обращен для раскрытия изначального значения. Если хеш-функция получает одно и то же вводное значение, то на выходе она производит одно и то же хеш-значение. При этом используются комплексные алгоритмы хеширования, чтобы минимизировать вероятность формирования одинакового хеш-значения разными вводными данными.

Давайте рассмотрим очень простую и не самую безопасную хеш-функцию, чтобы понять необратимую природу хешей. Допустим, приложение имеет пароли, состоящие только из чисел, и мы не хотим хранить их в чистом виде в базе данных. Нужно переписать хеш-функцию, которая применяет оператор модуля к переданному паролю и затем добавляет к нему 10.

Листинг 8.1. Очень простая хеш-функция

```
function hashMe(password: number): number{

    const hash = 10 + (password % 2);  <-- Создает хеш по модулю

    console.log(`Original password: ${password}, hashed value: ${hash}`);

    return hash;
}
```

Для любого четного числа выражение `input % 2` будет производить 0. Теперь давайте несколько раз вызовем функцию `hashMe()`, передавая в качестве вводных параметров разные четные числа:

```
hashMe (2);
hashMe (4);
hashMe (6);
hashMe (800);
```

Каждый из этих вызовов произведет хеш-значение 10, и вывод будет следующим:

```
Original password: 2, hashed value: 10
Original password: 4, hashed value: 10
Original password: 6, hashed value: 10
Original password: 800, hashed value: 10
```

Вы можете посмотреть эту функцию в действии на CodePen: http://mng.bz/BYqv.

Процесс, при котором хеш-функция генерирует одинаковый вывод для более чем одного ввода, называется *коллизией*. В криптографии различные защищенные алгоритмы хеширования (SHA) предлагают более или менее безопасные способы создания хешей и разрабатываются с учетом *противостояния коллизиям*,

максимально снижая вероятность нахождения двух вводов, производящих одинаковое хеш-значение.

Блок в блокчейне представлен хеш-значением, и очень важно, чтобы киберпреступники не могли заместить один блок другим, подготовив вредоносное содержимое, которое производит тот же хеш, что и действительный блок. В нашем примере хеш-функция, показанная на рис. 8.1, не имеет сопротивления коллизионным атакам. Блокчейны же, напротив, используют устойчивые к коллизиям алгоритмы хеширования вроде SHA-256, который получает строку любой длины и производит 256-битное хеш-значение, состоящее из 64 шестнадцатеричных чисел. Даже если вы решите сгенерировать SHA-256 хеш для всего текста этой книги, его длина будет также 64 шестнадцатеричных числа. В хешах SHA-256 существует 2^{256} возможных комбинаций битов, что превышает количество крупиц песка во всем мире.

> **ПРИМЕЧАНИЕ** Об алгоритмах SHA-256 можно почитать в Википедии: https://ru.wikipedia.org/wiki/SHA-2.

Для вычисления хеша SHA-256 в ОС Unix вы можете использовать утилиту shasum. В Windows же можно воспользоваться программой certUtil. Также существует несколько онлайн-генераторов SHA-256. Для программного создания хешей вы можете использовать модуль `crypto` в приложениях Node.js или объект `crypto` в современных браузерах.

Вот как можно вычислить SHA-256-хеш для текста `"hello world"` на macOS:

```
echo -n 'hello world' | shasum -a 256
```

Эта команда произведет следующий хеш:

```
b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9
```

Неважно, сколько раз вы повторите предыдущую команду для строки `"hello world"`, вы всегда будете получать одно и то же хеш-значение, но изменение даже одного ее знака уже приведет к генерации другого SHA-256-хеша. На языке функционального программирования можно сказать, что хеш-функция является *чистой функцией*, потому что всегда возвращает одно значение для заданного ввода.

> **ПРИМЕЧАНИЕ** Если вам интересны методологии и алгоритмы хеширования, обратитесь к обсуждению Араша Партоу (Arash Partow), посвященному именно алгоритмам хеширования (www.partow.net/programming/hashfunctions), или почитайте Википедию: https://ru.wikipedia.org/wiki/Хеш-функция.

Мы уже говорили, что блоки в блокчейне связаны при помощи хешей, а в следующем разделе познакомимся с самим внутренним содержимым блоков.

8.1.2. Из чего состоит блок?

Вы можете рассматривать блок как запись, или журнал учета. Каждый блок в блокчейне содержит данные приложения, а также имеет временную метку, свое собственное хеш-значение и хеш-значение предыдущего блока. В очень простом (и легко взламываемом) блокчейне приложение для добавления в цепочку нового блока может выполнить следующие действия:

1. Найти хеш последнего добавленного блока и сохранить его как ссылку на предыдущий блок.
2. Сгенерировать хеш-значение для нового блока.
3. Представить этот новый блок блокчейну для проверки.

Давайте рассмотрим эти шаги подробнее. В момент создания блокчейна в него был добавлен первый блок. Первый блок в цепочке называется *первичным*. Очевидно, что ему не предшествуют другие блоки, поэтому хеш предыдущего блока в нем не существует.

На рис. 8.4 показан пример блокчейна, где каждый блок имеет уникальный индекс, временную метку и два хеша — свой собственный и предыдущего блока. Обратите внимание, что в первичном блоке вместо хеша предыдущего блока мы видим пустую строку.

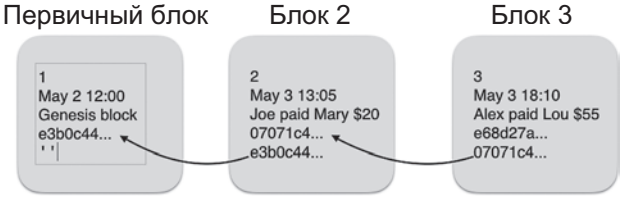


Рис. 8.4. Пример блокчейна

На рис. 8.4 в первичном блоке мы использовали пустые кавычки, чтобы показать отсутствие хеша предыдущего блока. Наши данные представлены в виде текста, описывающего транзакцию вроде «Joe paid Mary $20». В реальном блокчейне Joe и Mary были бы представлены длинными зашифрованными именами аккаунтов.

> **ПРИМЕЧАНИЕ** Можно рассматривать блокчейн как особый тип связанного списка, в котором каждый узел имеет ссылку только на предыдущий. Это не классический однонаправленный список, где каждый узел содержит ссылку на следующий.

Теперь давайте представим, что какой-то плохой парень узнал, что хешем одного из блоков нашего блокчейна является «e68d27a...». Может ли он

модифицировать его данные, утвердив, что заплатил Мэри $1000, а затем повторно сгенерировать другие блоки, чтобы хеш-значения сошлись? Дабы помешать такому сценарию, блокчейн требует разрешения алгоритмов, на что затрачиваются время и ресурсы. Именно поэтому члены блокчейна для добычи (майнинга) блока вынуждены тратить время и ресурсы, не имея возможности быстро сгенерировать хеш-значение.

8.1.3. Что такое добыча блока?

Все понимают, что означает добыча золота — вы выполняете некоторую работу, чтобы его получить, а затем обменять на деньги или товары. Чем больше людей добывают этот драгоценный металл, тем больше его оборот в мире. В прошлом золото и другие драгоценности определяли ценность бумажных денег.

В США денежное обеспечение осуществляется Федеральным резервом, состоящим из ряда коммерческих банков и возглавляемым советом директоров. Федеральный резерв имеет полномочия управлять бумажной валютой, монетами, денежными средствами на чековых и сберегательных счетах, а также прочими законными формами обмена.

Деньги в криптовалютах (вроде биткоина) «производятся» в качестве стимула для людей (майнеров) решать математические задачи, требуемые конкретным блокчейном. Поскольку каждый новый блок должен быть одобрен другими майнерами, значит, чем больше майнеров, тем безопаснее блокчейн.

В нашем распределенном блокчейне мы хотим убедиться, что могут быть добавлены только блоки с конкретными хешами. Например, наш блокчейн может требовать, чтобы каждый хеш начинался с `0000`. Хеш вычисляется на основе содержимого блока, и он не будет начинаться с четырех нулей, если только кто-нибудь не найдет дополнительное значение, чтобы добавить его к содержимому блока и в итоге произвести подобный хеш. Нахождение такого значения называется добычей блока.

Прежде чем новый блок добавляется в блокчейн, он передается всем узлам сети на обработку, и эти узлы, в свою очередь, начинают вычислять особое значение, которое производит допустимый хеш. Первый, кто находит такое значение, выигрывает. Выигрывает что? Блокчейн может предложить вознаграждение. Например, в блокчейне биткоина преуспевший добытчик данных зарабатывает биткоины.

Давайте предположим, что наш блокчейн требует, чтобы хеш каждого блока начинался с `0000`; в противном случае блок будет отвергнут. Например, приложение хочет добавить в цепочку новый блок (номер 4), и у него есть следующие данные:

```
4
June 4 15:30?
Simon refunded Al $200
```

Перед добавлением любого блока его хеш должен быть вычислен, поэтому мы конкатенируем предыдущие значения в одну строку и генерируем хеш SHA-256, выполнив следующую команду:

```
echo -n '4June 4 15:30?Simon refunded Al $200' | shasum -a 256
```

Сгенерированный хеш будет выглядеть так:

```
d6f9255c5fc579594bef56403778d475ab441abbd56bff788d597ae1e8d4ad22
```

Он не начинается с `0000`, следовательно, будет отвергнут нашим блокчейном. Нам нужно произвести добычу данных, чтобы найти значение, которое, будучи добавленным в наш блок, приведет к генерации хеша, начинающегося с `0000`. Брутфорс в помощь! Мы можем написать программу, которая будет добавлять последовательные числа (1, 2, 3 и т. д.) в конец нашей строки ввода до тех пор, пока сгенерированный хеш не будет начинаться с `0000`.

После нескольких попыток мы выяснили, что секретное значение для нашего блока — это 236499. Давайте добавим его в нашу строку ввода и повторно вычислим хеш:

```
echo -n '4June 4 15:30?Simon refunded Al $200236499' | shasum -a 256
```

Теперь сгенерированный хеш начинается с четырех нулей:

```
0000696c2bde5add287a7b6ccf9a7e57c9d69dad8a6a93922b0451a5150e6696
```

Превосходно! Мы знаем число, которое можно включить в содержимое блока, чтобы сгенерированный хеш соответствовал требованиям блокчейна. Это число может использоваться только один раз с конкретной строкой ввода. При изменении любого знака в этой строке будет генерироваться хеш, который уже не начитается с `0000`.

В криптографии число, которое может быть использовано только один раз, называется *nonce*, и значение 236499 как раз является числом *nonce* для нашего конкретного блока. Как насчет следующего: мы добавим свойство `nonce` в объект блока и позволим майнерам данных вычислить его значение для каждого нового блока?

Майнер должен потратить вычислительные ресурсы для нахождения *nonce* в качестве *доказательства проделанной работы*, являющейся необходимым условием для каждого блока при запросе на добавление его в блокчейн.

ДОБЫЧА БИТКОИНА

Теперь, когда вы видели, как новый блок может быть проверен и добавлен в блокчейн, то понимаете принцип работы добычи данных Биткоин. Предположим, есть новая транзакция в Биткоин между Джо и Мэри, и эта транзакция должна быть добавлена в блокчейн Биткоин (журнал учета). Ее нужно поместить в блок, который сперва должен быть проверен.

Любой участник блокчейна Биткоин может стать майнером данных — человеком (или фирмой), который хочет использовать свои компьютерные ресурсы, чтобы стать первым в решении сложной вычислительной задачи. Эта задача потребует намного больше вычислительных мощностей, чем рассмотренная задачка с четырьмя нулями.

Постепенно число майнеров растет, вычислительные ресурсы увеличиваются, и блокчейн Биткоин может повысить сложность задачи, используемой для добычи блока. Это делается, чтобы время, необходимое для добычи, оставалось одинаковым и составляло примерно десять минут на нахождение хешей с 15–20 стартовыми нулями.

Первый человек (например, майнер Питер), решивший задачу для конкретной транзакции (вроде «Джо заплатил Мэри пять биткоинов»), заработает только что выпущенный биткоин. Питер может также зарабатывать деньги на транзакционных сборах за добавление транзакций в блокчейн. Но что, если Питеру нравится Мэри и он решает оказать ей «услугу», сжульничав и повысив размер транзакции с пяти до пятидесяти? Такое невозможно, поскольку транзакция Джо будет содержать цифровую подпись, сделанную с использованием криптографии публичного и приватных ключей.

Ранее в этом разделе мы конкатенировали число блока, время и текст одной транзакции для вычисления хеша. Блокчейн Биткоина хранит блоки с множеством транзакций (около 2500 в каждом). Размер блока или количество транзакций не имеют большого значения. Увеличение числа начальных нулей, требуемых в хеш-коде, усложняет решение задачи.

Биткоины можно использовать для оплаты услуг, а также приобретать за условные деньги или другие криптовалюты. При этом их добыча является единственным процессом, приводящим к выпуску новых Биткоинов в оборот.

Мы напишем программу для вычисления *nonce* в процессе разработки нашего маленького блокчейна в следующем разделе. А пока давайте условимся о следующей структуре блока:

Листинг 8.2. Пример типа Block

```
interface Block {
    index: number;  <-- Последовательный номер блока
    timestamp: number;  <-- Первый параметр функции является поисковым критерием
    data: string;  <-- Данные об одной или нескольких транзакциях в приложении
    nonce: number;  <-- Число, вычисляемое майнерами
    hash: string;  <-- Хеш этого блока
    previousBlockHash: string;  <-- Хеш-значение предыдущего блока в блокчейне
}
```

Обратите внимание, что для объявления пользовательского типа `Block` мы использовали `interface` TypeScript. В следующем разделе мы решим, должен ли он оставаться `interface` или же должен стать `class`.

ЖУРНАЛЫ УЧЕТА И ПОДГОН ЦИФР

Каждому бизнесу необходимо вести учет транзакций. В прошлом эти транзакции записывались и классифицировались в книге вручную: продажи, покупки и т. д. Сегодня подобные записи хранятся в файлах, но принцип *журнала учета* остается все тем же. Для наших целей вы можете рассматривать блокчейн как представление журнала учета.

Выражение *подгон цифр* означает подделку финансовых отчетов. Но если журнал учета реализован в блокчейне, подгон цифр становится практически невозможным. Потребуется заговор более 50% узлов блокчейна, чтобы утвердить незаконное изменение блока, что относится скорее к теоретической вероятности, нежели к реальной возможности.

8.1.4. Мини-проект с хешем и nonce

Современные браузеры содержат объект `crypto` для поддержки криптографии. В частности, вы можете использовать API `crypto.subtle.digest()` для генерации хеша (см. документацию Mozilla, описывающую этот метод, здесь: http://mng.bz/dxKD).

Мы хотим дать вам в работу небольшое задание. После него мы предоставим его решение, но не будем особо его пояснять. Попробуйте понять код сами:

1. Напишите функцию `generateHash(input: string)`, получающую ввод `string` и находящую его SHA-256-хеш при помощи `crypto` API браузера.
2. Напишите функцию `calculateHashWithNonce(nonce: number)`, которая будет конкатенировать предоставленное число *nonce* со строкой ввода и вызывать `generateHash()`.

3. Напишите функцию `mine()`, которая будет циклически вызывать `calculateHashWithNonce()` до тех пор, пока сгенерированный хеш не будет начинаться с `0000`.

Функция `mine()` должна вывести сгенерированный хеш и *nonce* подобно приведенному ниже:

```
Hash: 0000bfe6af4232f78b0c8eba37a6ba6c17b9b8671473b0b82305880be077edd9,
➥ nonce: 107105
```

Следующий листинг показывает наше решение этого задания. Мы использовали ключевые слова JavaScript `async` и `await`, объясненные в приложении. Прочитайте код — вы должны понять, как он работает:

Листинг 8.3. Решение для проекта с хешем и перцем

```
import * as crypto from 'crypto';

let nonce = 0;

async function generateHash(input: string): Promise<string> {

    const msgBuffer = new TextEncoder().encode(input);

    const hashBuffer = await crypto.subtle.digest('SHA-256', msgBuffer);

    const hashArray = Array.from(new Uint8Array(hashBuffer));

    const hashHex = hashArray.map(b =>
➥ ('00' + b.toString(16)).slice(-2)).join('');
    return hashHex;
}

async function calculateHashWithNonce(nonce: number): Promise<string> {
    const data = 'Hello World' + nonce;
    return generateHash(data);
  }

async function mine(): Promise<void> {
    let hash: string;
        do {
          hash = await this.calculateHashWithNonce(++nonce);
        } while (hash.startsWith('0000') === false);

        console.log(`Hash: ${hash}, nonce: ${nonce}`);
}

mine();
```

Генерирует хеш SHA-256 из предоставленного ввода

Зашифровывает UTF-8

Хеширует сообщение

Преобразует ArrayBuffer в Array

Преобразует биты в хеш-строку

Добавляет nonce к строке, а затем вычисляет хеш

Находит nonce, который приведет к генерации хеша, начинающегося с четырех нулей

Использует await, так как это асинхронная функция

Решение можно посмотреть в действии на CodePen: http://mng.bz/rP4g. Изначально панель консоли будет пуста, но спустя несколько секунд работы она выведет следующее:

```
Hash: 0000bfe6af4232f78b0c8eba37a6ba6c17b9b8671473b0b82305880be077edd9,
➥ nonce: 107105
```

Если вы измените код метода `mine()`, заместив четыре нуля на пять, на вычисление уже потребуются минуты. Попробуйте указать десять нулей, и калькуляция может продлиться несколько часов.

Вычисление требует немало времени, но проверка выполняется быстро. Чтобы проверить, что программа в листинге 8.3 вычислила *nonce* (107105) верно, мы использовали утилиту shasum следующим образом:

```
echo -n 'Hello World107105' | shasum -a 256
```

Эта утилита вывела тот же хеш, что и наша программа:

```
0000bfe6af4232f78b0c8eba37a6ba6c17b9b8671473b0b82305880be077edd9
```

В методе `mine()` листинга 8.3 мы жестко закодировали требуемое число нулей как `0000`. Чтобы повысить эффективность этого метода, мы добавляем к нему аргумент:

```
mine(difficulty: number): Promise<void>
```

Значение `difficulty` можно использовать для представления количества нулей в начале хеш-значения. Повышение `difficulty` существенно увеличивает время, необходимое для нахождения *nonce*.

В следующем разделе мы начнем применять наши TypeScript-навыки и создадим простое блокчейн-приложение.

8.2. ВАШ ПЕРВЫЙ БЛОКЧЕЙН

Прочтение и понимание предыдущего раздела является предпосылкой для понимания содержимого этого, в котором мы создадим два блокчейн-приложения: одно без доказательства проделанной работы и второе с ним.

Первое приложение создаст блокчейн и предоставит API для добавления в него блоков. Перед добавлением нового блока в эту цепь мы будем вычислять для него SHA-256-хеш (без решения алгоритма) и хранить ссылку на хеш предыдущего блока. Мы также добавим в блок индекс, временную метку и данные.

Второе приложение не будет принимать блоки с произвольными хешами, но будет требовать добычу для получения чисел *nonce*, производящих хеши, начинающиеся с `0000`.

Обе программы будут запускаться из командной строки, использующей среду выполнения node.js, и генерировать SHA-256-хеши при помощи криптомодуля (https://nodejs.org/api/crypto.html).

8.2.1. Структура проекта

Каждая глава в части 2 является отдельным проектом со своим собственным файлом(и) package.json (зависимости) и tsconfig.json (опции компилятора TS). Вы можете найти исходный код проекта этой главы здесь: https://github.com/yfain/getts.

На рис. 8.5 приведен скриншот IDE VS Code после того, как мы открыли проект для главы 8, выполнили npm install и скомпилировали код с помощью компилятора tsc. В этом проекте есть два приложения, и их исходный код расположен в файлах src/bc101.ts и src/bc101_proof_of_work.ts.

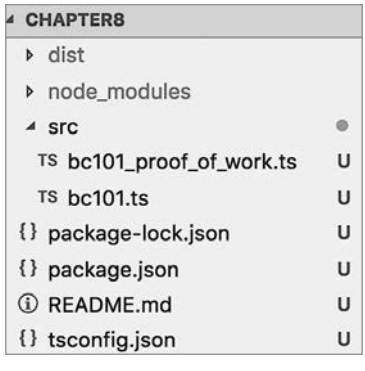


Рис. 8.5. Структура проекта блокчейн

При запуске компилятора tsc будет созданы директория dist и JS-код. Выполнение команды `npm install` из корневой директории проекта установит все зависимости проекта в директорию node_modules. Зависимости этого проекта перечислены в следующем файле package.json.

Листинг 8.4. Файл package.json проекта

```
{
    "name": "chapter8_blockchain",
    "version": "1.0.0",
    "license": "MIT",
    "scripts": {
        "tsc": "tsc"
```

Команда npm-сценария, запускающая локально установленный компилятор tsc

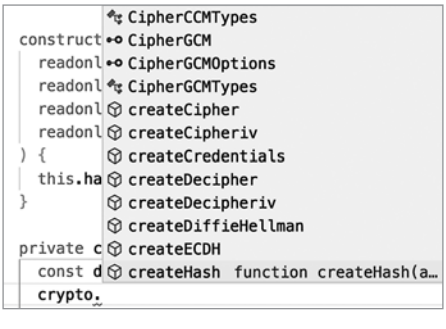
```
    },
    "devDependencies": {
        "@types/node": "^10.5.1",   <-- Файл определений типов для Node.js
        "typescript": "~3.0.0"      <-- Компилятор TypeScript
    }
}
```

ФАЙЛ ОПРЕДЕЛЕНИЙ ТИПОВ В ДЕЙСТВИИ

Мы представили файлы определений типов в главе 6. Эти файлы включают объявления типов для публичных API JavaScript. В нашем текущем проекте мы используем файл, описывающий API Node.js. Файлы определений типов имеют расширение *.d.ts.

Эти файлы позволяют модулю проверки типов TypeScript предупреждать вас, если вы пытаетесь неверно использовать API, например, когда функция ожидает численный аргумент, а вы вызываете ее со строковым. Такая проверка типов не была бы возможна, если бы вы просто использовали библиотеку JS, не имеющую аннотаций типов. Если же присутствуют файлы определений типов для библиотеки JS или модуля, то IDE может предложить контекстную помощь.

На следующем скриншоте показан редактор VS Code после того, как мы ввели слово `crypto.`, и IDE предложила контекстную помощь для API криптомодулей, которые идут с Node.js.



Автоподстановка для криптомодуля

Если бы мы не установили файл определений типов @types/node, то эту помощь бы не получили. Репозиторий npmjs.org содержит специальный раздел (или «ветку») @type, где хранятся тысячи файлов определений типов для популярных JS-библиотек.

Компилятор tsc является одной из зависимостей этого проекта, и мы определили пользовательскую команду npm для выполнения раздела scripts. npm-скрипты позволяют переопределять существующие npm-команды или определять собственные. Вы можете определить команду с любым именем и отдать npm-распоряжение о ее запуске, введя `npm run command-name`.

Согласно содержимому нашего проекта, вы можете запустить компилятор tsc так:

```
npm run tsc
```

А почему бы просто не выполнить команду tsc из командной строки? Это допустимо, если компилятор tsc установлен глобально на компьютере, с которого вы его запускаете. Это также может быть актуально, если вы выполняете команду на своем компьютере, где имеете полный контроль над глобально установленными инструментами.

Однако это не будет касаться случаев, когда в вашей фирме есть отдельная команда, ответственная за сборку и развертывание проектов на своих компьютерах. Они могут потребовать, чтобы вы предоставили код приложения в одном пакете с утилитами сборки. Когда вы запускаете любую программу, используя команду `npm run`, npm будет искать указанную программу в директории `node_modules/bin`.

В нашем проекте после выполнения `npm install` компилятор tsc будет установлен локально в node_modules/bin, и наш пакет будет включать необходимые для сборки приложения инструменты.

Наш проект также включает следующий файл конфигурации tsconfig.json с опциями компилятора TypeScript:

Листинг 8.5. tsconfig.json: конфигурация для компилятора tsc

```
{
  "compilerOptions": {
    "module": "commonjs",  <-- Как генерировать код для JavaScript-модулей
    "outDir": "./dist",  <-- Директория, где хранятся скомпилированные JS-файлы
    "target": "es2017",  <-- Компилирует в синтаксис ES2017
    "lib": [
      "es2017"  <-- Этот проект использует API, описанный в библиотеке es2017
    ]
  }
}
```

В то время как опции компилятора `outDir` и `target` самоописательны, `module` и `lib` в дополнительном описании все же нуждаются.

До версии ES6 JS-разработчики использовали разный синтаксис для разделения кода на модули. Например, формат AMD был популярен для браузерных приложений, а CommonJS использовался Node.js-разработчиками. В ES6 уже

были введены ключевые слова `import` и `export`, чтобы скрипт из одного модуля мог импортировать экспорт другого.

В TypeScript мы всегда используем ES6-модули, и если сценарию нужно загрузить из модуля некий код, мы используем ключевое слово `import`. Например, чтобы использовать код из криптомодуля Node.js, мы можем добавить в наш сценарий следующую строку:

```
import * as crypto from 'crypto';
```

Но среда выполнения Node.js реализует для модулей спецификацию CommonJS, которая требует, чтобы вы писали JS-код так:

```
const crypto = require("crypto");
```

Указывая в листинге 8.5 опцию компилятора `"module": "commonjs"`, мы даем команду tsc преобразовать инструкцию `import` в `require()`, и все члены модуля с квалификатором `export` будут добавлены в конструкцию `module.export(…)`, как предписано спецификацией CommonJS.

Что касается lib, то TS идет с набором библиотек, описывающих API, предоставляемые браузерами и спецификациями JS различных версий. Вы можете выборочно сделать эти библиотеки доступными для вашей программы посредством опции компилятора lib. Например, если вы хотите использовать в своей программе Promise (введенный в ES2015) и запускать ее в целевых браузерах, поддерживающих промисы, то можете использовать следующую опцию компилятора:

```
{
    "compilerOptions": {
        "lib": [ "es2015" ]
    }
}
```

Опция `lib` включает только определения типов; она не предоставляет фактическую реализацию API. По сути, вы говорите компилятору: «Не беспокойся, когда увидишь в этом коде `Promise`, — среда выполнения движка JS реализует этот API внутренне». Но вы должны либо запустить код в среде, поддерживающей `Promise` нативно, либо включить полизаполняющую библиотеку, которая предоставляет реализацию `Promise` для устаревших браузеров.

> **ПРИМЕЧАНИЕ** Список доступных библиотек можно найти по адресу https://github.com/Microsoft/TypeScript/tree/master/lib.

Теперь, когда мы прошлись по файлам конфигурации нашего проекта, давайте просмотрим код в двух TS-файлах, находящихся в директории src, как было изображено ранее на рис. 8.5.

8.2.2. Создание примитивного блокчейна

Сценарий chapter8/src/bc101.ts — это первая версия нашего блокчейна. Она содержит классы `Block` и `Blockchain` наряду с коротким скриптом, использующем API `Blockchain`'а для создания трех блоков. В этой главе мы не будем использовать браузеры — наши сценарии будут выполняться в среде Node.js. Давайте взглянем на код bc101.ts, начинающийся с класса `Block`.

Этот класс объявляет свойства, необходимые для каждого блока (вроде индекса и хеш-значений текущего и предыдущего блоков), а также метод для вычисления их хешей при помощи криптомодуля Node.js. В процессе инициализации объекта `Block` мы вычисляем его хеш на основе конкатенированных значений всех его свойств.

Листинг 8.6. Класс Block из bc101.ts

```
import * as crypto from 'crypto';

class Block {
    readonly hash: string;  <-- Хеш этого блока

    constructor (
        readonly index: number,  <-- Последовательный номер этого блока
        readonly previousHash: string,  <-- Хеш предыдущего блока
        readonly timestamp: number,  <-- Время создания блока
        readonly data: string  <-- Данные приложения
      ) {
        this.hash = this.calculateHash();  <-- Вычисляет хеш этого блока при его создании
      }

      private calculateHash(): string {
        const data = this.index + this.previousHash + this.timestamp +
                     this.data;
        return crypto                          <-- Создает экземпляр объекта Hash
                .createHash('sha256')              для генерации SHA-256-хешей
                .update(data)  <-- Вычисляет и обновляет хеш-значение внутри объекта Hash
                .digest('hex');  <-- Преобразует хеш-значение в шестнадцатеричную строку
      }
};
```

Конструктор класса Block вызывает метод `calculatehash()`, который начинает с конкатенации значений свойств блока: `index`, `previoushash`, `timestamp` и `data`. Эта конкатенированная строка передается модулю `crypto`, который вычисляет ее хеш в виде строки из шестнадцатеричных знаков. Этот хеш присваивается свойству `hash` только что созданного объекта `Block`, который, в свою очередь, будет передан объекту `Blockchain` для добавления в цепочку.

Данные транзакций блока хранятся в свойстве `data`, которое имеет тип `string` в классе `Block`. В реальном приложении свойство `data` имело бы пользовательский

тип, описывающий структуру данных, но в нашем примитивном блокчейне вполне сойдет и тип `string`.

Теперь давайте создадим класс `Blockchain`, использующий для хранения блоков массив и имеющий метод `addBlock()`, который выполняет три действия:

1. Создает экземпляр объекта `Block`.
2. Получает хеш-значение последнего добавленного блока и хранит его в свойстве `previousHash` нового блока.
3. Добавляет в массив новый блок.

Когда объект `Blockchain` будет инстанцирован, его конструктор создаст первичный блок, который не будет иметь ссылки на предыдущий.

Листинг 8.7. Класс Blockchain

```
class Blockchain {
    private readonly chain: Block[] = [];  <-- Наш блокчейн хранится здесь

    private get latestBlock(): Block {          <-- Геттер для получения ссылки
        return this.chain[this.chain.length - 1];   на последний добавленный блок
    }

    constructor() {
        this.chain.push(  <-- Создает первичный блок и добавляет его в цепочку
            new Block(0, '0', Date.now(),
            'Genesis block'));
    }

    addBlock(data: string): void {
        const block = new Block(  <-- Создает новый экземпляр Block и заполняет его свойства
            this.latestBlock.index + 1,
            this.latestBlock.hash,
            Date.now(),
            data
        );

        this.chain.push(block);  <-- Добавляет блок в массив
    }
}
```

Теперь можно вызвать метод `Blockchain.addBlock()` для добычи блоков. Следующий код создает экземпляр `Blockchain` и дважды вызывает `addBlock()`, добавляя два блока с данными: `First block` и `Second block` соответственно. Первичный блок создается в конструкторе `Blockchain`.

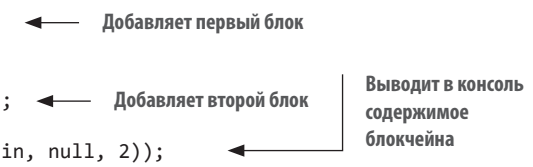
Листинг 8.8. Создание блокчейна из трех блоков

```
console.log('Creating the blockchain with the genesis block...');
const blockchain = new Blockchain();  <-- Создает новый блокчейн
```

```
console.log('Mining block #1...');
blockchain.addBlock('First block');

console.log('Mining block #2...');
blockchain.addBlock('Second block');

console.log(JSON.stringify(blockchain, null, 2));
```



Файл bc101.ts включает скрипты, показанные в листингах 8.6–8.8. Чтобы выполнить этот сценарий, скомпилируйте его и запустите JS-версию bc101.js в среде выполнения Node.js:

```
npm run tsc
node dist/bc101.js
```

Сценарий bc101.js выведет содержимое нашего блокчейна в консоль, как показано в следующем листинге. Массив `chain` хранит три блока нашего простейшего блокчейна.

Листинг 8.9. Вывод консоли, произведенный bc101.js

```
Creating the blockchain with the genesis block...
Mining block #1...
Mining block #2...
{
    "chain": [
      {
        "index": 0,
        "previousHash": "0",
        "timestamp": 1532207287077,
        "data": "Genesis block",
        "hash": "cc521dd5bbf1786977b14d16ce5d7f8da0e9f3353b3ebe0762ad9258c
       8ab1a 04"
      },
      {
        "index": 1,
        "previousHash": "cc521dd5bbf1786977b14d16ce5d7f8da0e9f3353b3ebe0762ad92
       58c8ab1a04",
        "timestamp": 1532207287077, "data": "First block",
        "hash": "52d40c33a8993632d51754c952fdb90d61b2c8bf13739433624bbf6b04
       933e 52"
      },
      {
        "index": 2,
        "previousHash": "52d40c33a8993632d51754c952fdb90d61b2c8bf13739433624bbf
       6b04933e52",
        "timestamp": 1532207287077, "data": "Second block",
        "hash": "0d6d43368772e2bee5da8a1cc92c0c7f28a098bfef3880b3cc8caa5f40
       c597 76"
      }
   ]
}
```

> **ПРИМЕЧАНИЕ** При выполнении сценария bc101.js вы не увидите те же хеш-значения, что приведены в листинге 8.9, так как мы используем для генерации хеша временную метку, которая у каждого читателя будет своей.

Обратите внимание, что `previousHash` каждого блока (за исключением первичного) имеет то же значение, что и свойство `hash` предыдущего блока цепи. Программа выполняется очень быстро, но реальный блокчейн потребовал бы майнеров данных, которые затратили бы необходимое число циклов CPU на решение алгоритма, чтобы сгенерированные хеши соответствовали требованиям, как это было описано в разделе 8.1.3.

Давайте сделаем процесс добычи блока немного реалистичнее, добавив в него решение задачи.

8.2.3. Создание блокчейна с доказательством проделанной работы

Следующая версия нашего блокчейна расположена в файле bc101_proof_of_work.ts. Этот сценарий имеет много общего с bc101.ts, но дополнительно имеет код, обеспечивающий добычу майнерами данных как доказательство работы, благодаря чему их блоки могут стать кандидатами на добавление в блокчейн.

Сценарий bc101_proof_of_work также имеет классы `Block` и `Blockchain`, но если последний и дублирует своего собрата из листинга 8.7, то класс `Block` содержит дополнительный код.

В частности, класс `Block` имеет свойство `nonce`, вычисленное в новом методе `mine()`. `Nonce` будет конкатенирован с другими свойствами блока для генерации хеша, начинающегося с пяти нулей.

Процесс вычисления *nonce*, отвечающий нашим требованиям, займет какое-то время. На этот раз мы хотим, чтобы хеши начинались с пяти нулей. Метод `mine()` будет вызывать `calculateHash()` с разными значениями *nonce* до тех пор, пока сгенерированный хеш не будет начинаться с `00000`. Новая версия класса `Block` показана в следующем листинге.

Листинг 8.10. Класс Block из bc101_proof_of_work.ts

```
class Block {
    readonly nonce: number;  ◄——  Новое свойство nonce
    readonly hash: string;

constructor (
    readonly index: number,
    readonly previousHash: string,
```

```
    readonly timestamp: number,
    readonly data: string
    ) {
        const { nonce, hash } = this.mine();  <-- Вычисляет nonce и хеш
        this.nonce = nonce;
        this.hash = hash;
    }

    private calculateHash(nonce: number): string {
        const data = this.index + this.previousHash + this.timestamp +
➥ this.data + nonce;  <-- Nonce является частью ввода для вычисления хеша
        return crypto.createHash('sha256').update(data).digest('hex');
    }

    private mine(): { nonce: number, hash: string } {
        let hash: string;
        let nonce = 0;

        do {                                          <-- Использует полный перебор
            hash = this.calculateHash(++nonce);           для добычи данных
        } while (hash.startsWith('00000') === false);  <-- Выполняет этот цикл, пока хеш
                                                           не будет начинаться с 00000
        return { nonce, hash };
    }
};
```

Обратите внимание, что метод `calculateHash()` почти идентичен методу из листинга 8.6. Единственное отличие в том, что мы присоединили значение перца строке ввода, используемой для вычисления хеша. Метод `mine()` продолжает вызывать `calculateHash()` в цикле, передавая в качестве аргумента *nonce* последовательные числа `0`, `1`, `2`, … . Рано или поздно вычисленный хеш будет начинаться с `00000`, и метод `mine()` вернет хеш вместе с вычисленным перцем.

Мы хотели бы обратить ваше внимание на строку, вызывающую метод `mine()`:

```
const { nonce, hash } = this.mine();
```

Фигурные скобки слева от знака равенства представляют JavaScript-деструктуризацию (см. приложение). Метод `mine()` возвращает объект с двумя свойствами, и мы извлекаем их значения в две переменные: `nonce` и `hash`.

Мы просмотрели только класс `Block` из сценария bc101_proof_of_work.ts, потому что оставшаяся его часть такая же, как в bc101.ts. Можно запустить эту программу так:

```
node dist/bc101_proof_of_work.js
```

Этот сценарий не завершит выполнение так же быстро, как bc101.ts. Для этого может потребоваться несколько секунд, которые он потратит на добычу блока. Следующий листинг демонстрирует вывод этого сценария.

Листинг 8.11. Вывод консоли, произведенный bc101_proof_of_work.ts

```
Creating the blockchain with the genesis block...
Mining block #1...
Mining block #2...
{
    "chain": [
      {
        "index": 0,
        "previousHash": "0",
        "timestamp": 1532454493124,
        "data": "Genesis block",
        "nonce": 2832,
        "hash": "000005921a5611d92cdc81f89d554743d7e33af2b35b4cb1a0a52
       cd4664445 ca"
      },
      {
        "index": 1,
        "previousHash": "000005921a5611d92cdc81f89d554743d7e33af2b35b4cb1a0a52c
       d4664445ca",
        "timestamp": 1532454493140,
        "data": "First block",
        "nonce": 462881,
        "hash": "000009da95386579eee5e944b15eab2539bc4ac223398ccef8d40ed83502d4
       31"
      },
      {
        "index": 2,
        "previousHash": "000009da95386579eee5e944b15eab2539bc4ac223398ccef8d40e
       d83502d431",
        "timestamp": 1532454494233,
        "data": "Second block",
        "nonce": 669687,
        "hash":
"0000017332a9321b546154f255c8295e4e805417e50b78609ff59a10bf9c237c"
      }
   ]
}
```

И снова массив `chain` хранит блокчейн из трех блоков, но на этот раз каждый из них имеет в свойстве `nonce` разное значение, а хеш теперь начинается с `00000`, что служит доказательством проделанной работы: мы выполнили добычу блока и решили для блока алгоритм.

Взгляните еще раз на код в листинге 8.10 и найдите знакомые элементы синтаксиса TS. Каждое из шести свойств класса имеет тип и помечено как `readonly`. Свойства `nonce` и `hash` в этом классе были объявлены явно, а четыре дополнительных свойства были созданы компилятором TS, так как с каждым аргументом конструктора мы использовали квалификатор `readonly`.

Оба метода класса явно объявляют типы своих аргументов и возвращают значения. Они оба были объявлены с уровнем доступа `private`, то есть могут быть вызваны только внутри класса.

Возвращаемый тип метода `mine()` объявлен как `{ nonce: number, hash: string }`. Так как этот тип используется всего единожды, мы не стали создавать для него пользовательский тип данных.

ИТОГИ

- Главный смысл блокчейна в том, что он предлагает децентрализованную обработку транзакций без опоры на единый орган управления.
- В блокчейне каждый блок идентифицируется по хеш-значению и привязывается к предыдущему блоку через хранение его хеш-значения.
- Прежде чем вставлять новый блок в блокчейн, каждому узлу, заинтересованному в вычислении приемлемого хеш-значения за награду, предлагается математическая задача. Когда мы говорим «узел», то имеем в виду компьютер, сеть или ферму, представляющую одного участника блокчейна.
- В криптографии число, которое может использоваться только один раз, называется *nonce*, и майнер данных для «добычи» nonce должен затратить вычислительные ресурсы — так он получит доказательство проделанной работы, которое необходимо для любого блока, чтобы быть рассмотренным на предмет добавления в блокчейн.
- В нашем примере блокчейн-приложения приемлемое хеш-значение должно было начинаться с пяти нулей, и мы вычислили nonce, чтобы гарантировать, что хеш блока действительно начинается с пяти нулей. Вычисление *nonce* требует времени, что задерживает добавление нового блока в блокчейн, но может быть использовано как доказательство работы для награждения узла блокчейна, вычислившего его раньше остальных.

9 Разработка узла блокчейна на основе браузера

В этой главе:

- ✓ Создание веб-клиента для блокчейна.
- ✓ Создание небольшой библиотеки для генерации хеша.
- ✓ Выполнение блокчейн-приложения и отладка TypeScript в браузере.

В главе 8 мы разработали приложение, которое создает блокчейн, и предоставили сценарий для добавления в него блоков. Мы запускали это приложение из командной строки, и оно выполнялось в среде Node.js.

В этой главе мы модифицируем блокчейн-приложение, чтобы оно выполнялось в браузере. Мы не станем использовать здесь фреймворк, поэтому у него будет достаточно простой UI, вместо этого задействуем стандартные API-методы браузера вроде `document.getElementById()` и `addEventListener()`.

В этом приложении каждый блок будет хранить данные о нескольких транзакциях. Они будут не простыми строками, как в главе 8, а пользовательскими типами TypeScript. Мы накопим несколько транзакций, а затем создадим блок для внедрения в блокчейн. Помимо этого, мы создали небольшую библиотеку, содержащую код для добычи блоков и генерации верного хеша. Эта библиотека может использоваться в браузере, а также в среде Node.js.

Приложение этой главы включает практические примеры следующих элементов TS (и JS) синтаксиса, рассмотренных в части 1 этой книги:

- Использование ключевых слов `private` и `readonly`.
- Использование интерфейсов и классов TS для объявления пользовательских типов.
- Клонирование объектов с помощью JS-оператора распространения.
- Использование ключевого слова `enum`.
- Использование `async`, `await` и `Promise`.

Мы начнем с рассмотрения структуры проекта и запуска нашего блокчейн-веб-приложения.

9.1. ЗАПУСК БЛОКЧЕЙН-ВЕБ-ПРИЛОЖЕНИЯ

Начнем с того, что покажем вам конфигурирование этого проекта. Затем вы узнаете команды для его компиляции и развертывания, а в завершение увидите, как пользователь может работать с этим приложением в браузере.

9.1.1. Структура проекта

Исходный код этого проекта можно найти здесь: https://github.com/yfain/getts. На рис. 9.1 показана структура проекта.

Исходники (TypeScript, HTML и CSS) расположены в субдиректориях с названиями browser, lib и node. Основной механизм создания блокчейна реализован в директории lib, но мы предоставили два демоприложения — одно для веб-браузера и второе для среды Node.js. Вот что содержат поддиректории src:

- lib — реализует создание блокчейна и добычу блока. Она также содержит универсальную функцию, генерирующую хеши для среды браузера и Node.js.
- browser — содержит код, реализующий UI-блокчейн веб-приложения. Этот код использует код из директории lib.
- node — содержит небольшое приложение, которое вы можете запустить независимо. Оно тоже использует код из директории lib.

Этот проект будет иметь в файле package.json некоторые зависимости, которые вы еще не встречали (листинг 9.1). Поскольку это веб-приложение, понадобится веб-сервер, и в этой главе мы будем использовать пакет под названием serve, который доступен на npmjs.org. С точки зрения развертывания это приложение будет иметь не только файлы JS, но также CSS- и HTML-файлы, скопированные в директорию развертывания dist. Эту работу будет выполнять пакет copyfiles.

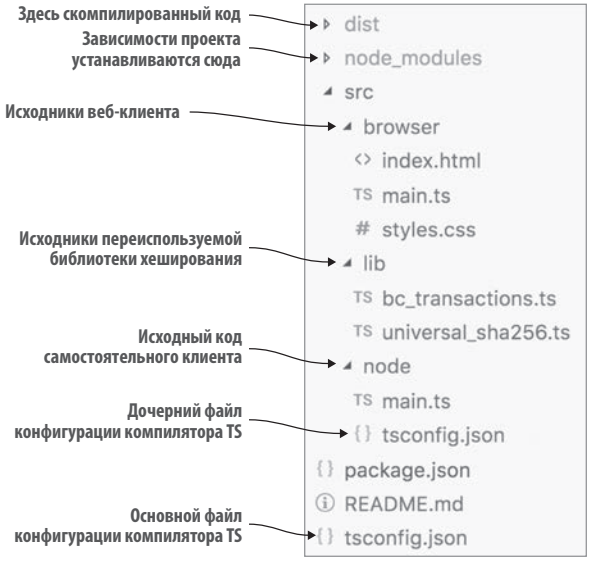


Рис. 9.1. Структура проекта для блокчейн-приложения

Чтобы не вызывать сценарий copyfiles, добавим пару команд в раздел npm-сценариев файла package.json. Мы начали использовать npm-сценарии в разделе 8.2.1, но раздел scripts файла package.json этой главы будет содержать больше команд.

После того как вы выполните команду `npm install`, в директорию node_modules будут установлены все зависимости проекта, а исполняемые файлы copyfiles и serve будут установлены в node_modules/.bin. Исходники этого проекта находятся в директории src, а развертываемый код будет сохранен в dist.

Листинг 9.1. Файл package.json

```
{
    "name": "TypeScript_Quickly_chapter9",
    "version": "1.0.0",
    "license": "MIT",
    "scripts": {
        "start": "serve",
        "compileDeploy": "tsc && npm run deploy",
        "deploy": "copyfiles -f src/browser/*.html src/browser/*.css dist"
    },
    "devDependencies": {
        "@types/node": "^10.5.1",
        "serve": "^10.0.1",
        "copyfiles": "^2.1.0",
        "typescript": "~3.0.0"
    }
}
```

npm-команда start будет запускать веб-сервер

Комбинирует две команды: tsc и deploy

Команда deploy скопирует HTML- и CSS-файлы

Пакет serve является новой dev-зависимостью

Пакет copyfiles является новой dev-зависимостью

Обратите внимание, что этот проект имеет два файла tsconfig.json, используемые компилятором TypeScript. Основной tsconfig.json расположен в корневой директории проекта и определяет опции компилятора для всего проекта, например целевую версию JS и используемые библиотеки.

Листинг 9.2. Основной файл tsconfig.json

```
{
    "compilerOptions": {
    "sourceMap": true,  <-- Генерирует файлы карт кода
    "outDir": "./dist",  <-- Скомпилированные JavaScript-файлы помещаются в директорию dist
    "target": "es5",
    "module": "es6",  <-- Использует синтаксис модулей ES6
    "lib": [
        "dom",  <-- Использует определения типов для API DOM браузера
        "es2018"  <-- Использует определения типов, поддерживаемые ES2018
    ]
  }
}
```

Мы хотим, чтобы для кода, выполняемого в браузере, компилятор TypeScript генерировал JS, использующий модули (как рассказано в разделе A.11 приложения).

Помимо этого, мы хотим генерировать файлы карт кода, которые отображают строки TS-кода в соответствующие строки сгенерированного JS. При наличии карт кода вы можете отлаживать TS-код во время выполнения веб-приложения в браузере, даже несмотря на то что браузер выполняет JS-версию кода. В разделе 9.5 мы покажем вам, как это делается. Если браузерная панель инструментов разработчика открыта, она загрузит файл карт кода вместе с файлом JS, тем самым позволив произвести в ней отладку TS-кода.

Другой файл tsconfig.json расположен в директории src/node. Этот файл наследует все свойства из файла tsconfig.json, находящегося в корне проекта, согласно опции `extends`. Первым tsc загрузит файл tsconfig.json, а затем уже его наследованный вариант, переопределяя или добавляя свойства.

Листинг 9.3. Дочерний файл конфигурации
src/node/tsconfig.json child config file

```
{
    "extends": "../../tsconfig.json",  <-- Наследует свойства из этого файла
    "compilerOptions": {
        "module": "commonjs"
    }
}
```

В базовом файле tsconfig.json для свойства `module` установлено значение `es6`, что вполне подходит для генерации JS, выполняемого в браузере. Дочерний файл

конфигурации в директории node переопределяет свойство module на значение commonjs, следовательно, компилятор TS будет генерировать связанный с модулем код согласно правилам CommonJS.

> **ПРИМЕЧАНИЕ** Описание всех возможных опций tsconfig.json можно найти в документации TypeScript: www.typescriptlang.org/docs/handbook/tsconfig-json.html.

9.1.2. Развертывание приложения с помощью npm-сценариев

Чтобы выполнить развертывание нашего веб-приложения, нужно скомпилировать TS-файлы в директорию dist, а также скопировать в нее index.html и styles.css.

Раздел scripts файла package.json (см. листинг 9.1) содержит три команды: start, compileDeploy и deploy.

```
"scripts": {
    "start": "serve",
    "compileDeploy": "tsc && npm run deploy",
    "deploy": "copyfiles -f src/browser/*.html src/browser/*.css dist"
},
```

Запускает веб-сервер

Выполняет команды tsc и deploy

Копирует HTML- и CSS-файлы из директории src/browser в dist

Команда deploy просто копирует HTML- и CSS-файлы из директории src/browser в директорию dist.

Команда compileDeploy выполняет две команды: tsc и deploy. В npm-сценариях для указания последовательности выполнения команд используется двойной амперсанд (&&), следовательно, чтобы скомпилировать TS-файлы и скопировать файлы index.html и styles.css в директорию dist, нужно выполнить следующую команду:

```
npm run compileDeploy
```

После ее выполнения директория dist будет содержать файлы, показанные на рис. 9.2.

> **ПРИМЕЧАНИЕ** Исходный код реального приложения содержит сотни файлов, и прежде чем развертывать их на веб-сервере, следует применить ряд инструментов для их оптимизации и связывания в меньшее количество. Один из наиболее популярных бандлеров — это Webpack, о котором мы говорили в разделе 6.3.

Теперь можно запустить веб-сервер при помощи следующей команды:

```
npm start
```

Эта команда запустит его на localhost через порт 5000. Консоль при этом покажет вывод, представленный на рис. 9.3.

Рис. 9.2. Файлы для развертывания нашего веб-приложения

```
Serving!

- Local:            http://localhost:5000
- On Your Network:  http://10.0.0.6:5000

Copied local address to clipboard!
```

Рис. 9.3. Запуск веб-сервера

СОВЕТ npm поддерживает ограниченное число скриптов, и `start` входит в их число (см. документацию к `npm` здесь: https://docs.npmjs.com/misc/scripts). С этими скриптами вам нет нужды использовать опцию `run`, поэтому мы не применяли команду `run`, как в `npm run start`. Тем не менее пользовательские скрипты вроде `compileDeploy` команду `run` требуют.

ПРИМЕЧАНИЕ В главе 10 мы создадим приложение, имеющее раздельный код для клиента и сервера. Мы будем запускать сервер, используя npm-пакет nodemon.

9.1.3. Работа с блокчейн-веб-приложением

Откройте адрес http://localhost:5000/dist в браузере, и веб-сервер отправит в него dist/index.html. Файл index.html загрузит скомпилированные JS-файлы, и спустя несколько секунд, затраченных на добычу блока, UI блокчейн-веб-приложения будет выглядеть, как показано на рис. 9.4.

Посадочная страница этого приложения показывает начальное состояние нашего блокчейна с одним только первичным блоком. Пользователь может добавлять транзакции и щелкать по кнопке Transfer Money (отправить деньги) после каждой, что будет добавлять транзакции в поле Pending Transactions (Ожидающие транзакции). Кнопка Confirm Transactions (Подтвердить транзакции) включена, и окно браузера будет выглядеть как на рис. 9.5.

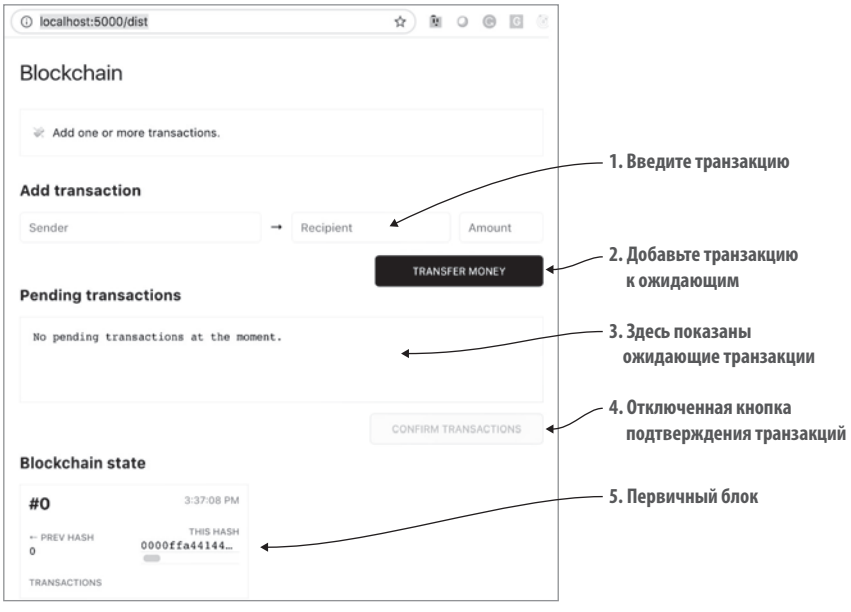


Рис. 9.4. Запуск веб-сервера

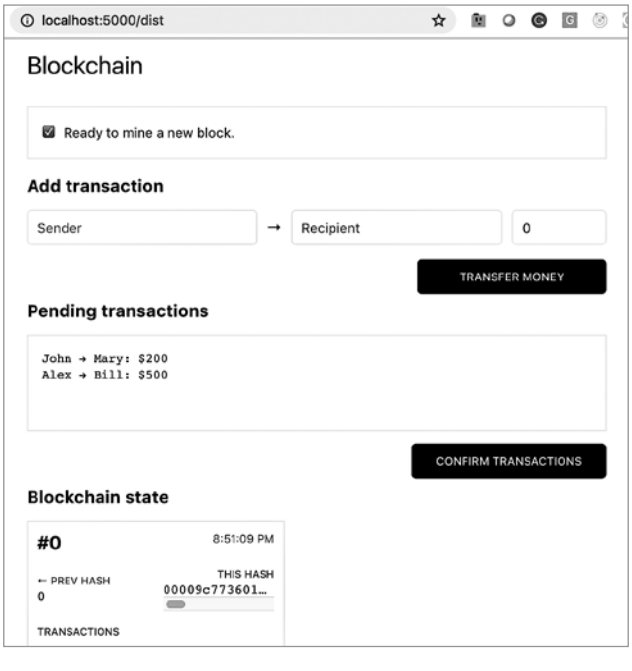


Рис. 9.5. Создание ожидающих транзакций

Как можно видеть, мы добавили ожидающие транзакции, еще не создав и не представив наш блок на рассмотрение в блокчейн. Именно для этого служит кнопка Confirm Transactions. На рис. 9.6 показано окно браузера после того, как мы щелкнули по этой кнопке и приложение затратило некоторое время на добычу данных.

ПРИМЕЧАНИЕ Для простоты мы предположили, что если Джон платит Мэри определенную сумму денег, то у него они есть. В реальных же приложениях сперва проверяется баланс аккаунта и только затем создается ожидающая транзакция.

Как и в главе 8, новый блок, #1, был создан с хеш-значением, начинающимся с четырех нулей. Но теперь блок включает *две* транзакции: одну между Джоном и Мэри и вторую между Алексом и Биллом. Блок также был добавлен в блокчейн. Как вы видите, ожидающих транзакций не осталось.

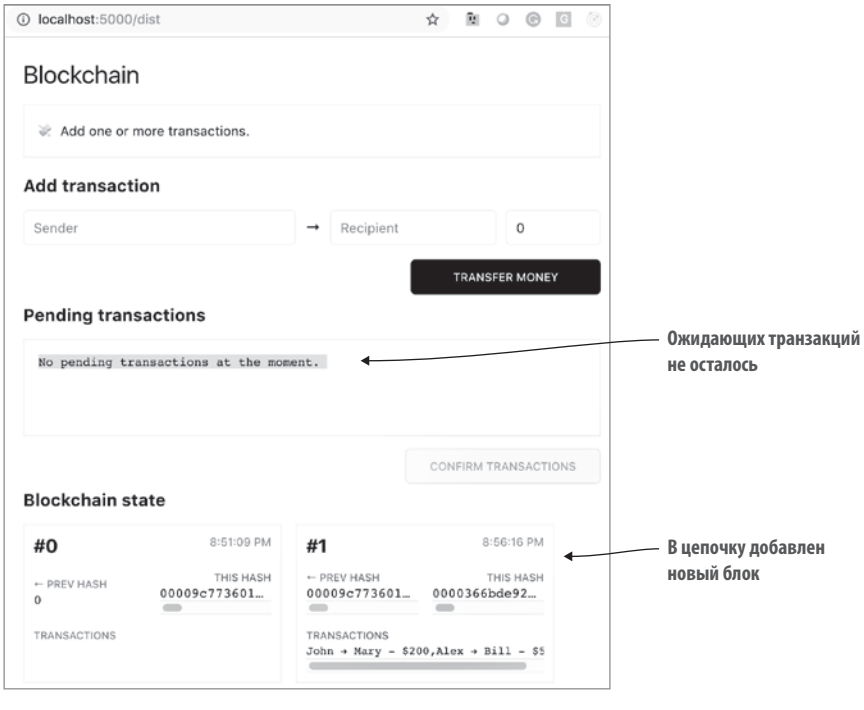


Рис. 9.6. Добавление нового блока в блокчейн

Вам может стать интересно, почему кажущиеся несвязанными транзакции помещаются в один блок. Дело в том, что добавление блока в блокчейн — это медленная операция, и создание нового блока для каждой транзакции замедлило бы этот процесс еще сильнее.

Чтобы добавить в блокчейн какой-либо контекст, давайте представим, что этот блокчейн был создан для крупного агентства недвижимости. Джон покупает квартиру у Мэри и должен предоставить доказательства ее оплаты. Аналогичным образом Алекс платит Биллу за дом, осуществляя другую транзакцию. Пока эти транзакции находятся в состоянии ожидания, как доказательство платежа они не рассматриваются. Но как только они добавятся в блок, а блок — в блокчейн, сделка будет считаться свершенной.

Теперь, когда вы видели приложение в действии, давайте рассмотрим его код, начиная с UI.

9.2. ВЕБ-КЛИЕНТ

Директория browser содержит три файла: index.html, main.ts и styles.css. Мы рассмотрим код первых двух, начав с index.html.

Листинг 9.4. browser/index.html: сценарий, загружающий веб-приложение

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Blockchain</title>
    <link rel="stylesheet" href="dist/styles.css">  <-- Включает CSS
    <script type="module" src="dist/browser/main.js"></script>  <-- Включает главный модуль нашего приложения
</head>
<body>
    <main>
        <h1>Blockchain</h1>
        <aside>
            <p id="status">⧗ Initializing the blockchain, creating the genesis
➥ block...</p>
    </aside>

    <section>  <-- Раздел для добавления транзакций
        <h2>Add transaction</h2>
        <form>
            <input id="sender" type="text" autocomplete="off" disabled
➥ placeholder="Sender">
            <span>•</span>
            <input id="recipient" type="text" autocomplete="off" disabled
➥ placeholder="Recipient">
            <input id="amount" type="number" autocomplete="off" disabled
➥ placeholder="Amount">
            <button id="transfer" type="button" class="ripple" disabled>
                TRANSFER MONEY</button>  <-- Добавляет транзакции в список ожидания
        </form>
    </section>
```

```
    <section>  ◄──  Раздел для отображения ожидающих транзакций
        <h2>Pending transactions</h2>
        <pre id="pending-
      transactions">No pending transactions at the moment.</pre>
        <button id="confirm" type="button" class="ripple" disabled>
                CONFIRM TRANSACTIONS</button>  ◄──  Начинает добычу нового блока
        <div class="clear"></div>                  с ожидающими транзакциями
    </section>

    <section>  ◄──  Здесь отображается содержимое блокчейна
        <h2>Blockchain state</h2>
            <div class="wrapper">
                <div id="blocks"></div>
                <div id="overlay"></div>
            </div>
        </section>
    </main>
</body>
</html>
```

Раздел <head> файла index.html включает теги для загрузки styles.css и main.js. Последний представляет собой скомпилированную версию main.ts. Фактически наш проект содержит и другие файлы TS, но поскольку мы спроектировали наше приложение на основе модулей, сценарий main.ts начинается с импорта членов JS-модуля lib/bc_transactions.ts, как показано в следующем листинге:

Листинг 9.5. Первая часть browser/main.ts

```
import { Blockchain, Block } from '../lib/bc_transactions.js';  ◄──  Импортирует классы
                                                                    Block и Blockchain
enum Status {  ◄──  Объявляет возможные статусы приложения
  Initialization = '⧖ Initializing the blockchain, creating the genesis
➥ block...',
  AddTransaction = '✉ Add one or more transactions.',
  ReadyToMine    = '✓ Ready to mine a new block.',
  MineInProgress = '⧖ Mining a new block...'
}

// Получает HTML-элементы  ◄──  Получает ссылки на все важные HTML-элементы
const amountEl              = document.getElementById('amount')
➥ as HTMLInputElement;
const blocksEl              = document.getElementById('blocks') as
➥ HTMLDivElement;
const confirmBtn            = document.getElementById('confirm')
➥ as HTMLButtonElement;
const pendingTransactionsEl =
➥ document.getElementById('pending-transactions') as HTMLPreElement;
const recipientEl           = document.getElementById('recipient')
➥ as HTMLInputElement;
const senderEl              = document.getElementById('sender')
➥ as HTMLInputElement;
```

```
const statusEl              = document.getElementById('status')
➥ as HTMLParagraphElement;
const transferBtn           = document.getElementById('transfer')
➥ as HTMLButtonElement;
```

> **ПРИМЕЧАНИЕ** Обратите внимание, что в теге `<script>`, загружающем main.ts, мы используем атрибут `type="module".` Подробнее о типе `module` см. в разделе A.11 приложения.

В главе 4 мы рассказывали о перечислениях, называемых константами. В листинге 9.5 мы используем их для объявления конечного числа статусов приложения: `initialization`, `AddTransaction`, `ReadyToMine` и `MineInProgress`. Константа `statusEl` представляет HTML-элемент, в котором мы хотим отображать текущий статус приложения.

> **ПРИМЕЧАНИЕ** Иконки, которые вы видите в строках `enum`, являются символами эмодзи. Вы можете вставлять их в строки в macOS через нажатие Cmd-Ctrl-пробел, а в Windows 10 — через нажатие Win-. (Win и точка) или Win-; (Win и точка с запятой).

Остальная часть листинга 9.5 показывает код, получающий ссылки на разные HTML-элементы, которые либо хранят значения, введенные пользователем, либо отображают блоки нашего блокчейна.

Каждый из этих HTML-элементов имеет уникальный `id` (см. листинг 9.4), и для получения этих объектов DOM мы используем метод `getElementById()` API браузера.

Листинг 9.6 показывает выражение немедленно вызываемой функции (IIFE) `main()`, и мы используем здесь ключевые слова `async/await` (см. раздел A.10.4 приложения). Эта функция создает новый блокчейн с начальным первичным блоком и присваивает кнопкам слушателей событий, которые добавляют ожидающие транзакции и инициируют добычу нового блока.

Листинг 9.6. Вторая часть browser/main.ts

```
(async function main(): Promise<void> {

    transferBtn.addEventListener('click', addTransaction);
    confirmBtn.addEventListener('click', mineBlock);

    statusEl.textContent = Status.Initialization;

    const blockchain = new Blockchain();
    await blockchain.createGenesisBlock();
    blocksEl.innerHTML = blockchain.chain.map((b, i) =>
➥ generateBlockHtml(b, i)).join('');
```

Добавляет кнопкам слушателей событий

Показывает начальный статус при помощи enum

Создает экземпляр Blockchain

Создает первичный блок

Генерирует HTML для отображения блоков

```
    statusEl.textContent = Status.AddTransaction;
    toggleState(true, false);

    function addTransaction() {
        blockchain.createTransaction({
            sender: senderEl.value,
            recipient: recipientEl.value,
            amount: parseInt(amountEl.value),
        });

        toggleState(false, false);
        pendingTransactionsEl.textContent =
➥ blockchain.pendingTransactions.map(t =>
            `${t.sender} • ${t.recipient}: $${t.amount}`).join('\n');
          statusEl.textContent = Status.ReadyToMine;

        senderEl.value = '';
        recipientEl.value = '';
        amountEl.value = '0';
    }

    async function mineBlock() {
        statusEl.textContent = Status.MineInProgress;
        toggleState(true, true);
        await blockchain.minePendingTransactions();

        pendingTransactionsEl.textContent = 'No pending transactions at
➥ the moment.';
        statusEl.textContent = Status.AddTransaction;
        blocksEl.innerHTML = blockchain.chain.map((b, i) =>
➥ generateBlockHtml(b, i)).join('');
        toggleState(true, false);
    }
})();
```

Добавляет ожидающую транзакцию

Отображает ожидающие транзакции как строки

Сбрасывает значения формы

Добывает блок и отображает его на веб-странице

Создает новый блок, вычисляет хеш и добавляет его в блокчейн

Отображает только что добавленный блок на веб-странице

В листинге 9.6 мы используем класс `Blockchain` (рассмотренный в разделе 9.3), который содержит массив `chain`, хранящий содержимое нашего блокчейна. Класс `Blockchain` имеет метод `minePendingTansactions()`, который добавляет транзакции в новый блок.

Этот процесс добычи начинается, когда пользователь кликает по кнопке Confirm Transactions. В паре мест мы вызываем `blockchain.chain.map()`, чтобы преобразовать блоки в текст или HTML-элементы для отображения на веб-странице. Весь этот рабочий процесс изображен на рис. 9.7.

При любом вызове асинхронной функции мы изменяем содержимое HTML-элемента `statusEl`, чтобы держать пользователя в курсе текущего статуса веб-приложения.

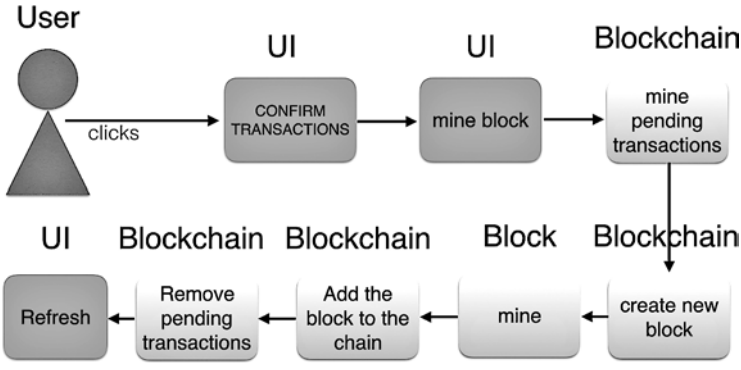


Рис. 9.7. Рабочий процесс user — blockchain

В листинге 9.7 показана оставшаяся часть main.ts, содержащая две функции: `toggleState()` и `generateBlockHtml()`.

Листинг 9.7. Третья часть browser/main.ts

```
function toggleState(confirmation: boolean, transferForm: boolean): void {
    transferBtn.disabled = amountEl.disabled = senderEl.disabled =
➥ recipientEl.disabled = transferForm;
    confirmBtn.disabled = confirmation;
}

function generateBlockHtml(block: Block, index: number) {
    return `
        <div class="block">
            <span class="block index">#${index}</span>
            <span class="block timestamp">${new
➥ Date(block.timestamp).toLocaleTimeString()}</span>
            <div class="prev-hash">
                <div class="hash-title">• PREV HASH</div>
                <div class="hash-value">${block.previousHash}</div>
            </div>
            <div class="this-hash">
                <div class="hash-title">THIS HASH</div>
                <div class="hash-value">${block.hash}</div>
            </div>
            <div class="block transactions">
            <div class="hash-title">TRANSACTIONS</div>
            <pre class="transactions- value">${block.transactions.map(t =>
                    `${t.sender} •
➥ ${t.recipient} - $${t.amount}`)}</pre>
        </div>
      </div>
   `;
}
```

Отключает/включает форму и кнопку подтверждения

Генерирует HTML блока

Функция `toggleState()` имеет два параметра `boolean`. В зависимости от значения первого параметра, мы либо включаем, либо отключаем форму, которая состоит из полей ввода `Sender` (отправитель), `Recipient` (получатель), `Amount` (количество), а также кнопки Transfer Money (отправить деньги). Второй параметр включает или отключает кнопку Confirm Transactions.

Функция `generateBlockHtml()` возвращает контейнер `<div>` с информацией по каждому блоку, как изображено на рис. 9.6. Мы используем несколько селекторов классов CSS, которые определяются в файле browser/styles.css (не показанном здесь).

Подытоживая, можно сказать, что код в сценарии browser/main.ts отвечает за взаимодействие с пользователем и делает следующее:

1. Получает доступ ко всем HTML-элементам.
2. Устанавливает слушателей для кнопок, которые добавляют ожидающие транзакции и создают новый блок.
3. Создает экземпляр блокчейна с первичным блоком.
4. Определяет метод, создающий ожидающие транзакции.
5. Определяет метод, добывающий новый блок с ожидающими транзакциями.
6. Определяет метод, генерирующий HTML для отображения блоков блокчейна.

Теперь, когда вы видели, как реализуется UI браузера, давайте рассмотрим код, создающий блокчейн, добавляющий блоки и обрабатывающий хеши.

9.3. ДОБЫЧА БЛОКОВ

Наш проект содержит директорию lib, являющуюся небольшой библиотекой, поддерживающей создание блоков с множественными транзакциями. Эта библиотека также проверяет, где выполняется наше блокчейн-приложение: в браузере или в самостоятельном движке JS, благодаря чему для генерации SHA-256-хешей используется правильный API.

Состоит описываемая библиотека из двух файлов:

- bc_transactions.ts — реализует создание блоков с транзакциями.
- universal_sha256.ts — проверяет среду и экспортирует соответствующую функцию `sha256()`, которая для генерации хешей будет использовать API браузера или Node.

В главе 8 (листинг 8.6) мы использовали Node crypto API, чтобы синхронно вызывать три функции:

```
crypto.createHash('sha256').update(data).digest('hex');
```

Теперь мы бы хотели генерировать хеши как в Node.js, так и в браузерах. Но в отличие от пакета, который мы используем с Node.js, в браузере crypto API асинхронен, поэтому код генерации хеша должен быть обернут в асинхронную функцию. К примеру, функция `main()`, вызывающая crypto API, возвращает `Promise` и помечена как `async`:

```
async mine(): Promise<void> {...}
```

Как правило, вместо простого возвращения из функции значения мы оборачиваем ее в `Promise`, который сделает функцию `mine()` асинхронной.

Мы рассмотрим код bc_transactions.ts в двух частях. Листинг 9.8 показывает код, импортирующий функцию генерации SHA-256, а также объявляющий интерфейс `Transaction` и класс `Block`. Блок может содержать более одной транзакции, а интерфейс `Transaction` объявляет структуру одной транзакции.

Листинг 9.8. Первая часть lib/bc_transactions.ts

```
import {sha256} from './universal_sha256.js';

export interface Transaction {
    readonly sender: string;
    readonly recipient: string;
    readonly amount: number;
}

export class Block {
    nonce: number = 0;
    hash: string;

    constructor (
        readonly previousHash: string,
        readonly timestamp: number,
        readonly transactions: Transaction[]
    ) {}

    async mine(): Promise<void> {
        do {
          this.hash = await this.calculateHash(++this.nonce);
        } while (this.hash.startsWith('0000') === false);
    }

    private async calculateHash(nonce: number): Promise<string> {
        const data = this.previousHash + this.timestamp +
➥ JSON.stringify(this.transactions) + nonce;
        return sha256(data);
    }
}
```

Импортирует функцию для генерации хеша

Пользовательский тип, представляющий одну транзакцию

Пользовательский тип, представляющий один блок

Передает в только что созданный блок массив транзакций

Асинхронная функция для добычи блока

Использует полный перебор для нахождения правильного nonce

Асинхронная функция обертывания для генерации хеша

Вызывает функцию, использующую crypto API, и генерирует хеш

UI нашего блокчейна позволяет пользователю создавать несколько транзакций, используя кнопку Transfer Money (см. рис. 9.5), и только нажатие кнопки Confirm Transactions создает экземпляр `Block`, передавая массив Transaction в его конструктор.

В главе 2 мы показали, как в TS объявлять пользовательские типы при помощи классов и интерфейсов. В листинге 9.8 вы можете видеть оба варианта: тип `Transaction` объявлен как интерфейс, но тип `Block` уже объявлен как класс. Мы не смогли сделать `Block` интерфейсом, так как хотели, чтобы он имел конструктор, давая нам возможность использовать оператор `new`, как это можно увидеть в листинге 9.9. Тип `Transaction` является TS-интерфейсом, который не позволит нам в процессе разработки допустить ошибки типов, но строки, объявляющие `Transaction`, не будут перенесены в скомпилированный JS.

Наш тип `Transaction` весьма прост. В реальном блокчейне мы могли ввести в интерфейс `Transaction` больше свойств. Например, покупка недвижимости является многоэтапным процессом, требующим нескольких платежей с промежутками во времени (начальный залог, платеж за уточнение правового статуса, оплата страховки и т. д.). В реальном блокчейне мы могли бы добавить `propertyID` для определения собственности (дом, земля или квартира) и тип `type` для указания типа транзакции.

Каждая транзакция будет включать свойства, описывающие область деятельности конкретного блокчейна, — у вас всегда будет тип транзакции. Блок тоже будет иметь свойства, требуемые реализацией блокчейна, вроде даты создания, хеш-значений текущего и предыдущего блоков. Помимо этого, тип блока может включать служебные методы, такие как `mine()` и `calculateHash()` в листинге 9.8.

> **ПРИМЕЧАНИЕ** У нас уже был класс `Block` в главе 8 (см. листинг 8.6), который хранил данные в виде `string` в свойстве `data`. На этот раз данные хранятся более структурированным способом в свойстве типа `Transaction()`. Вы также могли заметить, что три свойства класса `Block` были объявлены неявно через аргументы конструктора.

Наш класс `Block` содержит два метода: `mine()` и `calculateHash()`. `Mine()` продолжает увеличивать значение перца и вызывать `calculateHash()` до тех пор, пока хеш-значение не будет содержать в начале четыре нуля.

Согласно своей сигнатуре, функция `mine()` возвращает `Promise`, но где же ее инструкция `return`? На самом деле нам не нужно, чтобы эта функция что-либо возвращала — в момент генерации правильного хеша цикл должен просто закончиться. Но любая функция, отмеченная ключевым словом `async`, должна возвращать `Promise`, являющийся обобщенным типом (рассмотренным в главе 4), и используется с параметром типа. Используя `Promise<void>`, мы указываем, что эта функция возвращает `Promise` с пустым значением, поэтому инструкция `return` не требуется.

Листинг 9.9. Вторая часть lib/bc_transactions.ts

```
export class Blockchain {
    private readonly _chain: Block[] = [];
    private _pendingTransactions: Transaction[] = [];

    private get latestBlock(): Block {
        return this._chain[this._chain.length - 1];
    }

    get chain(): Block[] {

        return [ ...this._chain ];
    }

    get pendingTransactions(): Transaction[] {
        return [ ...this._pendingTransactions ];
    }

    async createGenesisBlock(): Promise<void> {
        const genesisBlock = new Block('0', Date.now(), []);

        await genesisBlock.mine();
        this._chain.push(genesisBlock);
    }

    createTransaction(transaction: Transaction): void {
        this._pendingTransactions.push(transaction);
    }

    async minePendingTransactions(): Promise<void> {
        const block = new Block(this.latestBlock.hash,
            Date.now(), this._pendingTransactions);
        await block.mine();
        this._chain.push(block);
        this._pendingTransactions = [];
    }
}
```

Геттер для последнего блока блокчейна

Геттер для всех блоков блокчейна

Геттер для всех ожидающих транзакций

Создает первичный блок

Создает хеш для первичного блока

Добавляет первичный блок в цепочку

Добавляет ожидающую транзакцию

Создает блок с ожидающими транзакциями и добавляет его в блокчейн

Создает хеш для нового блока

Добавляет новый блок в блокчейн

Поскольку метод `calculateHash()` не должен использоваться сценариями, находящимися вне `Block`, мы объявили его как `private`. Эта функция вызывает `sha256()`, которая получает строку и генерирует хеш. Обратите внимание, что для преобразования массива типа Transaction в `string` мы используем `JSON.stringify()`. Функция `sha256()` реализована в сценарии universal_sha256.ts, и мы рассмотрим ее чуть позже в разделе 9.4.

ПРИМЕЧАНИЕ Для простоты наша функция `calculateHash()` конкатенирует несколько транзакций в строку и затем вычисляет хеш. В реальных блокчейнах для вычисления хешей нескольких транзакций используется более эффективный алгоритм, называемый *Деревом Меркла* (https://ru.wikipedia.org/wiki/Дерево_хешей). Используя этот алгоритм, программа может создать дерево хешей (по одному для двух транзакций), и если кому-то потребуется вмешаться в одну транзакцию, то уже не придется проходить по всем транзакциям, чтобы пересчитать и проверить итоговый хеш.

Первая строка в листинге 9.8 импортирует файл ./universal_sha256.js, даже несмотря на то, что директория lib имеет только его TS-версию. TypeScript не позволяет нам использовать расширения .ts для ссылки на импортируемые имена файлов. Это гарантирует, что ссылки на внешние сценарии не изменятся после компиляции, когда все файлы будут иметь расширения .js. Эта инструкция импорта выглядит так, как будто мы импортируем одну функцию — `sha256()`, но изнутри она будет импортировать разные функции, использующие различные crypto API в зависимости от среды выполнения приложения. Мы покажем вам, как это делается, в разделе 9.4, когда будем рассматривать код universal_sha256.ts.

Во второй части файла bc_transactions.ts мы объявляем класс Blockchain.

Добавление в блокчейн нового блока намеренно требует время, чтобы предотвратить атаки двойного расходования (см. врезку «Атаки двойного расходования»). Например, Биткоин устанавливает для этого процесса около 10 минут, контролируя сложность алгоритма, который требуется решить узлам блокчейна. Создание нового блока для каждой транзакции сильно замедлило бы блокчейн, поэтому каждому блоку разрешено хранить множество транзакций. Мы накапливаем ожидающие транзакции в свойстве `pendingTransactions` и затем создаем новый блок, который и хранит их все. К примеру, один блок Биткоин содержит около 2500 транзакций.

АТАКИ ДВОЙНОГО РАСХОДОВАНИЯ

Предположим, у вас в кармане только две купюры по $1 и вы хотите купить чашку кофе, которая стоит $2. Вы даете бариста две однодолларовые купюры, он отдает вам кофе. В итоге денег у вас в кармане не остается, то есть купить вы больше ничего не сможете до тех пор, пока их не украдете или не подделаете. Подделка денег требует времени и не может осуществляться на месте в кофейне.

Цифровая валюта также может быть подделана. Например, мошенник может попытаться заплатить заданную сумму денег нескольким получателям. Предположим, у Джо есть только один биткоин, и он отправляет его Мэри (создавая блок с транзакцией «Джо Мэри 1»), а затем тут же отправляет один биткоин Алексу (создавая другой блок с транзакцией «Джо Алекс 1»). Таков пример *атаки двойного расходования*.

Биткоин и другие блокчейны реализуют механизмы для предотвращения подобных атак, применяя для этого согласованный процесс проверки каждого блока и разрешения конфликтов. В разделе 10.1 мы объясним правило длиннейшей цепочки, которое может использоваться для предотвращения добавления недействительных блоков.

Когда пользователь добавляет транзакции при помощи UI, показанного на рис. 9.5, мы вызываем метод `createTransaction()` для каждой из них, и он добавляет одну транзакцию в массив `pendingTransactions` (см. листинг 9.9). В конце метода `minePendingTransactions()`, когда новый блок добавлен в блокчейн, мы удаляем из этого массива все ожидающие транзакции.

Обратите внимание, что мы объявили переменную `_pendingTransactions` класса как `private`, поэтому она может быть изменена только через метод `createTransaction()`. Мы также предоставили следующий публичный геттер, возвращающий массив ожидающих транзакций:

```
get pendingTransactions(): Transaction[] {
    return [ ...this._pendingTransactions ];
}
```

У этого метода есть только одна строка, создающая клон массива `_pendingTransactions` (используя JS-оператор распространения, рассмотренный в разделе A.7 приложения). Создав клон, мы делаем копию данных транзакций. Кроме того, каждое свойство интерфейса `Transaction` является readonly, поэтому любая попытка изменить данные этого массива будет приводить к ошибке TS. Этот геттер используется в файле browser/main.ts, который отображает ожидающие транзакции в UI, как это показано в листинге 9.6. Та же техника клонирования используется с геттером `chain()`, возвращающим клон блокчейна.

Первичный блок создается вызовом метода `createGenesisBlock()` (листинг 9.9), за что отвечает сценарий browser/main.ts (см. листинг 9.6). Первичный блок содержит пустой массив транзакций, а вызов `mine()` для этого блока вычисляет его хеш. Добыча блока может занять некоторое время. Мы же не хотим, чтобы UI приостанавливался, поэтому `mine()` сделан асинхронным. Кроме того, мы добавили в его вызов ключевое слово `await`, чтобы гарантировать, что блок будет добавлен в блокчейн только после завершения добычи.

Наше блокчейн-приложение было создано в образовательных целях, поэтому при каждом его запуске пользователем блокчейн будет содержать только первичный блок. Пользователь начинает создавать ожидающие транзакции, и в определенный момент он щелкает по кнопке Confirm Transactions. Это приведет к вызову метода `minePendingTransactions()`, который создает новый экземпляр Block, вычисляет его хеш, добавляет блок в блокчейн и обнуляет массив `_pendingTransactions`.

> **ПРИМЕЧАНИЕ** После добычи нового блока майнер должен быть награжден, и мы позаботимся об этом в блокчейн-приложении, рассматриваемом в главе 10.

Взгляните еще раз на код метода `mineBlock()` в листинге 9.6, чтобы разобраться, как результаты добычи блока отображаются в UI.

Неважно, добываете ли вы первичный блок или же блок с транзакциями, процесс вызывает метод `mine()` в классе `Block` (см. листинг 9.8). Метод `mine()` выполняет цикл, вызывая `calculateHash()`, который, в свою очередь, вызывает `sha256()`, рассматриваемую далее.

9.4. ИСПОЛЬЗОВАНИЕ CRYPTO API ДЛЯ ГЕНЕРАЦИИ ХЕШЕЙ

Так как мы хотели создать блокчейн, который можно выполнять как в веб, так и в самостоятельном приложении, то для генерации SHA-256-хешей нужно было использовать два разных crypto API:

- Для веб-приложений мы можем вызывать crypto API объекта, поддерживаемый всеми браузерами.
- Самостоятельные приложения будут выполняться в среде Node.js, имеющей криптомодуль (https://nodejs.org/api/crypto.html). Мы будем использовать его для генерации хешей так же, как делали это в листинге 8.6.

Нам нужно, чтобы наша небольшая библиотека принимала решение о том, какой API использовать в среде выполнения. Поэтому клиентские приложения будут использовать одну функцию, не зная, какой конкретно crypto API будет в итоге выбран. Файл lib/universal_sha256.ts в следующем листинге объявляет три функции: `sha256_node()`, `sha256_browser()` и `sha256()`. Обратите внимание, что экспортируется только последняя.

Листинг 9.10. lib/universal_sha256.ts: обертка для crypto APIs

```
function sha256_node(data: string): Promise<string> {
    const crypto = require('crypto');
    return Promise.resolve(crypto.createHash('sha256').update(data)
➥ .digest('hex'));
}

async function sha256_browser(data: string): Promise<string> {

    const msgUint8Array = new TextEncoder().encode(data);

    const hashByteArray = await crypto.subtle.digest('SHA-256',
                          msgUint8Array);

    const hashArray = Array.from(new Uint8Array(hashByteArray));

    const hashHex = hashArray.map(b => ('00' +
➥ b.toString(16)).slice(-2)).join('');
    return hashHex;
```

Функция для использования в среде Node.js

Генерирует SHA-256-хеш

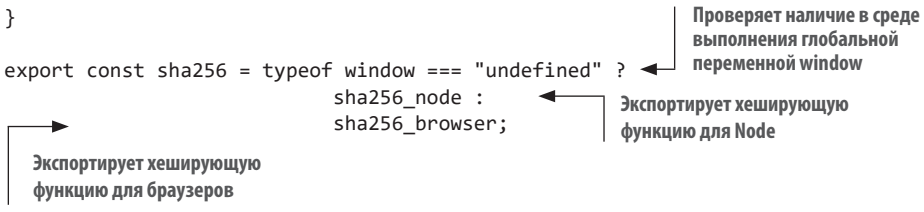
Функция для использования в браузере

Кодирует переданную строку в UTF-8

Хеширует данные

Преобразует ArrayBuffer в Array

Преобразует байты в шестнадцатеричную строку



Когда JS-файл содержит инструкции импорта или экспорта, он становится ES6-модулем (подробности в приложении). Универсальный модуль sha256.ts объявляет функции `sha256_node()` и `sha256_browser()`, но не экспортирует их. Эти функции становятся приватными и могут использоваться только внутри модуля.

Экспортируется для импорта другими сценариями только функция `sha256()`. Ее задача максимально проста — выяснить, выполняется модуль в браузере или нет. В зависимости от результата мы будем экспортировать либо `sha256_browser()`, либо `sha256_node()`, но под именем `sha256()`.

Мы нашли простое решение: если среда выполнения содержит глобальную переменную `window`, мы предполагаем, что этот код выполняется в браузере, и под именем `sha256()` экспортируем `sha256_browser()`. В противном случае под тем же именем мы отправляем `sha256_node()`. Другими словами, это можно описать как динамический экспорт.

Мы уже использовали crypto API Node.js в главе 8, но здесь мы обернули этот код в `Promise`:

```
Promise.resolve(crypto.createHash('sha256').update(data).digest('hex'));
```

Мы сделали это для выравнивания сигнатур функций `sha256_node()` и `sha256_browser()`.

Функция `sha256_browser()` выполняется в браузере и использует асинхронный crypto API, который и ее делает асинхронной. Согласно требованиям crypto API браузера, мы начинаем с использования метода `TextEncoder.encode()` API веб-кодирования (см. документацию к Mozilla http://mng.bz/1wd1), который кодирует строку в UTF-8 и возвращает результат в особом типизированном JS-массиве неприсвоенных 8-битных целых чисел (см. http://mng.bz/POqY). Затем crypto API браузера генерирует хеш в форме массивоподобного объекта (объекта со свойством `length` и индексированными элементами). После этого мы используем метод `Array.from()` для создания реального массива.

На рис. 9.8 приведен скриншот, сделанный в отладчике Chrome. Он демонстрирует фрагменты данных в переменных `hashByteArray` и `hashArray`. Точка останова была установлена сразу перед вычислением значения переменной `hashHex`. (В разделе 9.6 мы объясним, как производить отладку TS-кода в браузере.)

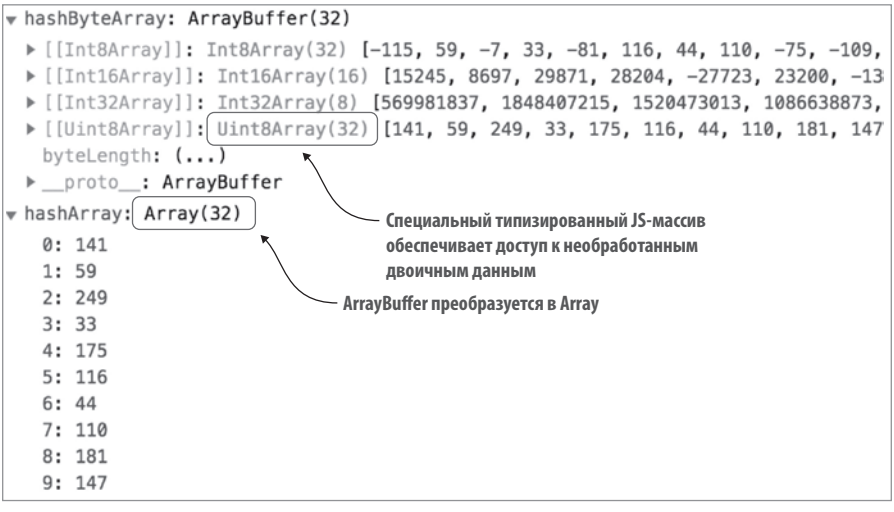


Рис. 9.8. Переменные hashByteArray и hashArray в отладчике

В завершение мы хотим преобразовать вычисленный хеш в строку из шестнадцатеричных значений и делаем это при помощи следующей инструкции:

```
const hashHex = hashArray.map(
                b => ('00' + b.toString(16)).slice(-2))
                .join('');
```

Мы преобразуем каждый элемент `hashArray` в шестнадцатеричное значение методом `Array.map()`. Некоторые из этих значений представлены одним знаком, а некоторые нуждаются в двух. Чтобы гарантировать, что однознаковые значения имеют перед собой `0`, мы конкатенируем `00` и шестнадцатеричное значение, а затем используем метод `slice(-2)`, чтобы взять только два знака справа. Например, сначала шестнадцатеричное значение становится `00a`, а затем `0a`.

Метод `join(' ')` конкатенирует все преобразованные элементы `hashArray` в строку `hashHex`, определяя в качестве разделителя пустую строку (как бы без разделителя). На рис. 9.9 показан фрагмент результата применения `map()` наряду с итоговым хешем в переменной `hashHex`.

Вы можете спросить, почему сигнатура функции `sha256_browser` объявляет возвращаемый тип `Promise`, но фактически возвращает строку. Мы объявили эту функцию как async, поэтому она автоматически обертывает свое возвращаемое значение в `Promise`.

К данному моменту вы должны уже понять код в директории lib и то, как его использует веб-клиент. Последняя часть рассмотрения — это самостоятельный клиент, который может использоваться вместо веб-клиента.

Рис. 9.9. Применение map() и join()

9.5. САМОСТОЯТЕЛЬНЫЙ БЛОКЧЕЙН-КЛИЕНТ

Директория node этого проекта содержит небольшой сценарий, который может создать блокчейн без веб UI, но мы хотели показать вам, что код в директории lib повторно используемый и что если приложение выполняется в Node.js, будет задействован верный crypto API. Листинг 9.11 показывает сценарий, который не нуждается в браузере, а выполняется в среде Node. Первая строка импортирует класс `Blockchain`, который скрывает детали конкретного crypto API.

Эта программа начинается с вывода сообщения о создании нового блокчейна. Процесс создания первичного блока может занять некоторое время, и первый `await` ожидает его завершения.

Что произошло бы, не используй мы `await` в первой строке, вызывающей `bc.createGenesisBlock()`? Код приступил бы к добыче блоков до создания первичного блока, и сценарий бы провалился, выдав ошибки среды выполнения.

После создания первичного блока сценарий переходит к созданию двух ожидающих транзакций и добыче нового блока. И снова `await` будет ожидать завершения этого процесса, а после мы создадим еще две транзакции и добудем еще один блок. В завершение программа выведет содержимое блокчейна в консоль.

ПРИМЕЧАНИЕ Node.js поддерживает `async` и `await`, начиная с версии 8.

Помните, что директория node содержит собственный файл tsconfig.json, как это было показано на рис. 9.1. А его содержимое было приведено в листинге 9.3.

Листинг 9.11. node/main.ts: самостоятельный сценарий для добычи блока

```
import { Blockchain } from '../lib/bc_transactions';

(async function main(): Promise<void> {
    console.log('? Initializing the blockchain, creating the genesis
➥ block...');

    const bc = new Blockchain();

    await bc.createGenesisBlock();

    bc.createTransaction({ sender: 'John', recipient: 'Kate', amount: 50 });
    bc.createTransaction({ sender: 'Kate', recipient: 'Mike', amount: 10 });

    await bc.minePendingTransactions(); 4((CO12-5))

    bc.createTransaction({ sender: 'Alex', recipient: 'Rosa', amount: 15 });
    bc.createTransaction({ sender: 'Gina', recipient: 'Rick', amount: 60 });

    await bc.minePendingTransactions();

    console.log(JSON.stringify(bc, null, 2));
})();
```

Создает новый блокчейн

Создает первичный блок

Создает ожидающую транзакцию

Создает новый блок и добавляет его в блокчейн

Выводит содержимое блокчейна

Перед стартом программы убедитесь, что код скомпилирован посредством запуска tsc. Запустить tsc нужно именно из директории src/node, чтобы обеспечить выбор компилятором опций из всей иерархии файлов tsconfig.json. Согласно основному tsconfig.json, скомпилированный код будет сохранен в директорию dist.

> **ПРИМЕЧАНИЕ** Опция компилятора `-p` позволяет вам указывать путь к действительному файлу конфигурации JSON. Например, вы могли бы скомпилировать TS-код, выполнив следующую команду: `tsc -p src/node/tsconfig.json`.

Теперь вы можете дать команду Node.js о запуске JS-версии кода, показанного в листинге 9.11. Если вы по-прежнему находитесь в директории src/node, то можете запустить приложение так:

```
node ../../dist/node/main.js
```

В листинге 9.12 показан вывод этой команды в консоли.

В то время как веб-версия этого приложения более интерактивна, самостоятельный его вариант выполняет весь процесс в пакетном режиме. Тем не менее в центре внимания этой главы была разработка именно веб-приложения. Теперь же мы покажем вам, как производить отладку TS-кода этого веб-приложения в браузере.

Листинг 9.12. Создание блокчейна в самостоятельном приложении

```
? Initializing the blockchain, creating the genesis block...
{
    "_chain": [
        {  ◄—— Первичный блок
            "previousHash": "0",
            "timestamp": 1540391674580,
            "transactions": [],
            "nonce": 239428,
            "hash": "0000d1452c893a79347810d1c567e767ea55e52a8a5ffc9743303f780b6c
➥ 308f"
        },
        {   ◄—— Второй блок
            "previousHash": "0000d1452c893a79347810d1c567e767ea55e52a8a5ffc974330
➥ 3f780b6c308f",
            "timestamp": 1540391675729,
            "transactions": [
                {
                    "sender": "John",
                    "recipient": "Kate",
                    "amount": 50
                },
                {
                    "sender": "Kate",
                    "recipient": "Mike",
                    "amount": 10
                }
            ],
            "nonce": 69189,
            "hash": "00006f79662bde59ff46cd57cff928977c465d931b2ba2d11e05868afcfe
➥ e836"
        },
        {  ◄—— Третий блок
            "previousHash": "00006f79662bde59ff46cd57cff928977c465d931b2ba2d11e05
➥ 868afcfee836",
            "timestamp": 1540391676138,
            "transactions": [
                {
                    "sender": "Alex",
                    "recipient": "Rosa",
                    "amount": 15
                },
                {
                    "sender": "Gina",
                    "recipient": "Rick",
                    "amount": 60
                }
            ],
            "nonce": 33462,
            "hash": "0000483b745526f48afde33435c21517dd72ea0a25407bc35be3f921029a
➥ 3209"
        }
    ],
    "_pendingTransactions": []  ◄—— Массив ожидающих транзакций пуст
}
```

9.6. ОТЛАДКА TYPESCRIPT В БРАУЗЕРЕ

Писать код на TS интересно, но браузеры этот язык не понимают. Они загружают и запускают только JS-версию приложения. Более того, исполняемый файл JS может быть оптимизирован и сжат, что делает его нечитаемым. Но при этом существует способ загружать в браузер оригинальный TS-код.

Для этого вам нужно сгенерировать файлы карт кода, которые отобразят исполняемые строки кода обратно в соответствующий исходный код, которым в нашем случае является TypeScript. Если браузер загрузит файлы карт кода, то вы сможете производить отладку оригинальных источников. В файле tsconfig.json, показанном в листинге 9.2, мы дали компилятору команду сгенерировать карты кода, которые в итоге будут иметь расширение .js.map, как показано на рис. 9.2.

Когда браузер загружает JS-код веб-приложения, он загружает только .js-файлы, даже если развернутое приложение включает файлы .js.map. Но если вы откроете инструменты разработчика, то браузер также загрузит и карты кода.

> **ПРИМЕЧАНИЕ** Если ваш браузер не загружает файлы карт кода для приложения, проверьте настройки инструментов разработчика. Убедитесь, что включена опция JavaScript source maps.

Теперь давайте загрузим наш веб-клиент (как объясняется в разделе 9.1) в браузере Chrome и откроем из вкладки Sources инструменты разработчика. В итоге экран будет разделен на три части, как показано на рис. 9.10.

Слева вы можете найти и выбрать исходный файл из вашего проекта. Мы выбрали universal_sha256.ts, и его TS-код показан в среднем окне. Справа вы видите панель отладки.

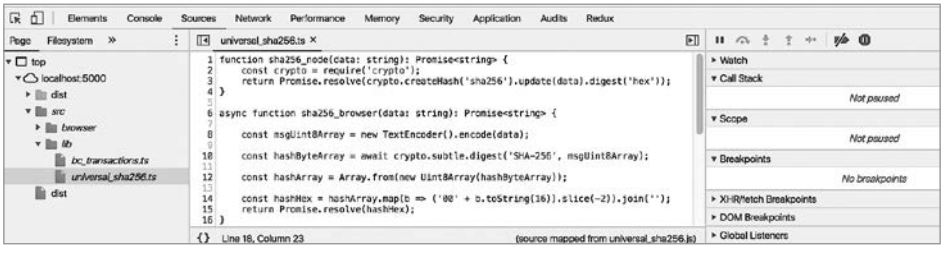


Рис. 9.10. Вкладка Sources of инструментов разработчика в Chrome

Давайте установим точку останова на строке 12, щелкнув слева от ее номера, а затем обновим окно браузера. Приложение остановит выполнение на указанной точке, и окно браузера будет выглядеть так, как показано на рис. 9.11.

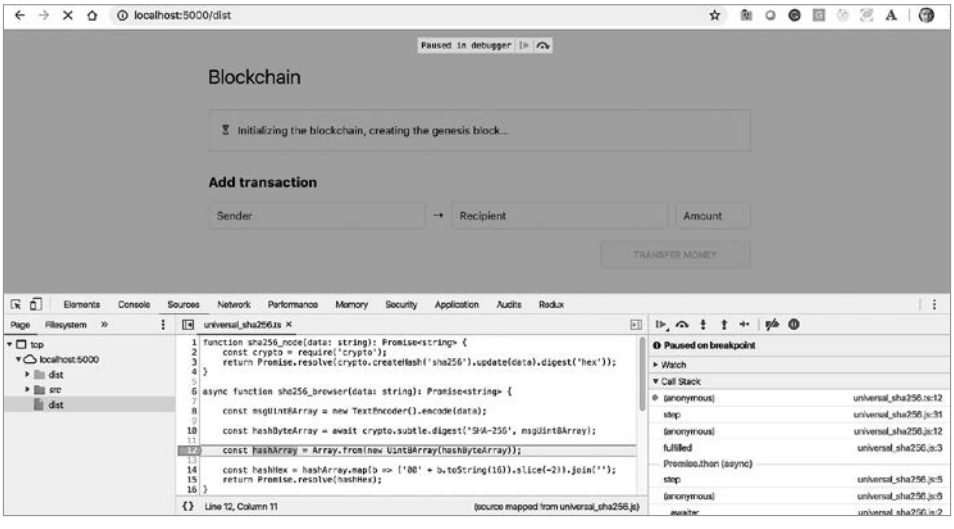


Рис. 9.11. Программа прервана на точке останова

Вы можете видеть значения переменных при наведении указателя мыши на их имена. На рис. 9.12 показаны значения переменной `msgUint8Array`, так как мы навели указатель на ее имя в строке 8.

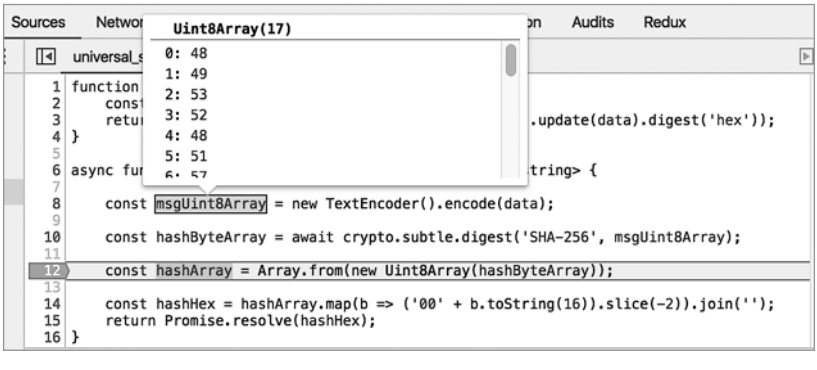


Рис. 9.12. Просмотр значений переменной при наведении указателя на ее имя

Вы также можете просматривать значения любой переменной или выражения, добавив их в область Watch (просмотра), расположенную в панели отладки справа. На рис. 9.13 показана программа, остановленная на строке 15. Мы добавили пару имен переменных и одно выражение, щелкнув на знаке «плюс» в области просмотра справа.

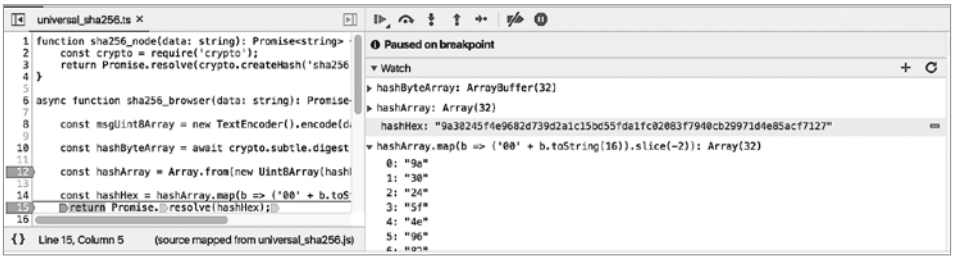


Рис. 9.13. Переменные в области просмотра

Если хотите удалить переменную или выражение из этой области, щелкните на знаке «минус» справа от них. На рис. 9.13 вы можете видеть минус справа от переменной `hashHex`.

Браузер Chrome имеет полноценный встроенный отладчик. Узнать о нем подробнее вы можете из видео «Debugging JavaScript — Chrome DevTools 101» по ссылке http://mng.bz/JzqK. С помощью карт кода вы сможете производить отладку TS даже, если JS был оптимизирован, минимизирован или минифицирован.

> **ПРИМЕЧАНИЕ** Разные IDE также имеют свои отладчики, и в главе 10 мы используем отладчик редактора VS Code для отладки самостоятельного TS-сервера. Но когда дело доходит до веб-приложений, мы предпочитаем выполнять отладку прямо в браузере, так как для обнаружения проблемы может потребоваться больше информации о контексте выполнения (сетевые запросы, хранилище приложения, хранилище сессий и т. д.).

ИТОГИ

- Как правило, TS-приложение представляет собой проект, состоящий из множества файлов. Одни содержат настройки конфигурации компилятора и бандлера, а другие исходный код.
- Вы можете организовать исходный код проекта, разделив на несколько директорий. В нашем приложении директория `lib` повторно использовалась двумя разными клиентскими приложениями — веб-приложением (в директории `browser`) и самостоятельным приложением (в директории `node`).
- Вы можете использовать npm-сценарии, чтобы создавать пользовательские команды для развертывания приложения, запуска веб-серверов и т. д.

10 Клиент-серверное взаимодействие посредством Node.js, TypeScript и WebSocket

В этой главе:

- ✓ Зачем блокчейну может понадобиться сервер.
- ✓ Правило длиннейшей цепочки.
- ✓ Как с помощью TypeScript создать WebSocket-сервер на Node.js.
- ✓ Практическое применение интерфейсов, абстрактных классов, квалификаторов доступа, перечислений и обобщенных типов в TypeScript.

В предыдущей главе вы узнали, что каждый майнер блока может получать несколько ожидающих транзакций, создавать действительный блок, включающий доказательство проделанной работы, и добавлять новый блок в блокчейн. Такой рабочий процесс легко отслеживается, когда есть только один майнер, создающий доказательство работы. В реальности же могут быть тысячи майнеров по всему миру, которые стремятся найти подходящий хеш для блока с одними и теми же транзакциями, что может вызывать конфликты.

В текущей главе мы используем TS, чтобы создать сервер, использующий для трансляции сообщений узлам блокчейна протокол WebSocket. Веб-клиенты также могут совершать запросы к этому серверу.

Хотя нашей основной деятельностью и остается написание кода в TS, мы впервые в этой книге используем несколько JS-пакетов:

- ws — библиотека Node.js, поддерживающая протокол WebSocket;
- express — небольшой фреймворк Node.js, предлагающий поддержку HTTP;
- nodemon — инструмент, перезапускающий Node.js приложения при обнаружении изменений файла сценария;
- lit-html — HTML-шаблоны в JavaScript для отображения DOM браузера.

Эти пакеты включены в файл package.json в качестве зависимостей (см. раздел 10.4.2).

Прежде чем обсуждать TS-код блокчейн-приложения этой главы, нужно рассмотреть следующие моменты:

- принцип блокчейна, известный как *правило длиннейшей цепочки*;
- создание и запуск блокчейн-приложения, эмулирующего более одного майнера блока.

Нам также потребуется пройтись по следующим элементам инфраструктуры:

- строение проекта, его файлы конфигурации и npm-сценарии;
- протокол WebSocket — что он собой представляет и чем он лучше HTTP для реализации сервера уведомлений. В качестве примера мы создадим простой WebSocket-сервер, который может передавать сообщения веб-клиенту.

Начнем с правила длиннейшей цепочки.

10.1. РАЗРЕШЕНИЕ КОНФЛИКТОВ С ПОМОЩЬЮ ПРАВИЛА ДЛИННЕЙШЕЙ ЦЕПОЧКИ

В главе 9 для начала добычи блока мы использовали упрощенный способ — пользователь для создания нового блока должен был щелкнуть по кнопке Confirm Transaction. В этой же главе мы рассмотрим более реалистичный пример блокчейна, в котором уже есть 100 блоков и пул ожидающих транзакций. Несколько майнеров могут взять эти ожидающие транзакции (например, по десять каждый) и начать добывать блоки.

> **ПРИМЕЧАНИЕ** Читая о добыче блоков в этой главе, помните, что мы говорим о децентрализованной сети. Узлы блокчейна работают параллельно, что может приводить к конфликтным ситуациям, если о праве добычи блока заявит не один узел. Именно поэтому для разрешения подобных конфликтов необходим механизм консенсуса.

Давайте возьмем троих произвольных майнеров, M1, M2 и M3, и предположим, что они нашли верный хеш (доказательство работы) и передают свои версии

нового блока под номером 101 в качестве кандидата на добавление в блокчейн. Каждый из претендующих блоков может содержать различные транзакции, но при этом каждый хочет стать блоком номер 101.

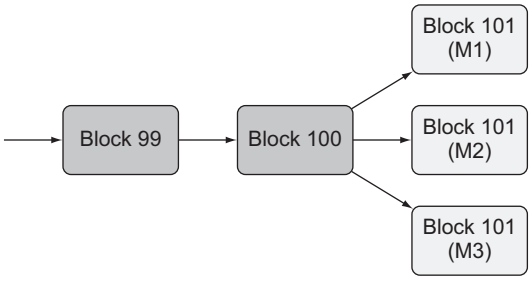


Рис. 10.1. Разветвленный блокчейн

Майнеры располагаются на разных континентах, и в узлах этих майнеров создаются ответвления, как показано на рис. 10.1. Временно существуют три ответвления, чьи первые 100 блоков идентичны, но 101-й в каждой ветви свой. В этот момент все эти три блока проверены (имеют верные хеши), но ни один еще не подтвержден.

Как решить, какой из них должен быть добавлен в блокчейн? Для ответа на этот вопрос мы используем *правило длиннейшей цепочки*. Многоузловая цепочка может содержать множество блоков, претендующих на добавление в сеть, но блокчейн подобен живому растущему организму, который добавляет узлы непрерывно. К моменту завершения добычи всеми майнерами одна из их цепочек будет использована неким другим майнером для добавления дополнительных блоков, став в связи с этим длиннейшей.

Для простоты мы рассмотрим только одного из наших трех майнеров, хотя их могут быть сотни одновременно. Наши майнеры на основе одной из версий блока 101 добывают блоки 102, 103 и т. д. Предположим, что какой-то другой майнер, M4, уже запросил длиннейшую цепочку и выбрал блок 101 из ветви M2, он также уже вычислил хеш для следующего блока-претендента, 102. Теперь давайте предположим, что M2 использует более мощный CPU, чем M1 и M3, следовательно, ветвь M2 в определенный момент была длиннейшей (имела дополнительный блок 101). Именно поэтому майнер M4 соединил блок 102 с блоком 101, как показано на рис. 10.2.

Теперь, несмотря на то что у нас есть три ветви, включающие блок 101, длиннейшая цепочка заканчивается блоком 102, и именно она будет принята всеми другими узлами. Ответвления, созданные майнерами M1 и M3, будут отвергнуты, а их транзакции отправлены обратно в пул ожидания. Далее другие майнеры смогут взять их в процессе создания других блоков.

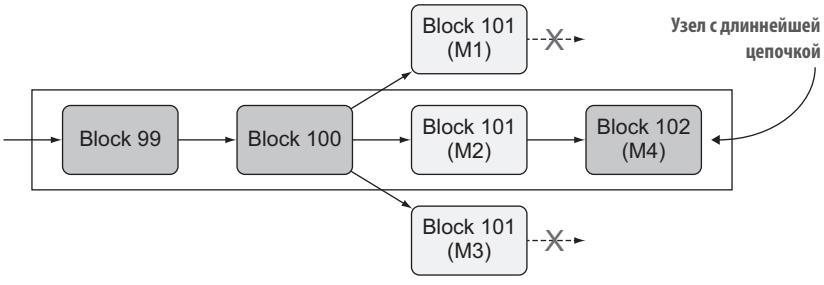


Рис. 10.2. M2 имеет длиннейшую цепочку

M2 получит награду за добычу блока 101, а майнеры M1 и M3 при этом просто потратили электричество на вычисление хеша для их версий блока 101. M4 аналогичным образом рисковал проделать бесполезную работу, когда выбрал блок M2 с верным, но неподтвержденным хешем.

ПРИМЕЧАНИЕ Блокчейны реализуют механизм, гарантирующий, что если транзакция включена в верный блок любым узлом, то при отклонении блоков она уже не отправляется обратно в пул ожидания.

В этом примере для упрощения мы использовали небольшое число блоков, но публичные блокчейны могут содержать тысячи узлов. В нашем сценарии был момент, когда ответвление майнера M2 представляло длиннейшую цепочку, которая была всего на один блок длиннее остальных ответвлений. В реальных блокчейнах может быть множество ответвлений различной длины. Поскольку блокчейн является распределенной сетью, блоки добавляются во множестве узлов, и каждый узел может иметь цепочку различной длины. Длиннейшая из них считается верной.

ПРИМЕЧАНИЕ Позднее в этой главе мы рассмотрим процесс запроса длиннейшей цепочки и получения ответов. На рис. 10.5–10.8 показано сообщение между двумя узлами в процессе запроса длиннейшей цепочки и анонсирования свежедобытых блоков.

Посмотрим, как правило длиннейшей цепочки помогает предотвратить *двойное расходование* и другие виды мошенничества. Предположим, что у некоего майнера есть друг Джон, имеющий на своем аккаунте $1000. Одна из транзакций в блоке 99 гласит: «Джон заплатил Мэри $1000». А что, если майнер решит смошенничать, произведя ответвление цепи и добавив другую транзакцию, гласящую уже: «Джон заплатил Алексу $1000» в блоке 100? Технически этот мошенник пытается обхитрить блокчейн, представив информацию так, что Джон потратил одну и ту же тысячу долларов дважды: первый раз через транзакцию к Мэри и второй раз — отправив средства Алексу. На рис. 10.3 показана попытка мошенника убедить других в том, что верный вид блокчейна — это ответвление, содержащее транзакцию John-to-Alex.

Вспомните, что хеш-значение блока легко проверить, в то время как его вычисление требует много времени. Итак, наш майнер-мошенник должен вычислить хеши для блоков 100, 101 и 102. Тем временем другие майнеры продолжают добывать новые блоки (103, 104, 105 и т. д.), добавляя их в цепочку, показанную в верхней части рис. 10.3. Никоим образом этот мошенник не сможет пересчитать все хеши и создать более длинную цепочку, чем все остальные узлы. Цепочка с большим объемом проделанной работы (длиннейшая) выиграет. Другими словами, шансы изменения существующих блоков близки к нулю, благодаря чему блокчейн считается неизменяемым.

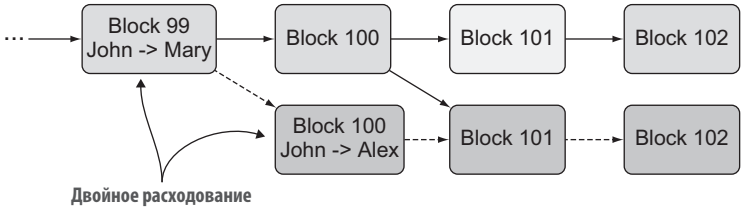


Рис. 10.3. Попытка двойного расходования

До сих пор мы не обсуждали, как узлы блокчейна общаются между собой. В следующем разделе мы как раз рассмотрим эту тему.

10.2. ДОБАВЛЕНИЕ СЕРВЕРА В БЛОКЧЕЙН

Как вы наверняка помните, мы восхваляли технологию блокчейн за ее полную децентрализованность и отсутствие сервера. Сервер, который мы добавим здесь, не будет центральным управляющим органом для создания, проверки и хранения блоков. Блокчейн может оставаться децентрализованным и по-прежнему использовать сервер для служебных сервисов вроде кэширования хешей, отправки новых транзакций, запросов длиннейшей цепочки или объявления о создании новых блоков. Позднее в этой главе (раздел 10.6.1) мы рассмотрим код подобного сервера. В этом же разделе мы обсудим процесс коммуникации между клиентами через этот сервер.

> **ПРИМЕЧАНИЕ** Антон Моисеев реализовал наше блокчейн-приложение с помощью одноранговой технологии под названием WebRTC (см. https://github.com/antonmoiseev/blockchain-p2p). При желании вы можете самостоятельно с ним поэкспериментировать.

Предположим, что узел M1 добыл блок и запросил длиннейшую цепочку у сервера, который отправляет этот запрос всем другим узлам этого блокчейна. Каждый из узлов отправляет ответ со своей цепочкой (только заголовки блоков), и сервер перенаправляет эти ответы M1.

Узел M1 получает длиннейшую цепочку и анализирует ее, проверяя хеши каждого блока. Затем он добавляет свежедобытый блок в эту длиннейшую цепочку, и она сохраняется локально. В течение некоторого времени узел M1 будет довольствоваться статусом «источника истины», поскольку у него будет длиннейшая цепочка до тех пор, пока кто-нибудь другой не добудет один или больше новых блоков.

Предположим, что M1 хочет создать новую транзакцию: «Джо отправил Мэри $1000». Создание нового блока для каждой транзакции было бы слишком медленным (пришлось бы вычислять множество хешей) и дорогим (счета за электричество). Как правило, один блок включает множество транзакций, и M1 может просто отправить свою новую транзакцию остальным членам блокчейна. Майнеры M2 и M3, а также любые другие будут делать то же самое для своих транзакций.

Все отправленные транзакции помещаются в пул ожидания, и любой узел может взять оттуда их группу (скажем, 10 штук) и начать добычу.

ПРИМЕЧАНИЕ Для нашей версии блокчейна мы будем использовать Node.js, чтобы создать сервер, реализующий отправку через WebSocket-соединение. В качестве альтернативы вы можете полностью обойтись без сервера, реализовав отправку с помощью предпочтительной одноранговой технологии вроде WebRTC (https://ru.wikipedia.org/wiki/WebRTC).

10.3. СТРУКТУРА ПРОЕКТА

Блокчейн-приложение из этой главы состоит из двух частей — сервера и клиента, обе из которых реализованы в TS. Сначала мы покажем структуру проекта и объясним, как запускать это приложение. Затем рассмотрим выборочные части кода, чтобы продемонстрировать практическое применение конкретных синтаксических конструкций TS.

Откройте в IDE проект в директории chapter10 и выполните `npm install` в окне терминала. Структура этого проекта показана на рис. 10.4.

В процессе сборки создается публичная директория, и файл public/index.html загружает скомпилированную версию файла client/main.ts наряду со всеми его импортируемыми сценариями. Это веб-клиент, который мы будем использовать для демонстрации узла блокчейн.

Файл server/main.ts содержит код, импортирующий дополнительные сценарии, а также запускающий серверы уведомлений WebSocket и блокчейн.

ПРИМЕЧАНИЕ В разделе 10.4 мы объясним, как создавать и запускать клиент и сервер.

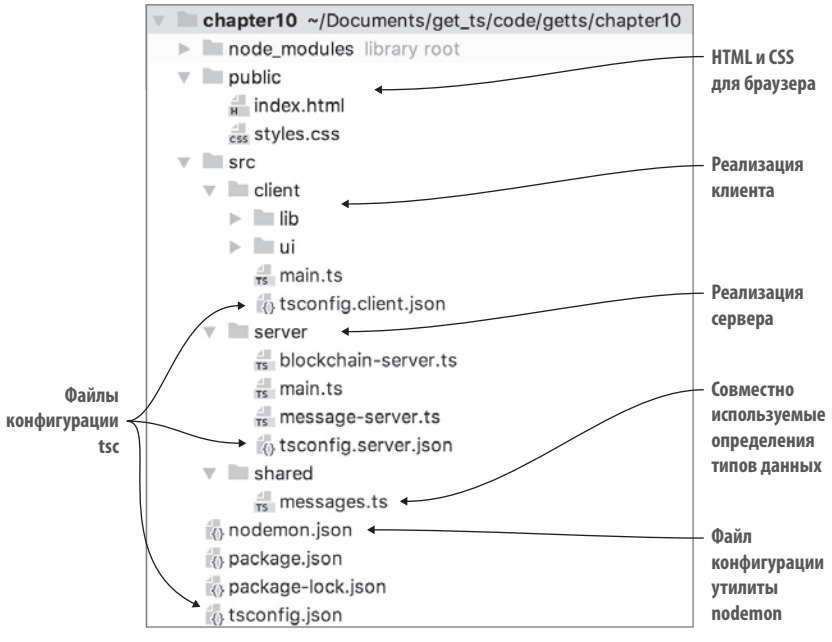


Рис. 10.4. Структура проекта

Заметили ли вы, что здесь присутствуют три файла конфигурации для компилятора TS? Файл tsconfig.json в корневой директории содержит опции компилятора, являющиеся общими и для клиента, и для сервера. Файлы tsconfig.client.json и tsconfig.server.json расширяют базовый tsconfig.json и добавляют опции компилятора, специфичные для клиента и сервера соответственно. Содержимое этих файлов мы рассмотрим в следующем разделе.

Наш сервер работает в среде Node.js, и его скомпилированный код будет храниться в директории build/server. Вы можете запустить этот сервер следующей командой:

```
node build/server/main.js
```

Как бы то ни было, исполняемый файл node не поддерживает перезагрузку в реальном времени (он не перезапустится, если вы измените и перекомпилируете TS-код сервера). Поэтому мы используем утилиту nodemon (https://nodemon.io), которая мониторит базу кода Node.js приложения и перезапускает Node.js-сервер при изменении этого файла. Если у вас установлена утилита nodemon, вы можете использовать ее вместо node. Например, вот как вы можете запустить среду Node.js и выполнить код, используя глобально установленную nodemon:

```
nodemon build/server/main.js
```

Если же у вас нет nodemon, то запустите ее так:

```
npx nodemon build/server/main.js
```

В нашем проекте мы будем запускать сервер с помощью команды nodemon, которая будет настроена в файле nodemon.json, рассмотренном в следующем разделе. Файл package.json включает раздел npm-сценариев, который мы также разберем в следующем разделе.

10.4. ФАЙЛЫ КОНФИГУРАЦИЙ ПРОЕКТА

Проект из этой главы включает несколько файлов конфигурации JSON, которые мы рассмотрим в следующем разделе.

10.4.1. Настройка компиляции TypeScript

Файл tsconfig.json может наследовать конфигурацию из другого файла при помощи свойства `extends`. В нашем проекте мы будем использовать три файла конфигурации:

- tsconfig.json содержит общие опции компилятора tsc для всего проекта;
- tsconfig.client.json содержит опции для компиляции клиентской части проекта;
- tsconfig.server.json включает опции для компиляции серверной части.

Следующий листинг демонстрирует содержимое основного файла tsconfig.json.

Листинг 10.1. tsconfig.json — общие опции tsc

```
{
    "compileOnSave": false,
    "compilerOptions": {
        "target": "es2017",
        "sourceMap": true,
        "plugins": [
            {
                "name": "typescript-lit-html-plugin".
            }
        ]
    }
}
```

Не выполняет автокомпиляцию при каждом изменении файлов TS

Компилирует в JavaScript при помощи синтаксиса, поддерживаемого ECMAScript 2017

Генерирует файлы карт кода

Этот плагин активирует HTML-автоподстановку для шаблонов строк с HTML-тегами

В качестве целевой версии компиляции мы указали `es2017`, потому что уверены, что пользователи этого приложения используют современные браузеры, поддерживающие все возможности спецификации ECMAScript 2017.

В следующем листинге показан файл tsconfig.client.json, содержащий те опции tsc, которые мы хотим использовать для компиляции клиентской части кода. Этот файл расположен в директории src/client.

Листинг 10.2. tsconfig.client.json — опции tsc для клиента

```
{
    "extends": "../../tsconfig.json",  <-- Наследует опции компиляции из этого файла
    "compilerOptions": {
        "module": "es2015",  <-- Использует в сгенерированном JS инструкции import/export.
        "outDir": "../../public",  <-- Помещает скомпилированный код в директорию public
        "inlineSources": true  <-- Выдает исходный TS-код и карты кода в одном файле
    }
}
```

Карты кода позволяют вам производить отладку TS в то время, как браузер выполняет JS. Они сообщают браузерным инструментам разработчика, какие строки скомпилированного кода (JavaScript) каким строкам исходного (TypeScript) соответствуют. Как бы то ни было, исходный код должен быть доступен браузеру, и вы можете либо развернуть TS наряду с JS на веб-сервере, либо использовать опцию `inlineSources`, как мы поступили здесь. Эта опция вложит оригинальный TS-код прямо в файлы карт кода.

> **ПРИМЕЧАНИЕ** Если вы не хотите раскрывать пользователям исходный код вашего приложения, не развертывайте карты кода в продакшене.

В следующем листинге показан файл tsconfig.server.json, который содержит tsc-опции для компиляции кода сервера. Этот файл расположен в директории src/server.

Листинг 10.3. tsconfig.server.json — опции tsc для сервера

```
                    Наследует опции конфигурации
                                из этого файла
{
    "extends": "../../tsconfig.json",
    "compilerOptions": {                          Преобразует инструкции import/export
        "module": "commonjs",                     в совместимый с commonjs код
        "outDir": "../../build"  <-- Помещает скомпилированный код в директорию build
    },
    "include": [
        "**/*.ts"  <-- Компилирует все .ts-файлы из всех поддиректорий
    ]
}
```

> **ПРИМЕЧАНИЕ** В идеальном мире вы бы никогда не встретили одинаковые опции компиляции в основном и наследованных файлах конфигурации. Но иногда некоторые IDE не обрабатывают наследование конфигурации tsc должным образом, поэтому может потребоваться продублировать опцию не в одном файле.

Теперь, когда у нас есть более одного файла конфигурации tsc, как мы дадим компилятору понять, какой использовать? В этом нам поможет опция `-p`. Следующая команда скомпилирует код веб-клиента, используя опции из tsconfig.client.json:

```
tsc -p src/client/tsconfig.client.json
```

Следующая команда скомпилирует код сервера, используя tsconfig.json:

```
tsc -p src/server/tsconfig.server.json
```

> **ПРИМЕЧАНИЕ** Если вы добавите в проект файлы конфигурации tsc с именами, отличными от tsconfig.json, то потребуется использовать опцию `-p` и указать путь к файлу, который вы хотите использовать. Например, если вы выполнили команду tsc в директории src/server, она не будет использовать опции из файла tsconfig.server.json, и вы можете столкнуться с неожиданными результатами компиляции.

Теперь давайте взглянем на зависимости и npm-сценарии этого проекта.

10.4.2. Что находится в package.json

В листинге 10.4 отображено содержимое package.json. В разделе scripts находятся пользовательские npm-команды, а раздел dependencies перечисляет всего три пакета, необходимых для запуска приложения (и клиентской, и серверной части):

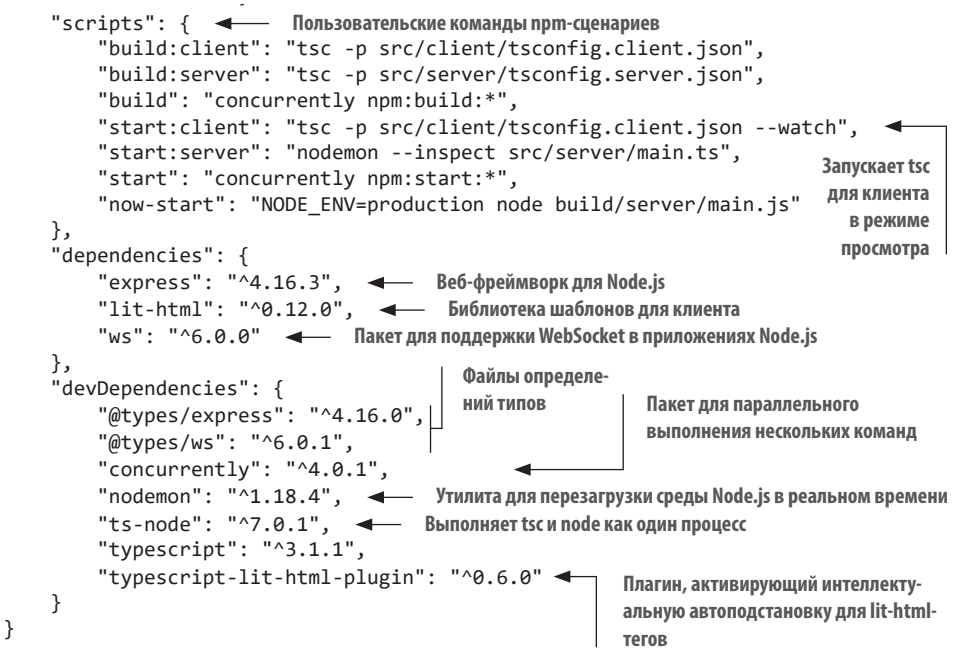
- *ws* — библиотека Node.js, поддерживающая протокол WebSocket;
- *express* — небольшой Node.js фреймворк, предлагающий поддержку HTTP;
- *lit-html* — HTML-шаблоны в JavaScript для отображения DOM браузера.

Веб-часть этого приложения использует библиотеку lit-html (https://github.com/Polymer/lit-html), выполняющую обработку шаблонов для JavaScript. При этом typescript-lit-html-plugin активирует автоподстановку (интеллектуальное дополнение ввода) в вашей IDE.

Раздел `devDependencies` включает пакеты, необходимые только в процессе разработки.

Листинг 10.4. package.json: зависимости нашего веб-приложения

```
{
    "name": "blockchain",
    "version": "1.0.0",
    "description": "Chapter 10 sample app",
    "license": "MIT",
```



`ts-node` запускает всего один процесс `Node`. После того как `Node` запущен, он регистрирует пользовательскую пару расширение/загрузчик при помощи механизма `require.extensions`. Когда вызов `require()`, совершенный `Node`, разрешается в файл с расширением .ts, `Node` вызывает пользовательский загрузчик, который по ходу компилирует TS в JS с использованием программного API tsc, не запуская при этом отдельный tsc-процесс.

Обратите внимание, что команда `start:client` запускает tsc в режиме просмотра (при помощи опции `--watch`). Это гарантирует, что как только вы измените и сохраните любой TS-код в клиенте, он будет перекомпилирован. Но что насчет перекомпиляции кода сервера?

10.4.3. Настройка nodemon

Мы можем запустить tsc-компилятор в режиме просмотра также и на сервере, и при изменении TS-кода JavaScript будет генерироваться повторно. Но наличия свежего JS-кода для сервера недостаточно — при каждом изменении кода нам также нужно перезапускать среду Node.js. Поэтому мы и установили утилиту nodemon, которая будет запускать процесс Node.js и мониторить JS-файлы в указанной директории.

Файл package.json (листинг 10.4) включает следующую команду:

```
"start:server": "nodemon --inspect src/server/main.ts"
```

> **ПРИМЕЧАНИЕ** Опция `--inspect` позволяет производить отладку выполняемого в Node.js кода в инструментах разработчика Chrome (см. «Руководство по отладке» для Node.js здесь http://mng.bz/wlX2). Nodemon просто передает Node.js опцию `--inspect`, что приводит к запуску последнего в режиме отладки.

Может показаться, что команда `start:server` для запуска TS-файла main.ts требует nodemon, но поскольку наш проект содержит файл nodemon.json, nodemon будет использовать его опции. Эти опции вы можете видеть ниже.

Листинг 10.5. nodemon.json: файл конфигурации для утилиты nodemon

```
{
    "exec": "node -r ts-node/register/transpile-only",   <-- Как запустить node
    "watch": [ "src/server/**/*.ts" ]                    <-- Наблюдает за всеми .ts файлами, размещенными во всех поддиректориях сервера
}
```

Команда `exec` позволяет нам определить опции для запуска Node.js. В частности, опция `-r` является сокращением для `--require module`, которая используется для предварительной загрузки модулей при запуске. В нашем случае она просит пакет `ts-node` предварительно загрузить более быстрый модуль транспиляции, который просто преобразует код из TS в JS, не выполняя при этом проверку типов. Этот модуль будет автоматически запускать компилятор TS для каждого файла с расширением .ts, загруженного Node.js.

Предварительно загружая модуль `transpile-only`, мы избавляемся от необходимости запускать для сервера отдельный tsc-процесс. Любой TypeScript-файл будет загружен и автоматически скомпилирован как часть одного процесса Node.js.

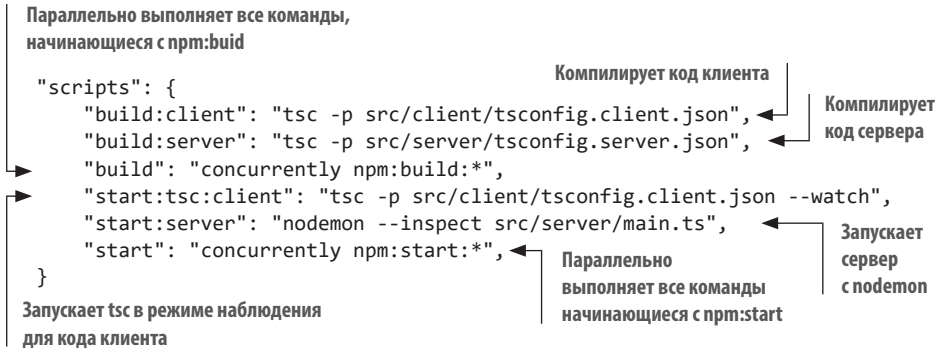
> **ПРИМЕЧАНИЕ** Вы можете использовать пакет ts-node разными способами. Например, для запуска Node.js с компиляцией TS: ts-node myScript.ts. Подробности вы можете найти на странице ts-node сайта npm: www.npmjs.com/package/ts-node.

Мы показали вам высокоуровневый обзор конфигурирования блокчейн-приложения. Следующим шагом будет рассмотрение этого приложения в действии.

10.4.4. Выполнение блокчейн-приложения

В этом разделе мы покажем, как запускать сервер и два клиента, эмулируя узлы блокчейна. Для запуска этих процессов будем использовать npm-сценарии, поэтому давайте подробнее рассмотрим раздел сценариев в файле package.json.

Листинг 10.6. Раздел сценариев файла package.json



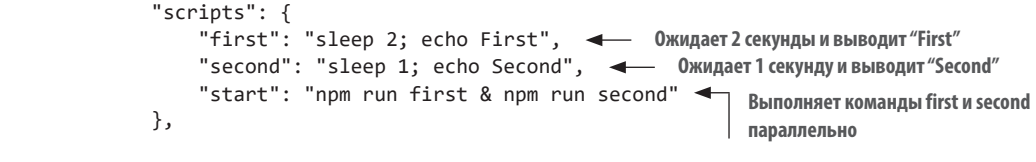

```
"scripts": {
    "build:client": "tsc -p src/client/tsconfig.client.json",
    "build:server": "tsc -p src/server/tsconfig.server.json",
    "build": "concurrently npm:build:*",
    "start:tsc:client": "tsc -p src/client/tsconfig.client.json --watch",
    "start:server": "nodemon --inspect src/server/main.ts",
    "start": "concurrently npm:start:*",
}
```

Первые две команды `build` запускают компилятор tsc для клиента и сервера соответственно. Эти процессы компилируют TS в JS. Компиляция кода клиента и сервера может выполняться конкурентно, поэтому третья команда использует npm-пакет под названием «concurrently» (www.npmjs.com/package/concurrently), который позволяет выполнять несколько команд конкурентно.

Команда `start:tsc:client` компилирует код клиента в режиме просмотра, а `start:server` запускает сервер, используя nodemon, как было описано в предыдущем разделе. Третья команда `start` конкурентно выполняет `start:tsc:client` и `start:server`.

Обычно можно выполнять более одной команды, просто добавляя между ними амперсанд. Например, вы можете определить две пользовательские команды `first` и `second`, а затем выполнить их конкурентно при помощи `npm start`. В npm-сценариях амперсанд означает «выполнить эти команды конкурентно».

Листинг 10.7. Конкурентное выполнение при помощи амперсанда




```
"scripts": {
    "first": "sleep 2; echo First",
    "second": "sleep 1; echo Second",
    "start": "npm run first & npm run second"
},
```

Если вы выполните `npm start`, она выведет в консоль `Second`, а затем `First`, доказывая тем самым, что команды были выполнены конкурентно. При замене `&` на `&&` первым в выводе будет `First`, а следом `Second`, что будет признаком последовательного выполнения.

Использование пакета concurrently вместо амперсанда дает вам четкое разделение сообщений, выведенных в консоль каждым конкурентным процессом.

ИСПОЛЬЗОВАНИЕ АМПЕРСАНДОВ В NPM-СЦЕНАРИЯХ В WINDOWS

Одиночный амперсанд (&) в Windows не выполняет команды параллельно. Для их одновременного выполнения можно использовать пакет npm-run-all (www.npmjs.com/package/npm-run-all).

Вот как будет выглядеть код из листинга 10.7 в Windows:

```
"scripts": {
    "first": "timeout /T 2 > nul && echo First",
    "second": "timeout /T 1 > nul && echo Second",
    "start": "run-p first second"
}
```

Timeout – это команда в Windows, альтернативная команде sleep в Unix

run-p – это команда из пакета npm-run-all

Вместо `sleep` мы используем `timeout` и указываем, как долго процесс должен оставаться неактивным при помощи параметра `/T`. В процессе выполнения команда `timeout` выводит в консоль обратный отсчет ожидания в секундах. Чтобы избежать появления этих сообщений вместе с выводом `First` и `Second`, мы перенаправляем вывод `timeout` в `nul`, которая эти сообщения убирает.

`run-p` – это команда из пакета npm-run-all, которая запускает указанные npm-сценарии параллельно.

Чтобы запустить блокчейн-приложение, выполните команду `npm start` в терминале. В следующем листинге показан вывод терминала. Мы выполняем команды конкурентно: `start:tsc:client` и `start:server`. Каждая строка в выводе терминала начинается с имени процесса (в квадратных скобках), который произвел это сообщение.

Листинг 10.8. Запуск блокчейн-приложения

```
> blockchain@1.0.0 start /Users/yfain11/Documents/get_ts/code/getts/chapter10
10:47:18 PM - Starting compilation in watch mode...
[start:tsc:client]
[start:server] [nodemon] 1.18.9
[start:server] [nodemon] to restart at any time, enter `rs`
[start:server] [nodemon] watching: src/server/**/*.ts
[start:server] [nodemon] starting `node -r ts-node/register/transpile-only
➥ --inspect src/server/main.ts`
[start:server] Debugger listening on
➥ ws://127.0.0.1:9229/2254fc00-3640-4390-8302-1e17285d0d23
[start:server] For help, see: https://nodejs.org/en/docs/inspector
[start:server] Listening on http://localhost:3000
[start:tsc:client] 10:47:21 PM - Found 0 errors. Watching for file changes.
```

Вывод процесса start:tsc:client

Отладчик Node.js выполняется локально через порт 9229

Вывод процесса start:server

Сервер запущен и работает через порт 3000

Вывод процесса start:tsc:client

ПРИМЕЧАНИЕ URL отладчика Node.js — это ws://127/0/0/1. Он начинается с ws, указывая, что инструменты разработчика подключаются к отладчику, используя протокол WebSocket, с которым вы познакомитесь в следующем разделе. После запуска отладчика Node.js вы увидите зеленый шестиугольник в панели инструментов разработчика Chrome. Ознакомьтесь подробнее с отладчиком Node.js в статье Пола Айриша (Paul Irish) «Debugging Node.js with Chrome DevTools» на Medium, по ссылке http://mng.bz/qX6J.

Теперь сервер активен, и ввод localhost:3000 запустит первый клиент нашего блокчейна. Спустя пару секунд будет сгенерирован первичный блок, и вы увидите веб-страницу, как показано на рис. 10.5.

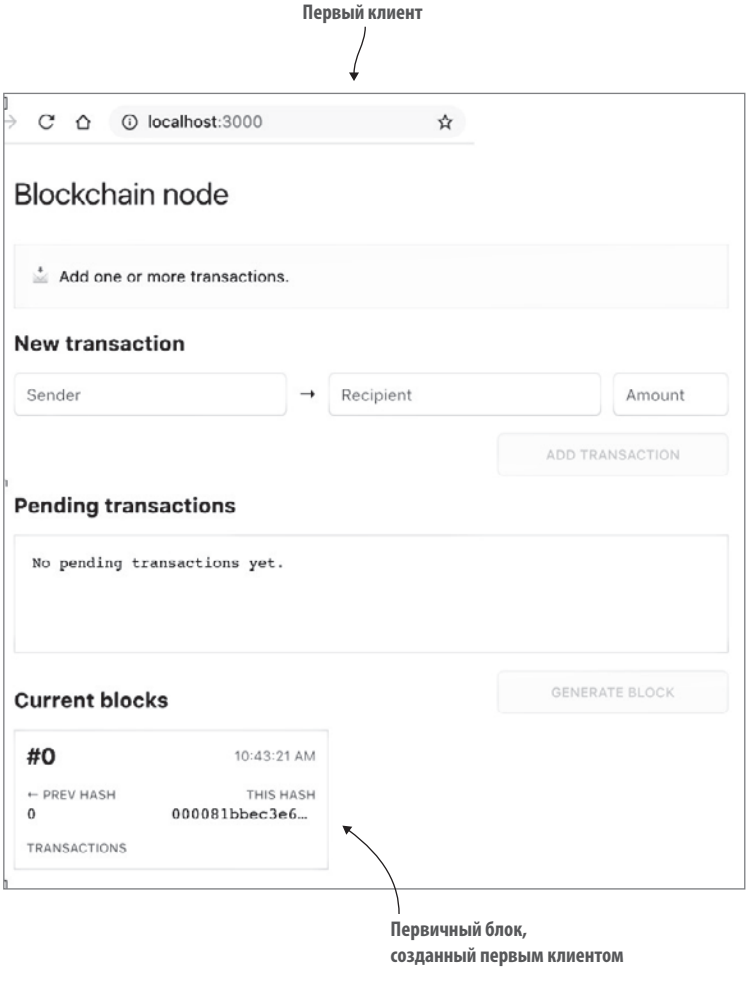


Рис. 10.5. Представление самого первого клиента блокчейна

Мы показываем, что происходит под капотом в разделе 10.6, после того как объясним взаимодействие клиентов друг с другом через WebSocket-сервер. На данный момент будет достаточно сказать, что, прежде чем создавать какие-либо блоки, клиент делает запрос к серверу, чтобы найти длиннейшую цепочку.

Запустите второй клиент, открыв отдельное окно браузера на localhost:3000, как показано на рис. 10.6.

Теперь обсудим случай использования, когда только первый клиент добавляет ожидающие транзакции, начинает добычу блока и приглашает другие узлы присоединиться к добыче. К этому времени наш блокчейн имеет первичный блок, и сервер передает его всем подключенным клиентам (на рис. 10.6 это клиент справа). Обратите внимание, что оба клиента видят один и тот же блок, добытый первым клиентом. Затем первый клиент вводит две транзакции, как показано на рис. 10.7. Пока никаких запросов для генерации нового блока сделано не было.

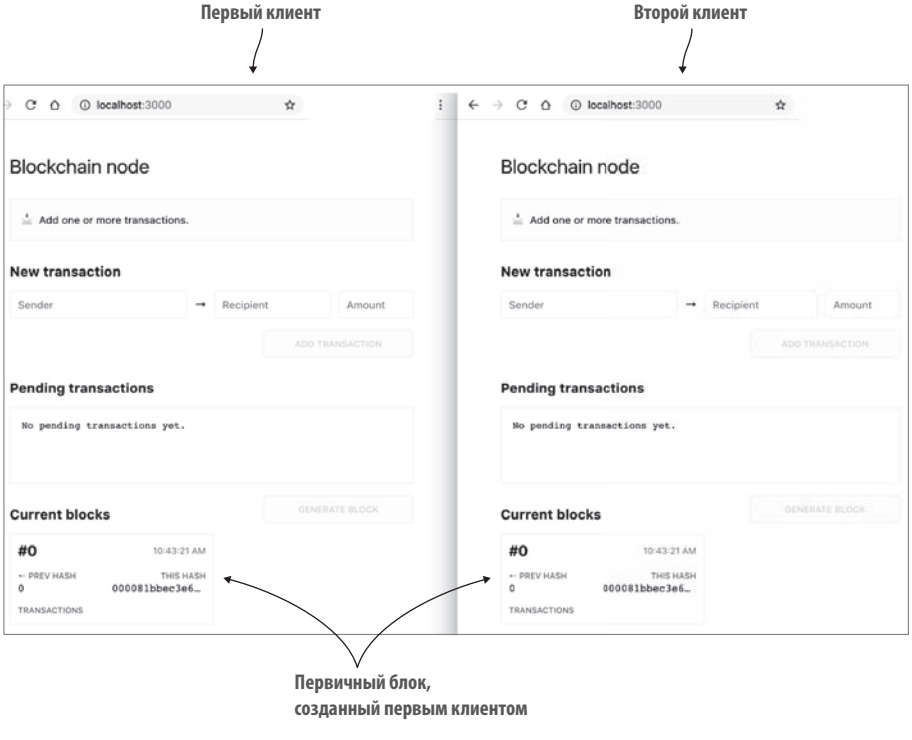


Рис. 10.6. Представление двух первых клиентов блокчейна

ПРИМЕЧАНИЕ В то время как клиент добавляет ожидающие транзакции, взаимосвязи с другими клиентами не происходит и сервер обмена сообщениями не используется.

Теперь первый клиент начинает добычу, щелкая по кнопке `GENERATE BLOCK`. За кадром он отправляет серверу сообщение с содержимым блока, говоря, что первый клиент начал добычу этого блока. Сервер передает это сообщение всем подключенным клиентам, и они также начинают добывать этот блок.

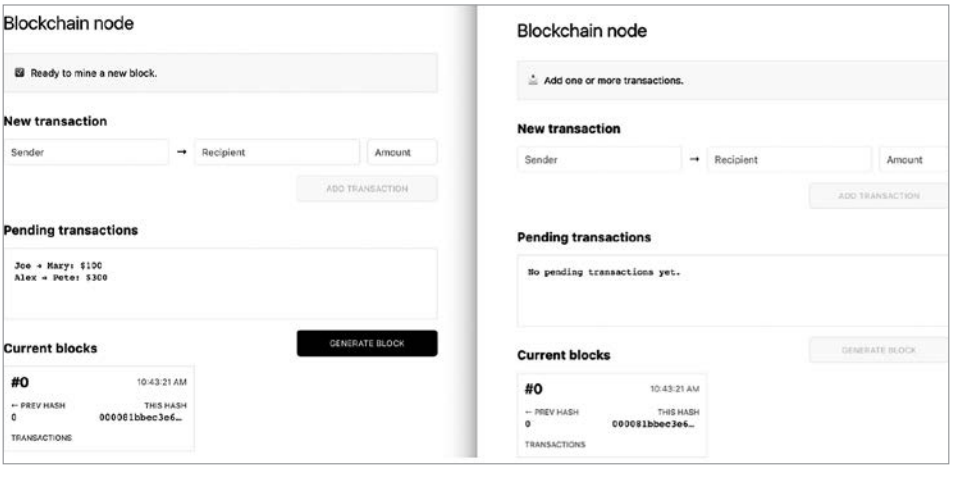


Рис. 10.7. Один клиент создал ожидающие транзакции

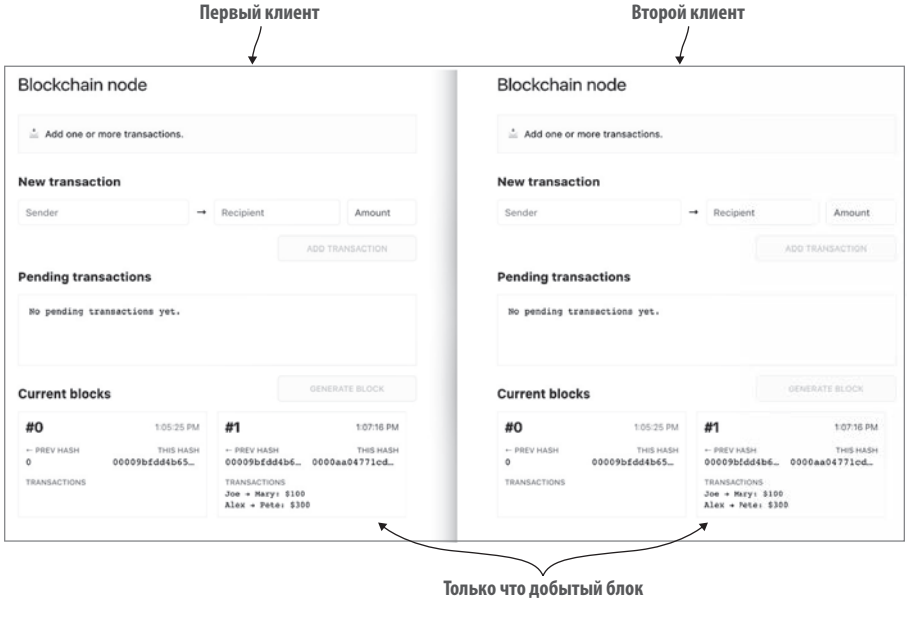


Рис. 10.8. Каждый клиент добавляет один и тот же блок

Один из клиентов окажется быстрее, и о добавлении его нового блока в блокчейн будет сообщено другим подключенным клиентам, чтобы они могли также добавить этот блок в свои версии блокчейна. После того как блоки приняты в блокчейн, все клиенты будут содержать одинаковые блоки, как показано на рис. 10.8.

В разделе 10.6 мы обсудим сообщения, проходящие через сокеты в процессе добычи блоков, и процесс добычи в многоузловой сети уже не будет выглядеть как магия. На данный момент рис. 10.9 показывает диаграмму последовательности, которая может помочь вам проследить обмен сообщениями, предполагая наличие двух узлов: M1 и M2.

В этом примере пользователь работает с узлом M2. В процессе создания ожидающих транзакций никакие сообщения другим узлам не передаются. Передача сообщений начинается, когда пользователь инициирует операцию по созданию блока. Узел M2 отправляет сообщение request new block (запрос нового блока) и начинает добычу в узле M2. Сервер рассылает это сообщение другим узлам (в нашем случае M1), которые также начинают добычу, соревнуясь с M2.

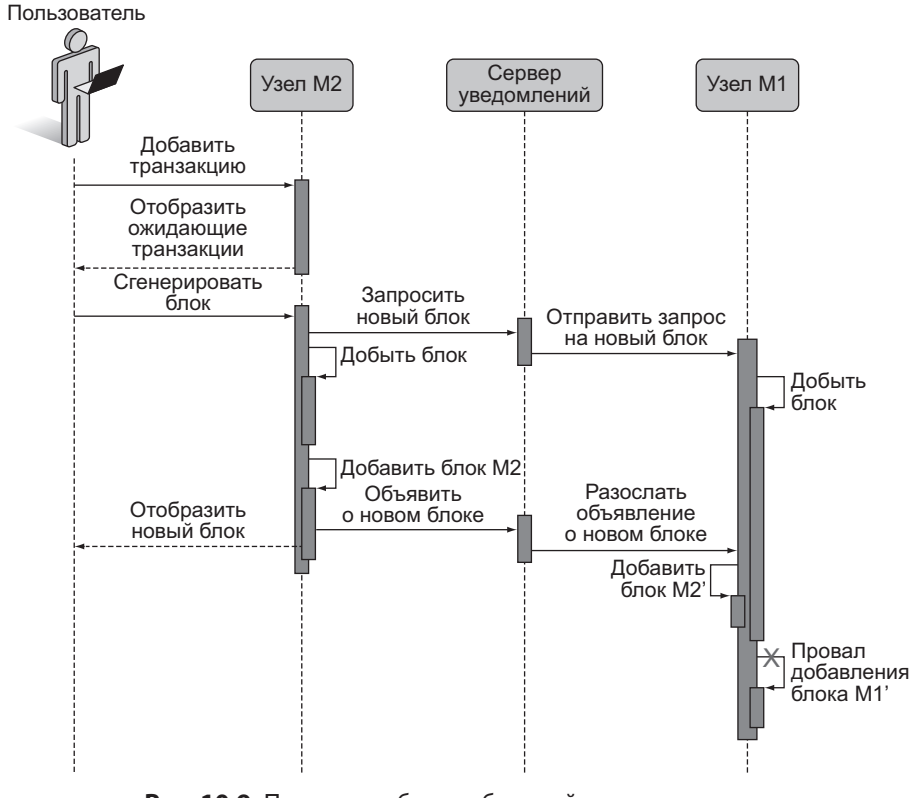


Рис. 10.9. Процесс добычи в блокчейне из двух узлов

На этой диаграмме узел M2 оказывается быстрее, первым объявляя о добыче нового блока, и сервер транслирует его сообщение всему блокчейну. Узел M1 добавил блок M2 в его локальный блокчейн. Несколько позже M1 тоже закончил добычу, но добавление блока M1 провалилось, так как уже был принят и добавлен блок, добытый M2, и существующая цепочка является длиннейшей. Другими словами, блок M2 был принят как выигравший по консенсусу между M1 и M2. В завершение новый блок отображается в UI пользователя.

Код, соответствующий провалившейся попытке добавления нового блока, будет показан позже в этой главе (во врезке «Когда новый блок отвергнут»).

> **ПРИМЕЧАНИЕ** В нашем упрощенном блокчейне UI и логика создания блоков реализованы в одном веб-приложении. Реальное же блокчейн-приложение имело бы раздельные приложения для добавления транзакций и добычи блоков.

Теперь, когда вы видели, как работает наше приложение, познакомимся с клиент-серверной коммуникацией через WebSocket. Если вы уже знакомы с WebSocket, то можете сразу перейти к разделу 10.6.

10.5. КРАТКОЕ ЗНАКОМСТВО С WEBSOCKETS

Блокчейн-приложение в этой главе будет передавать уведомления, используя протокол WebSocket (https://en.wikipedia.org/wiki/WebSocket), поэтому мы хотим в общих чертах представить вам этот двоичный протокол с низкой нагрузкой, поддерживаемый всеми современными браузерами, а также всеми веб-серверами, написанными на Node.js, .Net, Java, Python и т. д.

Протокол WebSocket позволяет осуществлять двунаправленный поток передачи сообщений с текстовыми и двоичными данными между браузерами и веб-серверами. В противоположность протоколу HTTP WebSocket не основан на принципе запрос-ответ. Это означает, что как клиентское, так и серверное приложение могут в реальном времени инициировать передачу данных другой стороне, как только эти данные станут доступны. Благодаря этому WebSocket очень подходит для ряда приложений:

- live-трейдинг, аукционы и спортивные уведомления;
- контроль медицинского оборудования через сеть;
- чаты;
- многопользовательские онлайн-игры;
- обновления потоков социальных сетей в реальном времени;
- блокчейн.

Подобные приложения имеют одну общую особенность: существует сервер (или устройство), которому может потребоваться отправить пользователю мгновенное уведомление о произошедшем событии. Это отличается от случая, когда пользователь сам решает отправить серверу запрос для получения обновленных данных.

К примеру, вы можете использовать WebSocket для мгновенной отправки уведомлений всем пользователям в момент продажи акций на фондовой бирже. Или же сервер может рассылать уведомления от одного узла блокчейна к другим. Важно понять, что WebSocket-сервер может передавать уведомление клиенту, не нуждаясь в получении от него запроса данных.

В примере нашего блокчейна майнер M1 может начать или закончить добычу блока, и все другие узлы должны сразу об этом узнать. M1 отправляет сообщение WebSocket-серверу, заявляя о новом блоке, и сервер может незамедлительно передать это сообщение всем остальным узлам.

10.5.1. Сравнение протоколов HTTP и WebSocket

При использовании основанного на запросах протокола HTTP клиент отправляет через соединение запрос и ожидает ответа. И запросы, и ответы используют одно и то же соединение между браузером и сервером. Сначала отправляется запрос, а затем по тому же каналу возвращается ответ. Представьте себе узкий мост через речку, проехать через который встречные машины могут только по очереди. Машины со стороны сервера могут проехать по мосту только после того, как проедут встречные машины со стороны клиента. В мире веба такой тип связи называется *полудуплексным*.

В противоположность этому протокол WebSocket позволяет данным перемещаться через одно соединение в обоих направлениях одновременно (*дуплекс*), и обмен данными может инициировать каждая сторона. Это напоминает дорогу с двусторонним движением. Еще одна аналогия — это беседа по телефону, когда двое собеседников могут и говорить, и быть услышанными одновременно. WebSocket-соединение остается активным продолжительное время, что дает дополнительную выгоду: низкая задержка при взаимодействии между клиентом и сервером.

Типичный HTTP-запрос/ответ добавляет к данным приложения несколько сотен байтов (HTTP-заголовки). Предположим, вы хотите написать веб-приложение, сообщающее последние цены на акции каждую секунду. В случае с HTTP такое приложение будет вынуждено отправлять HTTP-запрос (около 300 байт) и получать цену на акции, которая будет возвращена в виде HTTP-ответа размером еще 300 байт.

При использовании WebSocket вся эта нагрузка снижается до пары байтов. Кроме того, нет необходимости продолжать отправлять запросы новой цены каждую

секунду. Конкретные акции какое-то время могут вообще не продаваться, и новое значение будет передано сервером, только когда цена наконец-то изменится.

Каждый браузер для создания и управления соединением сокета с сервером поддерживает объект WebSocket (см. документацию Mozilla касательно WebSocket здесь: http://mng.bz/1j4g). Изначально браузер устанавливает регулярное HTTP-соединение с сервером, но затем ваше приложение запрашивает обновление соединения, указывая URL сервера, поддерживающего WebSocket. После этого сообщение происходит уже без участия HTTP. Адреса конечных точек WebSocket начинаются с ws вместо http (например, ws://localhost:8085). Аналогичным образом для защищенной связи стоит использовать wss вместо https.

Протокол WebSocket основан на событиях и обратных вызовах. Например, когда ваше браузерное приложение устанавливает соединение с сервером, оно получает событие `connection` и активирует для его обработки обратный вызов. Для обработки данных, которые сервер может отправить через это соединение, код клиента ожидает событие `message`, передающее соответствующий обратный вызов. Если соединение закрывается, то отправляется событие `close`, чтобы приложение могло соответственно отреагировать. В случае ошибки объект WebSocket получает событие `error`.

На стороне сервера вам потребуется обрабатывать аналогичные события. Их имена могут отличаться в зависимости от используемого на сервере ПО WebSocket. В блокчейн-приложении этой главы для реализации уведомлений с помощью WebSocket-сервера мы используем среду Node.js.

10.5.2. Передача данных от сервера Node к простому клиенту

Для более близкого знакомства с WebSocket давайте рассмотрим простой случай: сервер передает данные маленькому браузерному клиенту при его подключении к сокету. Для этого клиенту не потребуется отправлять запрос данных — сервер сам инициирует связь. Чтобы добавить поддержку WebSocket, мы используем npm-пакет ws (www.npmjs.com/package/ws), как вы видели в package.json в листинге 10.4. При этом нам потребуются определения типов `@types/ws`, чтобы компилятор TS не жаловался, когда мы будем использовать API из этого пакета.

Текущий раздел приводит пример совсем небольшого WebSocket-сервера: он будет передавать простому HTML/JavaScript-клиенту сообщение «This message was pushed by the WebSocket server» в момент подключения этого клиента к сокету. Мы намеренно не хотим, чтобы клиент отправлял серверу какие-либо запросы, чтобы вы могли видеть, что сервер способен отправлять данные без получения предварительных запросов.

В примере этого приложения создаются два сервера. HTTP-сервер (реализованный с помощью фреймворка Express) выполняется через порт 8000 и отвечает за отправку браузеру начальной HTML-страницы. Когда эта страница загружена, она тут же подключается к серверу WebSocket, выполняемому через порт 8085. А сервер в момент установки соединения передает сообщение с приветствием.

Код этого приложения расположен в файле server/simple-websocket-server.ts и показан в следующем листинге.

Листинг 10.9. simple-websocket-server.ts: простой WebSocket-сервер

```
import * as express from "express";
import * as path from "path";
import { Server } from "ws";

const app = express();

// HTTP сервер
app.get('/', (req, res) => res.sendFile(
    path.join( dirname, '../../public/simple-websocket-client.html')));

const httpServer = app.listen(8000, 'localhost', () => {
    console.log('HTTP server is listening on localhost:8000');
});

// WebSocket сервер
const wsServer = new Server({port: 8085});
console.log('WebSocket server is listening on localhost:8085');

wsServer.on('connection',
        wsClient => {
            wsClient.send('This message was pushed by the WebSocket server');

            wsClient.onerror = (error) =>
                console.log(`The server received: ${error['code']}`);
        }
);
```

Мы будем использовать сервер из модуля ws, чтобы инстанцировать WebSocket-сервер

Инстанцирует фреймворк Express

Когда HTTP-клиент подключается через корневой путь, HTTP-сервер отправляет обратно этот HTML-файл

Запускает HTTP-сервер через порт 8000

Запускает WebSocket-сервер через порт 8085

Прослушивает событие connection от клиентов

Передает сообщение только что подключившемуся клиенту

Обрабатывает ошибки соединения

ПРИМЕЧАНИЕ Мы могли бы запустить экземпляры серверов на одном порте, и сделаем это позже в листинге 10.12. Сейчас же для простоты объяснения оставим HTTP- и WebSocket-серверы на разных портах.

Как только какой-либо клиент подключится к нашему WebSocket-серверу через порт 8085, событие `connection` будет отправлено на сервер, при этом сервер также получит ссылку на объект, представляющий этот конкретный клиент. При помощи метода `send()` сервер отправляет клиенту приветствие. Если к тому же сокету через порт 8085 подключится другой клиент, то он получит то же приветствие.

РАЗРЕШЕНИЕ ПУТЕЙ В NODE.JS

Следующая строка кода начинается с `app.get()`, она отображает URL HTTP-запроса в конкретную конечную точку кода или файла на диске (в данном случае это `GET`, но мог быть `POST` или другой запрос). Давайте рассмотрим следующий фрагмент кода:

```
app.get('/',  ◄── Сервер получает HTTP GET с основным URL
    (req, res) => res.sendFile( ◄── Отправляет файл обратно клиенту через объект HTTP-ответа
        path.join( dirname, '../../public/simple-websocket-
➥ client.html'))); ◄── Создает абсолютный путь к HTML-файлу
```

Метод `path.join()` в качестве стартовой точки использует переменную среды `_dirname`, а затем создает полный абсолютный путь. `_dirname` представляет имя директории главного модуля.

Предположим, что мы запускаем сервер следующей командой:

```
node build/server/simple-websocket-server.js
```

В этом случае значение `_dirname` будет путем к директории build/server. Соответственно, код будет идти из этой директории на два уровня вверх и на один вниз в директорию public, где расположен файл simple-websocket-client.html.

```
path.join(_dirname,'../../public/simple-websocket-client.html')
```

Чтобы сделать эту строку на 100% кроссплатформенной, будет безопаснее написать ее без использования прямого слеша в качестве разделителя.

```
path.join( dirname, '..', '..', 'public', 'simple-websocket-
➥ client.html')
```

ПРИМЕЧАНИЕ Как только новый клиент подключается к серверу, ссылка на это соединение добавляется в массив `wsServer.clients`, благодаря чему вы при необходимости можете рассылать сообщения всем подключенным клиентам: `wsServer.clients.forEach(client ? client.send('…'));`. Содержимое файла public/simple-websocket-client.http показано в следующем листинге. Это простой JavaScript/HTTP-клиент, использующий браузерный объект WebSocket.

Листинг 10.10. simple-websocket-client.html: простой клиент WebSocket

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
    </head>
```

```
    <body>
        <span id="messageGoesHere"></span>

        <script type="text/javascript">
            const ws = new WebSocket("ws://localhost:8085");

            const mySpan = document.getElementById("messageGoesHere");

            ws.onmessage = function(event){
                mySpan.textContent = event.data;
            };

            ws.onerror = function(event) {
                console.log(`Error ${event}`);
            }
        </script>
    </body>
</html>
```

Устанавливает сокет-соединение

Получает ссылку на элемент DOM для показа сообщений

Обратный вызов для обработки сообщений

Отображает сообщение в элементе

В случае ошибки браузер выводит в консоль сообщение об ошибке

Когда браузер загружает simple-websocket-client.html, его сценарий подключается к вашему WebSocket-серверу по адресу ws://localhost:8085. В этот момент сервер обновляет протокол с HTTP на WebSocket. Обратите внимание, что протокол ws, а не http.

Чтобы увидеть этот пример в работе, выполните `npm install` и скомпилируйте код, используя пользовательскую команду из package.json:

```
npm run build:server
```

Скомпилированная версия всех TS-файлов из директории server будет сохранена в директории build/server. Запустите этот простой WebSocket-сервер так:

```
node build/server/simple-websocket-server.js
```

Вы увидите в консоли следующие сообщения:

```
WebSocket server is listening on localhost:8085
HTTP server is listening on localhost:8000
```

Откройте браузер Chrome и его инструменты разработчика по адресу http://localhost:8000. Вы увидите сообщение, как показано в верхнем левом углу рис. 10.10. Под вкладкой Network справа видны два запроса, отправленные к серверам, выполняющимся на localhost. Первый загружает файл simple-websocket-client.html через HTTP, а второй отправляется на сокет, открытый на порте 8085 нашего сервера.

В этом примере HTTP-протокол используется только для начальной загрузки HTML-файла. Затем клиент запрашивает обновление протокола на WebSocket (код статуса 101), и далее эта страница не будет использовать HTTP.

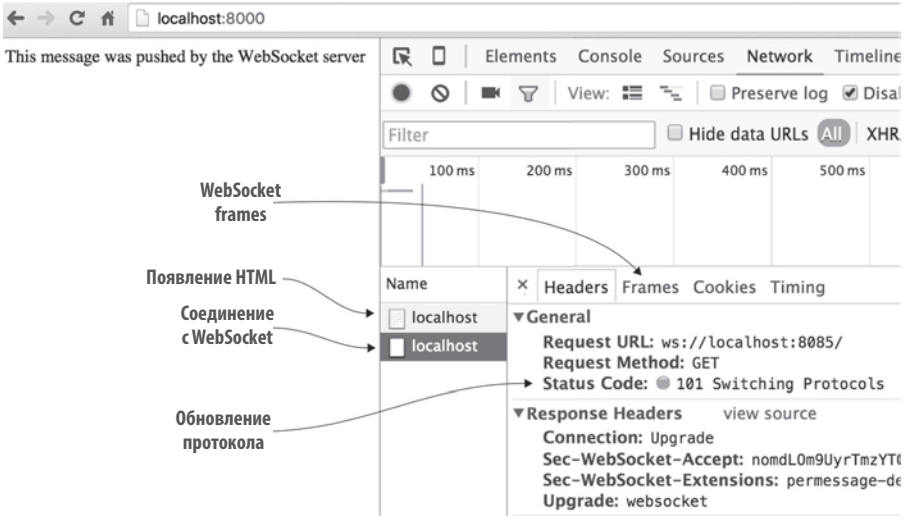


Рис. 10.10. Получение сообщения от сокета

Щелкните на вкладке Frames, и вы увидите содержимое сообщения, поступившего от этого сервера через сокет-соединение: «This message was pushed by the WebSocket server» (Это сообщение было передано WebSocket-сервером) (см. рис. 10.11).

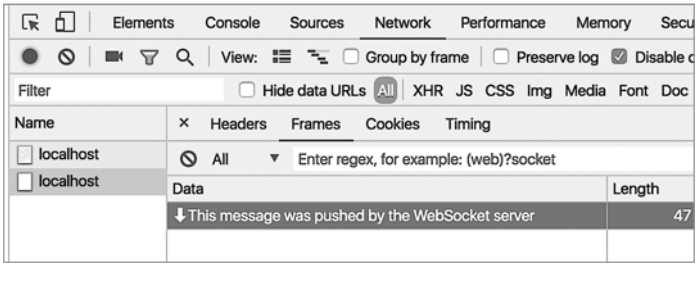


Рис. 10.11. Мониторинг содержимого фрейма

Обратите внимание на стрелку вниз рядом с сообщением во вкладке Frame. Ею обозначаются входящие сообщения сокета. Стрелка вверх, наоборот, обозначает сообщения, отправленные клиентом.

Для отправки сообщения от клиента серверу вызовите метод `send()` для браузерного объекта WebSocket:

```
ws.send("Hello from the client");
```

На деле перед отправкой сообщений вам следует всегда проверять статус соединения сокета, чтобы убедиться, что оно по-прежнему активно. Объект WebSocket содержит свойство `readyState`, которое может иметь одно из значений, показанных в табл. 10.1.

Таблица 10.1. Возможные значения WebSocket.readyState

Значение	Состояние	Описание
0	CONNECTING	Сокет был создан. Соединение еще не открыто
1	OPEN	Соединение открыто и готово к обмену
2	CLOSING	Соединение в процессе закрытия
3	CLOSED	Соединение закрыто или не может быть открыто

Вы увидите использование свойства `readyState` позже в коде сервера сообщений в листинге 10.17.

> **ПРИМЕЧАНИЕ** Если внимательно посмотреть на эту таблицу с ее ограниченным набором констант, рано или поздно на ум придут перечисления TypeScript, не так ли?

В следующем разделе мы разберем процесс добычи блока, и вы увидите, как в нем используется WebSocket-сервер.

10.6. РАССМОТРЕНИЕ ПРОЦЕССОВ УВЕДОМЛЕНИЯ

В этом разделе мы рассмотрим только те части кода, которые необходимы для понимания процесса взаимодействия сервера с клиентами блокчейна. Начнем мы с рассмотренного нами ранее сценария, описывающего взаимосвязь двух клиентов, но на этот раз оставим панель инструментов разработчика открытой, чтобы иметь возможность отслеживать сообщения, проходящие между клиентами и сервером через WebSocket-соединения.

Мы снова запустим сервер на порте 3000, используя команду `npm start`. Откройте первый клиент в браузере и подключитесь к localhost:3000, оставив открытыми вкладки Network > WS, как показано на рис. 10.12. Щелкните по имени localhost внизу слева, и вы увидите отправленные через сокет сообщения. Клиент подключается к серверу и делает запрос для нахождения длиннейшей цепочки, отправляя серверу сообщение типа `GET_LONGEST_CHAIN_REQUEST`.

Этот клиент оказался самым первым в блокчейне, и он получил от сервера сообщение с пустой `payload` (полезной нагрузкой), так как блокчейна еще не существует и в нем пока нет других узлов. Если бы другие узлы присутствовали,

сервер разослал бы по ним запрос, собрал ответы и отправил их запросившему длиннейшую цепочку узлу (клиенту).

ПРИМЕЧАНИЕ Слева в панели Frames стрелка вверх означает, что сообщение ушло к серверу. Стрелка, указывающая вниз, означает, что сообщение пришло от сервера.

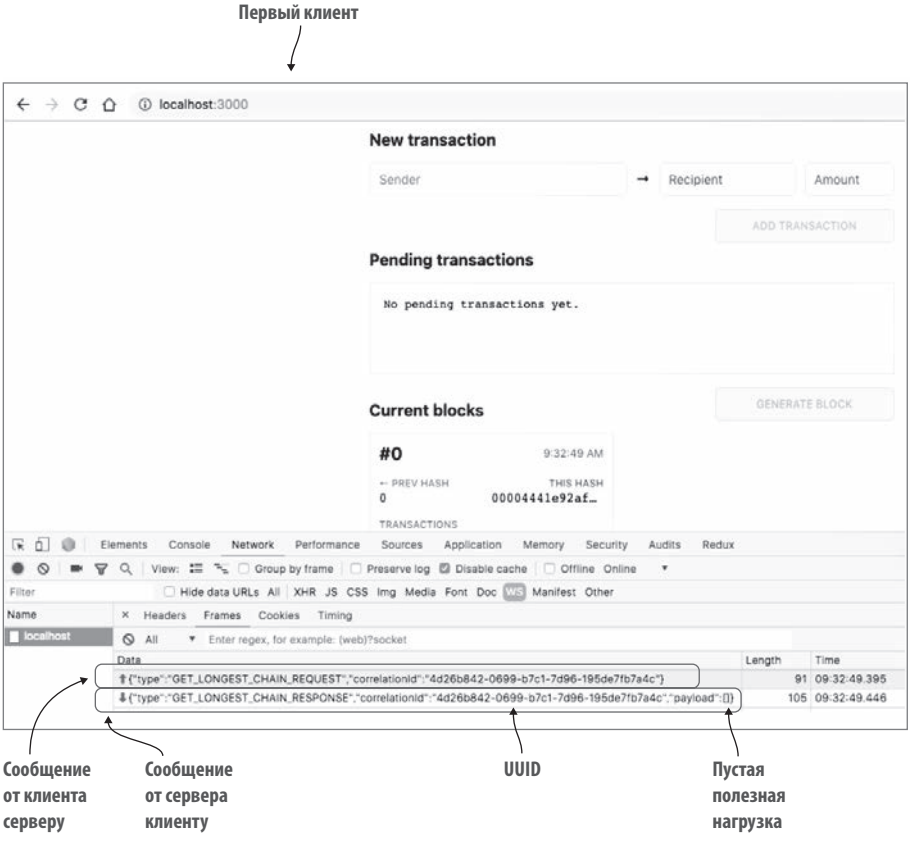


Рис. 10.12. Подключение первого клиента

Формат сообщения определен в файле shared/messages.ts, показанном в следующем листинге. Обратите внимание, что мы определяем пользовательские типы, используя ключевые слова `type`, `interface` и `enum`.

Листинг 10.11. shared/messages.ts: определение типов сообщений

```
export type UUID = string;   ◄—— Объявляет псевдоним типа для UUID

export interface Message {
```

```
    correlationId: UUID;  ◄——  Объявляет пользовательский тип Message
    type: string;
    payload?: any;  ◄——  payload опциональна
}

export enum MessageTypes {  ◄——  Объявляет перечисление с набором констант

    GetLongestChainRequest  = 'GET_LONGEST_CHAIN_REQUEST',
    GetLongestChainResponse = 'GET_LONGEST_CHAIN_RESPONSE',
    NewBlockRequest         = 'NEW_BLOCK_REQUEST',
    NewBlockAnnouncement    = 'NEW_BLOCK_ANNOUNCEMENT'
}
```

Сообщения отправляются асинхронно, и мы добавили свойство correlationId, чтобы иметь возможность сопоставлять исходящие и входящие сообщения. Если клиент отправит серверу сообщение `GET_LONGEST_CHAIN_REQUEST`, то оно будет содержать уникальный ID корреляции. Некоторое время спустя сервер отправляет сообщение `GET_LONGEST_CHAIN_RESPONSE`, которое будет также содержать ID корреляции. Сравнивая ID исходящего и входящего сообщений, мы можем найти совпадающие запросы и ответы. В качестве значений для `correlationId` мы используем повсеместно уникальные идентификаторы (UUID).

ГЕНЕРАЦИЯ UUID

Согласно спецификации RFC 4122 (www.ietf.org/rfc/rfc4122.txt), «UUID — это идентификатор, уникальный как в пространстве, так и во времени по отношению к пространству всех UUID. Поскольку UUID имеет фиксированный размер и содержит поле времени, значения могут быть пролонгированы (около 3400 г. н. э., в зависимости от используемого алгоритма)».

UUID — это строка фиксированной длины, состоящая из ASCII знаков и записываемая в следующем формате: `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx`. Ее пример вы можете видеть в сообщениях листинга 10.12. Мы позаимствовали этот код для генерации UUID со StackOverFlow (http://mng.bz/7z6e).

В нашем приложении все запросы инициируются клиентом, следовательно, UUID генерируются в коде клиента. Вы можете найти функцию `uuid()` в сценарии client/lib/cryptography.ts.

Так как UUID является строкой знаков, мы могли бы объявить для свойства `Message.correlationId` тип `string`. Но вместо этого мы создали псевдоним типа UUID при помощи ключевого слова `type` (см. листинг 10.11) и объявили для `correlationId` тип UUID, что повысило читаемость кода.

Когда клиент (узел блокчейна) начинает добычу, он отправляет сообщение `NEW_BLOCK_REQUEST`, приглашая другие узлы подключиться к процессу. Узел

заявляет о завершении добычи блока, отправляя другим узлам сообщение `NEW_BLOCK_ANNOUNCEMENT`, и его блок становится кандидатом на добавление в блокчейн.

10.6.1. Рассмотрение кода сервера

Мы запускаем сервер, загружая в среде Node.js сценарий server/main.ts (листинг 10.12). В этом сценарии мы импортируем фреймворк Express для настройки конечных точек HTTP и указания директории для развертывания кода веб-клиента. Мы также импортируем объект `http` из Node.js и пакет ws для поддержки WebSocket. Сценарий server/main.ts запускает два экземпляра сервера на одном порте: `httpServer`, поддерживающий HTTP, и `wsServer`, поддерживающий протокол WebSocket.

Листинг 10.12. server/main.ts: сценарий для запуска HTTP- и WebSocket-серверов

```
import * as express from 'express';
import * as http from 'http';
import * as path from 'path';
import * as WebSocket from 'ws';
import { BlockchainServer } from './blockchain-server';

const PORT = 3000;
const app = express();
app.use('/', express.static(path.join( dirname, '..', '..', 'public')));
app.use('/node_modules', express.static(path.join( dirname, '..', '..',
➥ 'node_modules')));

const httpServer: http.Server = app.listen(PORT, () => {
    if (process.env.NODE_ENV !== 'production') {
        console.log(`Listening on http://localhost:${PORT}`);
    }
});

const wsServer = new WebSocket.Server({ server: httpServer });
new BlockchainServer(wsServer);
```

Добавляет наш сценарий с поддержкой WebSocket

Импортирует класс BlockchainServer

Указывает расположение кода клиента

Инстанцирует Express

Указывает расположение node_modules, используемых клиентом

Запускает HTTP-сервер

Запускает WebServer

Запускает блокчейн-сервер уведомлений

Строки, начинающиеся с `app.use()`, отображают URL, поступающие от клиента, в конкретные ресурсы сервера. Это обсуждалось во врезке «Разрешение путей в Node.js» предыдущего раздела.

> **ПРИМЕЧАНИЕ** Фрагмент `'..', '..', 'public'` разрешится в `'../../public'` в Unix-системах и в `'..\..\public'` в Windows.

В процессе инстанцирования `WebSocket.Server` мы передаем ему экземпляр существующего HTTP-сервера. Это позволяет нам запустить на одном порте как HTTP-, так и WebSocket-сервер. В завершение мы инстанцируем TS-класс `BlockchainServer`, использующий WebSocket. Его код расположен в сценарии blockchain-server.ts. Класс `BlockchainServer` является подклассом `MessageServer`,

который заключает в себе всю работу, относящуюся ко взаимодействиям WebSocket. Позднее в этом же разделе мы рассмотрим его код.

В следующем листинге показана первая часть сценария blockchain-server.ts.

Листинг 10.13. Первая часть server/blockchain-server.ts

```
import * as WebSocket from 'ws';
import { Message, MessageTypes, UUID } from '../shared/messages';
import { MessageServer } from './message-server';

type Replies = Map<WebSocket, Message>;   <-- Ответы от узлов блокчейна

                                          Этот класс расширяет MessageServer
export class BlockchainServer extends MessageServer<Message> {
    private readonly receivedMessagesAwaitingResponse = new Map<UUID,
➥ WebSocket>();   <-- Коллекция клиентских сообщений, ожидающих ответов
    private readonly sentMessagesAwaitingReply = new Map<UUID, Replies>();
➥ // Used as accumulator for replies from clients.
                                                  Обработчик для всех
                                                    типов сообщений
    protected handleMessage(sender: WebSocket, message: Message): void {
        switch (message.type) {
            case MessageTypes.GetLongestChainRequest :
                return this.handleGetLongestChainRequest(sender, message);
            case MessageTypes.GetLongestChainResponse :
                return this.handleGetLongestChainResponse(sender, message);
            case MessageTypes.NewBlockRequest :
                return this.handleAddTransactionsRequest(sender, message);
            case MessageTypes.NewBlockAnnouncement :
                return this.handleNewBlockAnnouncement(sender, message);
            default : {
                console.log(`Received message of unknown type:
➥ "${message.type}"`);
            }
        }
    }

    private handleGetLongestChainRequest(requestor: WebSocket, message:
➥ Message): void {
        if (this.clientIsNotAlone) {
          this.receivedMessagesAwaitingResponse.set(message.correlationId,
➥ requestor);   <-- Сохраняет запрос клиента, используя в качестве ключа ID корреляции
          this.sentMessagesAwaitingReply.set(message.correlationId,
➥ new Map());   <-- Эта карта накапливает ответы от клиентов
          this.broadcastExcept(requestor, message);
        } else {                                          Рассылает сообщение
            this.replyTo(requestor, {                       другим узлам
              type: MessageTypes.GetLongestChainResponse,
              correlationId: message.correlationId,
              payload: []   <-- В блокчейне с одним узлом нет длиннейших цепочек
            });
      }
}
```

Вызывает подходящий обработчик, основываясь на типе сообщения

Метод `handleMessage()` служит в качестве диспетчера для сообщений, полученных от клиентов. У него есть инструкция `switch` для вызова подходящего обработчика на основе полученного сообщения. Например, если один из клиентов отправит сообщение `GetLongestChainRequest`, то будет вызван метод `handleLongestChainRequest()`. Сначала он сохраняет запрос (ссылку на открытый объект `WebSocket`) в карту, используя в качестве ключа ID корреляции. Затем он рассылает сообщение другим узлам, запрашивая их длиннейшие цепочки. Метод `handleLongestChainRequest()` может вернуть объект с пустой полезной нагрузкой, только если в блокчейне есть всего один узел. Метод `handleMessage()` был объявлен в суперклассе `MessageServer` как `abstract`, и мы это рассмотрим в следующем разделе.

> **ПРИМЕЧАНИЕ** Сигнатура метода `handleMessage()` включает ключевое слово `void`, которое означает, что он не возвращает значение. Так почему же его тело содержит ряд инструкций возврата? Как правило, каждый случай `case` в JS-инструкции `switch` должен оканчиваться на `break`, чтобы код не «проскакивал» к выполнению следующего `case`. Использование инструкций `return` в каждом `case` позволяет нам избежать инструкций `break` и по-прежнему гарантирует, что код не проскочит.

В листинге 10.14 показана вторая часть blockchain-server.ts. Она содержит код, обрабатывающий ответы от других узлов, отправляющих их длиннейшие цепочки.

Листинг 10.14. Вторая часть server/blockchain-server.ts

```
private handleGetLongestChainResponse(sender: WebSocket, message: Message):
➥ void {
    if (this.receivedMessagesAwaitingResponse.has(message.correlationId)) {
        const requestor =
➥ this.receivedMessagesAwaitingResponse.get(message.correlationId);
      if (this.everyoneReplied(sender, message)) {
          const allReplies =
➥ this.sentMessagesAwaitingReply.get(message.correlationId).values();
          const longestChain =
➥ Array.from(allReplies).reduce(this.selectTheLongestChain);
          this.replyTo(requestor, longestChain);
        }
    }
  }

  private handleAddTransactionsRequest(requestor: WebSocket, message:
➥ Message): void {
      this.broadcastExcept(requestor, message);
  }

  private handleNewBlockAnnouncement(requestor: WebSocket, message:
➥ Message): void {
```

Находит клиента, запросившего длиннейшую цепочку

Получает ссылку на объект сокета клиента

Находит длиннейшую цепочку

Передает длиннейшую цепочку клиенту, ее запросившему

```
        this.broadcastExcept(requestor, message);
    }

    private everyoneReplied(sender: WebSocket, message: Message): boolean {
        const repliedClients = this.sentMessagesAwaitingReply
            .get(message.correlationId)
            .set(sender, message);

        const awaitingForClients =
➥   Array.from(this.clients).filter(c => !repliedClients.has(c));

        return awaitingForClients.length === 1;
    }

    private selectTheLongestChain(currentlyLongest: Message,
            current: Message, index: number) {
        return index > 0 && current.payload.length >
➥   currentlyLongest.payload.length ?
            current : currentlyLongest;
    }

    private get clientIsNotAlone(): boolean {
        return this.clients.size > 1;
     }
}
```

Проверяет, все ли узлы ответили на запрос

Все ли узлы ответили?

Этот метод используется при сокращении массива длиннейших цепочек

Проверяет, превышает ли количество узлов блокчейна единицу

Когда узлы отправляют свои сообщения `GetLongestChainResponse`, сервер использует ID корреляции, чтобы найти клиента, запросившего длиннейшую цепочку. Когда все узлы ответили, метод `handleGetLongestChainResponse()` преобразует набор `allReplies` в массив и использует метод `reduce()` для нахождения длиннейшей цепочки. Затем при помощи метода `replyTo()` он отправляет ответ запрашивающему клиенту.

Все это хорошо, но где же код, поддерживающий протокол WebSocket и определяющий такие методы, как `replyTo()` и `broadcastExcept()`? Все эти механизмы расположены в абстрактном суперклассе `MessageServer`, показанном в листингах 10.15 и 10.16.

В процессе чтения кода `MessageServer` вы узнаете многие элементы синтаксиса TS, рассмотренные в части 1 этой книги. Во-первых, этот класс объявлен как `abstract` (см. раздел 3.1.5):

```
export abstract class MessageServer<T>
```

Вы не можете инстанцировать абстрактный класс, потребуется объявить подкласс, который обеспечит конкретную реализацию всех абстрактных членов. В нашем случае подкласс `BlockchainServer` реализует единственный абстрактный член — `handleMessage()`. Кроме того, объявление класса использует обобщенный тип `T` (см. раздел 4.2), который также используется в качестве типа аргумента

в методах `handleMessage()`, `broadcastExcept()` и `replyTo()`. В нашем приложении обобщенный тип `<T>` замещается конкретным типом `Message` (см. листинг 10.11), но в других приложениях это может быть другой тип.

Листинг 10.15. Первая часть server/message-server.ts

```
import * as WebSocket from 'ws';

export abstract class MessageServer<T> {

    constructor(private readonly wsServer: WebSocket.Server) {
        this.wsServer.on('connection', this.subscribeToMessages);
        this.wsServer.on('error', this.cleanupDeadClients);
    }

protected abstract handleMessage(sender: WebSocket, message: T): void;

protected readonly subscribeToMessages = (ws: WebSocket): void => {
    ws.on('message', (data: WebSocket.Data) => {
        if (typeof data === 'string') {
            this.handleMessage(ws, JSON.parse(data));
        } else {
            console.log('Received data of unsupported type.');
        }
    });
};

private readonly cleanupDeadClients = (): void => {
    this.wsServer.clients.forEach(client => {
        if (this.isDead(client)) {
            this.wsServer.clients.delete(client);
        }
    });
};
```

Подписка на сообщения от только что подключенного клиента

Очищает ссылки на отключившихся клиентов

Этот метод реализован в классе BlockchainServer

Получено сообщение от клиента

Передает сообщение обработчику

Производит очистку отключившихся клиентов

Объявляя метод `handleMessage()` как `abstract`, мы заявляем, что любой подкласс `MessageServer` может реализовывать этот метод по своему усмотрению до тех пор, пока сигнатура метода выглядит так:

```
protected abstract handleMessage(sender: WebSocket, message: T): void;
```

Так как мы применили эту сигнатуру метода в абстрактном классе, мы знаем, как должен быть вызван метод `handleMessage()` при реализации. Мы делаем это в `subscribeToMessages()` следующим образом:

```
this.handleMessage(ws, JSON.parse(data));
```

Строго говоря, мы не можем вызывать абстрактный метод, но в среде выполнения ключевое слово `this` будет относиться к экземпляру конкретного класса `BlockchainServer`, в котором метод `handleMessage()` уже не будет абстрактным.

В следующем листинге показана вторая часть класса `MessageServer`. Эти методы реализуют рассылку и ответ клиентам.

Листинг 10.16. Вторая часть server/message-server.ts

```
protected broadcastExcept(currentClient: WebSocket, message: Readonly<T>):
➥ void {  ◄── Производит рассылку по всем узлам
    this.wsServer.clients.forEach(client => {
        if (this.isAlive(client) && client !== currentClient) {
            client.send(JSON.stringify(message));
        }
    });
  }

  protected replyTo(client: WebSocket, message: Readonly<T>): void {
    client.send(JSON.stringify(message));
  }

  protected get clients(): Set<WebSocket> {
    return this.wsServer.clients;
  }

  private isAlive(client: WebSocket): boolean {
    return !this.isDead(client);
  }

  private isDead(client: WebSocket): boolean {
    return (
        client.readyState === WebSocket.CLOSING ||
        client.readyState === WebSocket.CLOSED
    );
  }
}
```

Отправляет сообщение одному узлу

Проверяет, не отключен ли конкретный клиент

В последней строке сценария server/main.ts (см. листинг 10.12) мы передали экземпляр WebSocket-сервера в конструктор класса `BlockchainServer`. Этот объект имеет свойство `clients`, являющееся коллекцией всех активных WebSocket-клиентов. Когда нам понадобится разослать сообщение всем клиентам мы переберем эту коллекцию как в методе `broadcastExcept()`. Если нам понадобится удалить ссылку на отключенного клиента, мы также используем свойство `clients` в методе `cleanupDeadClients()`.

Сигнатуры методов `broadcastExcept()` и `replyTo()` содержат аргумент отображенного типа `Readonly<T>`, который мы рассмотрели в разделе 5.2. Он получает тип `T` и помечает все его свойства как `readonly`. Мы используем `Readonly`, чтобы избежать случайного изменения значения внутри метода. Вы можете увидеть больше примеров во врезке «Примеры условных и отображенных типов» чуть позже в этой главе.

Продолжим рассмотрение рабочего процесса, начатого первым клиентом на рис. 10.12. Второй клиент присоединяется к блокчейну и отправляет сообщение

WebSocket-серверу, запрашивая длиннейшую цепочку, которой на данный момент является первичный блок. На рис. 10.13 показаны сообщения, переданные через WebSocket-соединения. Мы пронумеровали эти сообщения, чтобы было легче понять их последовательность:

1. Второй клиент (справа) подключается к серверу обмена сообщений, запрашивая длиннейшую цепочку. Сервер рассылает этот запрос другим клиентам.
2. Первый клиент (слева) получает запрос. Этот клиент имеет только первичный блок, который и является длиннейшей цепочкой.
3. Первый клиент отправляет обратно сообщение, содержащее в качестве полезной нагрузки длиннейшую цепочку.
4. Второй клиент получает длиннейшую цепочку в полезной нагрузке сообщения.

В этом примере у нас только два клиента, но WebSocket-сервер рассылает сообщение по всем подключенным клиентам, поэтому все они ответят.

Далее первый клиент создает две ожидающие транзакции, как показано на рис. 10.14. Это локальное событие, и никакие сообщения WebSocket-серверу не отправляются.

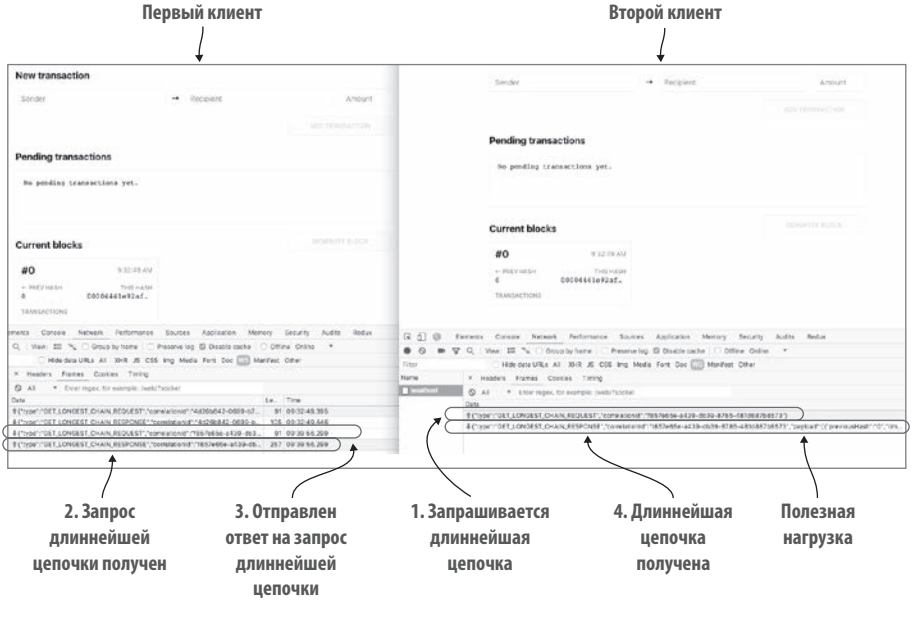


Рис. 10.13. Подключение второго клиента

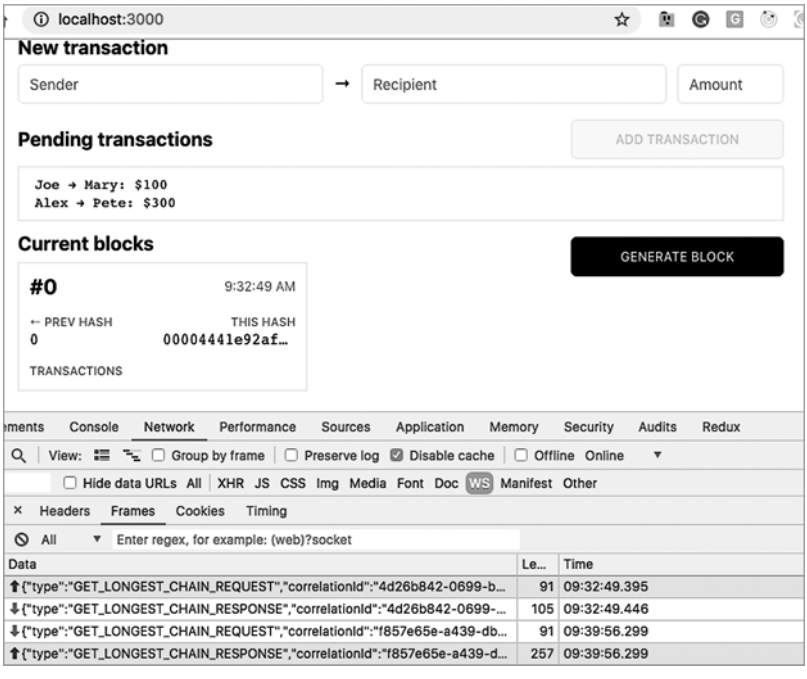


Рис. 10.14. При добавлении транзакций сообщения не отправляются

Теперь первый клиент щелкает по кнопке `GENERATE BLOCK`, запуская добычу блока и приглашая другие узлы присоединиться. Сообщение `NEW_BLOCK_REQUEST` является этим приглашением на добычу. Некоторое время спустя добыча завершается (в нашем случае первый блок закончил ее первым), и первый клиент анонсирует новый блок-претендент, отправляя сообщение `NEW_BLOCK_ANNOUNCEMENT`. На рис. 10.15 показаны сообщения, относящиеся к добыче блока, а также содержимое сообщения `NEW_BLOCK_ANNOUNCEMENT`.

Теперь оба клиента показывают два одинаковых блока: #0 и #1. Обратите внимание, что значение `hash` блока #0 и значение `previousHash` блока #1 совпадают.

В следующем листинге показано содержимое сообщения `NEW_BLOCK_ANNOUNCEMENT`. Мы сократили хеш-значения для улучшения читаемости.

Листинг 10.17. Сообщение, содержащее новый блок

```
correlationId: "2773c28b-aa55-e2ba-a6ec-75bb6b980d89"  ◄—— ID корреляции (UUID)
payload: {previousHash: "00004441e92af1",…}
    hash: "00001befe1b1e4df392e601..."  ◄—— Хеш-значение блока #1
    nonce: 51803  ◄—— Вычисленный nonce (доказательство работы)
    previousHash: "00004441e92a..."  ◄—— Хеш-значение блока #0
```

```
    timestamp: 1549207329217
    transactions: [{sender: "Joe", recipient: "Mary", amount: 100},
    {sender: "Alex", recipient: "Pete", amount: 300}]
type: "NEW_BLOCK_ANNOUNCEMENT"
```

Транзакции блока

Тип сообщения

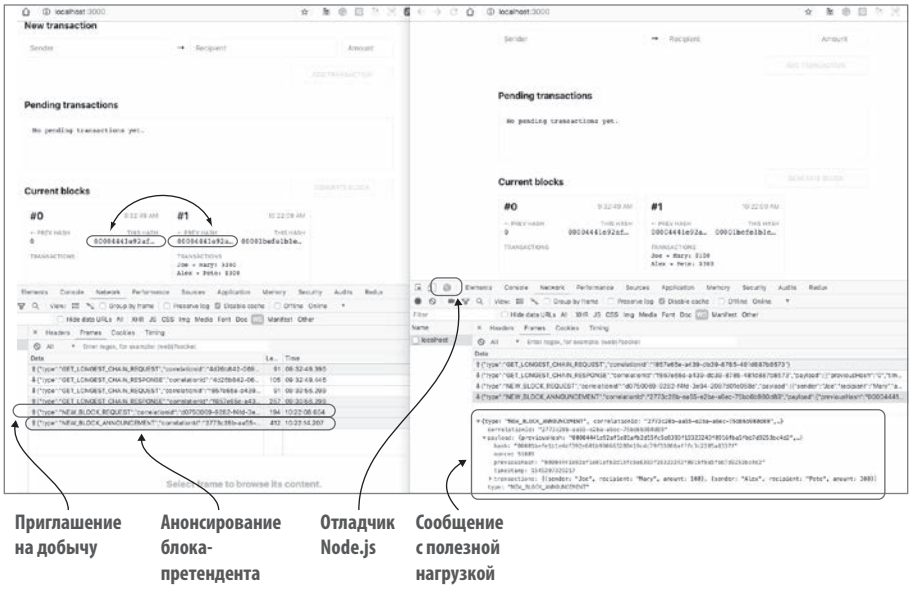


Рис. 10.15. Обмен сообщениями при добыче нового блока

Вернитесь к листингу 10.11, чтобы найти определения пользовательских типов данных, используемых в этом сообщении. Следующим вопросом будет: «Кто отправил сообщение с полезной нагрузкой, содержащей только что добытый блок?» Это сделал веб-клиент, и в следующем разделе мы рассмотрим его код, соответствующий этому процессу.

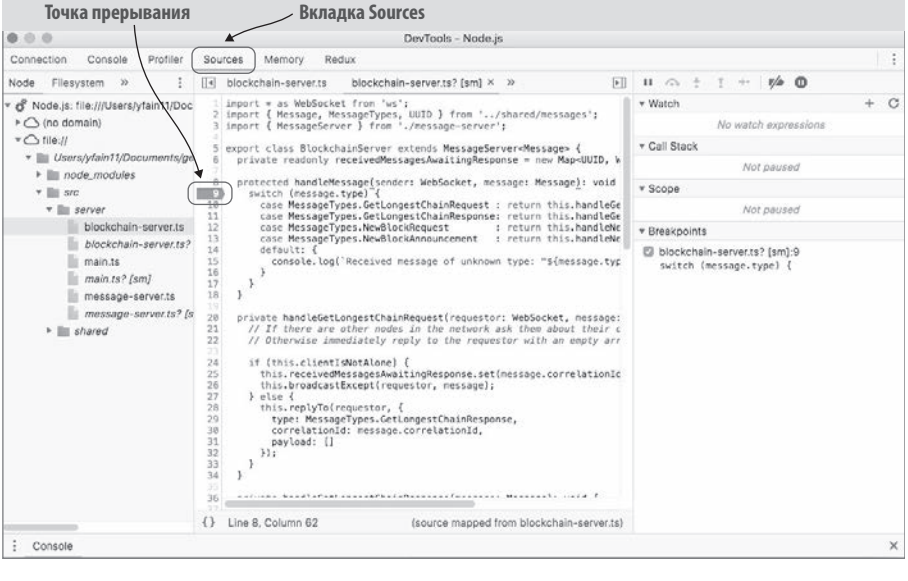
ОТЛАДКА NODE.JS КОДА В БРАУЗЕРЕ

Мы запустили Node.js с опцией `--inspect`, поэтому когда вы откроете инструменты разработчика Chrome, то слева во вкладке Elements увидите зеленый шестиугольник (см. рис. 10.15). Этот шестиугольник представляет отладчик Node.js — щелкните по нему, чтобы открыть инструменты разработчика в отдельном окне.

Node.js соединяется с инструментами разработчика через порт 9229, как вы видели в листинге 10.8, когда мы запускали сервер. По умолчанию окно Node.js показывает содержимое из вкладки Connection. Переключитесь на вкладку Sources, как показано на следующем рисунке, найдите TS-код, который хотите отладить,

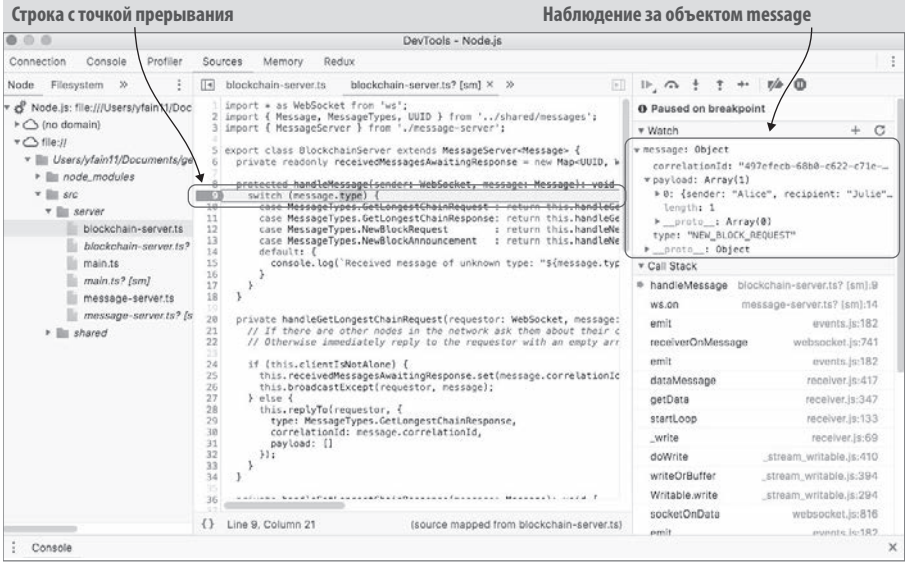
и добавьте точку останова. Далее показана точка останова, установленная на строке 9 сценария blockchain-server.ts

Открытие инструментов разработчика для Node.js



Выбор вкладки Sources

Мы хотим перехватить момент, когда клиент отправляет сообщение серверу. Следующий скриншот был сделан после того, как клиент создал транзакцию `Alice -> Julie -> $200` и щелкнул по кнопке `GENERATE BLOCK`.



Отладка Node.js в браузере Chrome

Процесс выполнения остановился на строке 9. Мы добавили переменную `message` в панель Watch справа вверху, чтобы производить ее отладку или вести за ней наблюдение. Для Node.js-кода доступны все возможности отладчика Chrome.

10.6.2. Рассмотрение кода клиента

Мы использовали слово «клиент», потому что в этом приложении мы обсуждаем использование взаимосвязей между веб-приложением клиента и сервером. Но более подходящим словом здесь было бы «узел». Код нашего узла реализован как веб-приложение, но в реальном блокчейне узлы, добывающие блоки, и UI, где пользователи могут добавлять транзакции, были бы отдельными приложениями.

ПРИМЕЧАНИЕ Главная тема этой главы — представление сервера обмена сообщениями. Мы не собираемся рассматривать каждую строчку кода, но покажем вам те его части, которые необходимы для понимания реализации клиента.

Код, выполняемый в браузере, расположен в директории src/client. Для HTML-отображения мы используем небольшую библиотеку под названием lit-html (www.npmjs.com/package/lit-html), которая позволяет писать HTML-шаблоны с помощью шаблонных литералов JavaScript. Эта библиотека использует размеченные шаблоны, являющиеся простыми функциями JS, совмещенными с HTML (как объясняется в разделе A.3.1 приложения). Коротко говоря, эта библиотека может принимать строку с HTML-разметкой, создавать узел DOM браузера и повторно отображать его при изменении данных.

Библиотека lit-html эффективно отображает содержимое строчных шаблонов в DOM, обновляя только те узлы, которые отвечают за представление обновленных значений. Листинг 10.18 взят из документации этой библиотеки (http://mng.bz/m4D4). Здесь она определяет размеченные шаблоны с выражением `${name}` и затем передает `"Steve"` в качестве имени для отображения: `<div> Hell Steve</div>`. Далее, когда значение переменной name изменяется на `"Kevin"`, она обновляет только имя в этом `<div>`.

Листинг 10.18. Как lit-html отображает HTML

```
import {html, render} from 'lit-html';

const helloTemplate = (name) => html`<div>Hello ${name}!</div>`;
render(helloTemplate('Steve'), document.body);
render(helloTemplate('Kevin'), document.body);
```

Объявляет размеченный шаблон

Импортирует функции html и render

Отображает приветствие Steve

Заменяет Steve на Kevin в

> **ПРИМЕЧАНИЕ** Суть lit-html в том, что она считывает HTML и один раз создает узел DOM, а после этого только отображает новые значения для вложенных переменных, когда они изменяются. Она не создает виртуальную DOM подобно React.js и не использует обнаружители изменений подобно Angular. Она просто обновляет значения переменных.

Когда вы будете читать сценарии в директории client/ui, вы увидите в них не HTML-файлы, а скорее вызовы функции `render()`, схожие с предыдущим примером. Если вы знакомы с библиотеками Angular или React, то представьте себе класс, имеющий в качестве UI-компонента функцию `render()`. Главный API библиотеки lit-html — это html, и вы увидите, что каждая функция `render()` его использует.

Рассмотрим небольшой класс `PendingTransactionsPanel`, отображающий панель с ожидающими транзакциями. В листинге 10.19 показан файл ui/pending-transactions-panel.ts, реализующий функцию `render()`.

Листинг 10.19. pending-transaction-panel.ts: панель Pending Transactions

```
import { html, TemplateResult }
➥ from '../../../node_modules/lit-html/lit-html.js';
import { BlockchainNode } from '../lib/blockchain-node.js';
```

```
import { Callback, formatTransactions, Renderable, UI } from './common.js';

export class PendingTransactionsPanel implements
➥ Renderable<Readonly<BlockchainNode>> {
                    Предоставляет конструктору функцию обратного вызова
    constructor(readonly requestRendering: Callback) {}

    render(node: Readonly<BlockchainNode>): TemplateResult {
        const shouldDisableGenerate = node.noPendingTransactions ||
                                      node.isMining;
        const formattedTransactions = node.hasPendingTransactions
            ? formatTransactions(node.pendingTransactions)
            : 'No pending transactions yet.';

        return html`
            <h2>Pending transactions</h2>
            <pre class="pending-transactions list">${formattedTransactions}</pre>
            <div class="pending-transactions form">${UI.button('GENERATE BLOCK',
➥ shouldDisableGenerate)}</form>
            <div class="clear"></div>
        `;
    }
}
```

Тип аргумента функции render() определен в blockchain-node.ts

lit-html нуждается в размеченном шаблоне html

Класс `PendingTransactionsPanel` реализует интерфейс `Renderable`, который требует наличия у него свойства `requestRendering` и метода `render()`, оба из которых определены в файле common.js следующим образом:

Листинг 10.20. Фрагмент из ui/common.ts

```
// сюда помещается другое определение типа
export type Callback = () => void;

export interface Renderable<T> {
    requestRendering: Callback;
    render(data: T): TemplateResult;
}
```

Определяет пользовательский тип функции обратного вызова

Определяет обобщенный интерфейс

Функция обратного вызова

Функция для отображения HTML с использованием lit-html

В коде клиента вы найдете много функций `render()`, но все они работают схожим образом: вставляют значения переменных JS в шаблоны HTML и обновляют соответствующую часть UI, когда значения этих переменных изменяются.

Клиентский класс высшего уровня называется `Application`, и он также реализует интерфейс `Renderable`, поэтому lit-html знает, как его отображать. Файл client/main.ts создает экземпляр класса `Application` и передает ему обратный вызов, который активирует его функцию `render()`.

Листинг 10.21. ui/main.ts: сценарий, создающий компонент высшего уровня

```
import { render } from '../../node_modules/lit-html/lit-html.js';
import { Application } from './ui/application.js';
```

```
let renderingIsInProgress = false;
let application = new Application(async () => {

    if (!renderingIsInProgress) {
        renderingIsInProgress = true;
        await 0;
        renderingIsInProgress = false;
        render(application.render(), document.body);
    }
});
```

Флаг для предотвращения двойного отображения

Передает обратный вызов экземпляру Application

Вызывает функцию render()

ПРИМЕЧАНИЕ Даже несмотря на то что функция `render()` получает в качестве аргумента `document.body`, библиотека lit-html не отображает повторно всю страницу, а делает это только для значений в изменившихся заместителях шаблонов.

Когда класс `Application` инстанцирован, он получает функцию обратного вызова (аргумент конструктора под названием `requestRendering`), которая вызывает функцию `render()`. Мы покажем вам два фрагмента класса `Application`, и там, где вы увидите `this.requestRendering()`, она активирует обратный вызов. Первый фрагмент сценария application.ts показан далее.

Листинг 10.22. Первый фрагмент ui/application.ts

```
export class Application implements Renderable<void> {
    private readonly node: BlockchainNode;
    private readonly server: WebsocketController;

    private readonly transactionForm = new
➥ TransactionForm(this.requestRendering);
    private readonly pendingTransactionsPanel =
➥ new PendingTransactionsPanel(this.requestRendering);
    private readonly blocksPanel = new BlocksPanel(this.requestRendering);

    constructor(readonly requestRendering: Callback) {

    this.server = new WebsocketController(this.handleServerMessages);
    this.node = new BlockchainNode();

    this.requestRendering();
    this.initializeBlockchain();
}

private async initializeBlockchain() {
    const blocks = await this.server.requestLongestChain();
    if (blocks.length > 0) {
        this.node.initializeWith(blocks);
    } else {
        await this.node.initializeWithGenesisBlock();
    }
```

Этот объект отвечает за обмен данными через WebSocket

Передает обратный вызов верхнего уровня каждому UI-компоненту

Ссылка на обратный вызов будет храниться в свойстве requestRendering

Подключается к WebSocket-серверу

Вся логика создания блокчейна и узлов расположена здесь

Инициализирует блокчейн

Запрашивает у всех узлов длиннейшую цепочку

```
    this.requestRendering();
}

render(): TemplateResult {  ◄── Отображает компоненты UI
    return html`
        <main>
            <h1>Blockchain node</h1>
            <aside>${this.statusLine}</aside>
            <section>${this.transactionForm.render(this.node)}</section>
            <section>
                <form @submit="${this.generateBlock}">
                    ${this.pendingTransactionsPanel.render(this.node)}
                </form>
            </section>
            <section>${this.blocksPanel.render(this.node.chain)}</section>
        </main>
    `;                                          Повторно отображает дочерний компонент
}
```

Начальное отображение инициируется из конструктора посредством вызова `this.requestRendering()`. Спустя несколько секунд выполняется второе отображение из метода `initializeBlockchain()`. Дочерние компоненты UI получают ссылку на обратный вызов `requestRendering` и могут решать, когда обновлять UI.

Метод `initializeBlockchain()` вызывается после начального отображения UI и запрашивает у WebSocket-сервера длиннейшую цепочку. Если этот узел не первый и не единственный, длиннейшая цепочка возвращается и отображается в панели блока в нижней части экрана. В противном случае генерируется и отображается первичный блок. Операции `requestLongestChain()` и `initializeWithGenesisBlock()` являются асинхронными, и код ожидает их завершения, используя ключевое слово `await`.

В листинге 10.23 показаны два метода из application.ts. Метод `handleServerMessages()` вызывается, когда WebSocket-сервер отправляет сообщение клиенту. Посредством инструкции `switch` он вызывает обработчик, соответствующий типу сообщения. А метод `handleGetLongestChainRequest()` вызывается, когда клиент получает запрос на отправку длиннейшей цепочки.

Листинг 10.23. Второй фрагмент ui/application.ts

```
                                Обрабатывает сообщения от WebSocket-сервера
private readonly handleServerMessages = (message: Message) => {
    switch (message.type) {  ◄── Передает сообщение соответствующему обработчику
        case MessageTypes.GetLongestChainRequest: return
                            this.handleGetLongestChainRequest(message);
        case MessageTypes.NewBlockRequest :
➥ return this.handleNewBlockRequest(message);
        case MessageTypes.NewBlockAnnouncement : return
➥ this.handleNewBlockAnnouncement(message);
```

```
        default: {
            console.log(`Received message of unknown type: "${message.type}"`);
        }
    }
}

private handleGetLongestChainRequest(message: Message): void {
    this.server.send({
        type: MessageTypes.GetLongestChainResponse,
        correlationId: message.correlationId,
        payload: this.node.chain
    });
```

Узел отправляет свою цепочку серверу

ПРИМЕРЫ УСЛОВНЫХ И ОТОБРАЖЕННЫХ ТИПОВ

В главе 5 мы представили вам условные и отображенные типы. Здесь вы увидите, как мы используем их в блокчейн-приложении. Не будем рассматривать сценарий client/lib/blockchain-node.ts целиком. Вы можете подробнее прочитать об обобщенных типах и отображенном типе `Pick` в документации TypeScript по ссылке http://mng.bz/5AJa.

До версии 3.5 TS не включал тип `Omit`, и нам приходилось объявлять его как пользовательский. С недавних пор этот тип был добавлен и стал встроенным, но мы решили сохранить пример кода этой сноски в целях наглядной демонстрации. Давайте обсудим пользовательские типы `Omit`, `WithoutHash` и `NotMinedBlock`, представленные в следующем коде:

```
export interface Block {
    readonly hash: string;
    readonly nonce: number;
    readonly previousHash: string;
    readonly timestamp: number;
    readonly transactions: Transaction[];
}

export type Omit<T, K> = Pick<T, Exclude<keyof T, K>>;
export type WithoutHash<T> = Omit<T, 'hash'>;
export type NotMinedBlock = Omit<Block, 'hash' | 'nonce'>;
```

Объявляет тип Block

Вспомогательный тип, использующий Pick

Объявляет тип, аналогичный Block, но без хеша

Объявляет тип, аналогичный Block, но без хеша и перца

Тип `Omit` в следующей строке позволяет нам объявить тип, который будет иметь все свойства типа `T`, за исключением `hash`:

```
type WithoutHash<T> = Omit<T, 'hash'>;
```

Процесс генерации хеш-значений блока занимает время, и пока у нас его нет, мы можем использовать тип `WithoutHash<Block>`, который не будет иметь свойства `hash`.

Для исключения можно указать более одного свойства:

```
type NotMinedBlock = Omit<Block, 'hash' | 'nonce'>;
```

Вы могли бы использовать тип так:

```
let myBlock: NotMinedBlock;

myBlock = {
    previousHash: '123',
    transactions: ["Mary paid Pete $100"]
};
```

Типом `NotMinedBlock` всегда будет `Block` минус `hash` и `nonce`. Если в какой-то момент кто-нибудь добавит в интерфейс `Block` другое необходимое свойство `readonly`, присвоение переменной `myBlock` не скомпилируется. Вместо этого компилятор будет жаловаться на то, что добавленное свойство должно быть инициализировано. В аналогичном сценарии в JavaScript мы бы получили ошибку среды выполнения.

Интерфейс определяет пользовательский тип `Block` с пятью обязательными свойствами, являющимися `readonly`, что означает возможность их инициализации только во время инстанцирования блока. Но создание блока требует времени, и не все значения для этих свойств будут доступны в процессе инстанцирования. Мы, конечно, хотим получить пользу от строгой типизации, но также нужна свобода инициализировать некоторые свойства после создания экземпляра `Block`.

В TypeScript есть отображенный тип `Pick` и условный тип `Exclude`, которые мы можем использовать для определения нового типа, исключающего некоторые из свойств существующего. `Exclude` перечисляет остающиеся свойства, а `Pick` позволяет вам создать тип на основе списка из этих оставшихся свойств.

Следующая строка означает, что мы хотим объявить обобщенный тип `Omit`, который может получить тип `T` и ключ `K`, а затем исключить из `T` свойства, совпадающие с `K`:

```
type Omit<T, K> = Pick<T, Exclude<keyof T, K>>;
```

Обобщенный тип `Omit` может использоваться с любым типом `T`, а `keyof T` возвращает список свойств конкретного типа. Например, если в качестве типа `T` мы передадим `Block`, то конструкция `keyof T` представит список свойств, определенных в интерфейсе `Block`. Использование `Exclude<keyof T, K>` позволяет нам удалить из списка некоторые свойства, а `Pick` создает новый тип из этого обновленного списка свойств.

В листингах 10.22 и 10.23 показана большая часть файла application.ts, где клиент обменивается данными с WebSocket-сервером через класс `WebSocketController`,

который получает обратный вызов для обработки сообщений через его конструктор. Давайте познакомимся с кодом класса `WebSocketController`, рассмотрев рабочий поток, когда пользователь щелкает по кнопке `GENERATE BLOCK`. Это должно привести к следующим действиям:

1. Рассылка сообщения `GET_LONGEST_CHAIN_REQUEST` через WebSocket-сервер.
2. Рассылка сообщения `NEW_BLOCK_REQUEST` другим узлам через WebSocket-сервер.
3. Добыча блока.
4. Обработка всех сообщений `GET_LONGEST_CHAIN_RESPONSE`, полученных от других узлов.
5. Рассылка сообщения `NEW_BLOCK_ANNOUNCEMENT` другим узлам и локальному сохранению блока-претендента.

В листинге 10.22 метод `render()` содержит форму, которая выглядит так:

```
<form @submit="${this.generateBlock}">
    ${this.pendingTransactionsPanel.render(this.node)}
</form>
```

UI-элемент с ожидающими транзакциями и кнопка `GENERATE BLOCK` реализованы в классе `PendingTransactionsPanel`. Мы используем директиву `@submit` из lit-html, и когда пользователь щелкает по кнопке `GENERATE BLOCK`, происходит вызов асинхронного метода `generateBlock()`.

Листинг 10.24. Метод generateBlock() класса Application

```
private readonly generateBlock = async (event: Event): Promise<void> => {
    event.preventDefault();

    this.server.requestNewBlock(this.node.pendingTransactions);
    const miningProcessIsDone =
➥ this.node.mineBlockWith(this.node.pendingTransactions);

    this.requestRendering();

    const newBlock = await miningProcessIsDone;
    this.addBlock(newBlock);
};
```

Предотвращает обновление страницы

Сообщает всем остальным узлам, что один из них начал добычу

Начинает добычу блока

Обновляет статус UI

Ожидает завершения добычи

Добавляет блок в локальный блокчейн

Когда мы заявляем, что начали добычу блока, то передаем список ожидающих транзакций `this.node.pendingTransactions`, чтобы другие узлы могли принять участие и попытаться добыть блок для этих же транзакций быстрее. Затем этот узел также приступает к добыче.

КОГДА НОВЫЙ БЛОК ОТВЕРГНУТ

Метод `Application.addBlock()` вызывает метод `addBlock()` в классе `BlockchainNode`. Мы уже рассматривали процесс добычи блока в главах 8 и 9, но хотелось бы отдельно подчеркнуть код, отвергающий попытку добавления нового блока:

```
const previousBlockIndex = this._chain.findIndex(b => b.hash ===
➥ newBlock.previousHash);
    if (previousBlockIndex < 0) {
        throw new Error(`${errorMessagePrefix} - there is no block in the
➥ chain with the specified previous hash "${newBlock.previousHash
➥ .substr(0, 8)}".`);
    }

const tail = this._chain.slice(previousBlockIndex + 1);
if (tail.length >= 1) {
    throw new Error(`${errorMessagePrefix} - the longer tail of the
➥ current node takes precedence over the new block.`);
}
```

Содержит ли новый блок существующий previousHash?

Есть ли уже в цепочке хотя бы один дополнительный блок?

Этот код относится к провалившейся попытке добавления нового блока, показанной в правом нижнем углу рис. 10.9. Сначала мы проверяем, существует ли в блокчейне значение `previousHash` нового блока. Оно может существовать, но не в последнем блоке.

Вторая инструкция `if` проверяет, есть ли хотя бы один блок после того, который содержит этот `previousHash`. Это могло бы означать, что в цепочку уже был добавлен минимум один новый блок (полученный от других узлов). В этом случае длиннейшая цепочка имеет приоритет, и только что сгенерированный блок отвергается.

Теперь давайте рассмотрим код, относящийся к обмену данными WebSocket. Если вы прочитаете код метода `addBlock()`, то увидите в нем следующие строки:

```
this.server.announceNewBlock(block);
```

Так узел просит WebSocket-сервер анонсировать свежедобытый блок-претендент. Класс `Application` также имеет метод `handleServerMessages()`, который обрабатывает сообщения, приходящие от сервера. Он реализован в файле client/lib/websocket-controller.ts, объявляющем интерфейс `PromiseExecutor` и класс `WebsocketController`.

Класс `Application` создает экземпляр `WebsocketController`, являющийся нашим единственным контактом для всех связей с сервером. Когда клиенту нужно отправить сообщение серверу, он использует такие методы, как `send()` или `requestLongestChain()`, но иногда сообщения будет отправлять сервер

клиентам. Именно поэтому мы передаем в конструктор `WebsocketController` метод обратного вызова.

Листинг 10.25. websocket-controller.ts: класс WebsocketController

```
export class WebsocketController {
    private websocket: Promise<WebSocket>;
    private readonly messagesAwaitingReply = new Map<UUID,
➥ PromiseExecutor<Message>>();  <-- Карта WebSocket-клиентов, ожидающих ответов

    constructor(private readonly messagesCallback: (messages: Message) =>
➥ void) {  <-- Передает конструктору обратный вызов
        this.websocket = this.connect();  <-- Подключается к WebServer
    }

    private connect(): Promise<WebSocket> {
        return new Promise((resolve, reject) => {
            const ws = new WebSocket(this.url);
            ws.addEventListener('open', () => resolve(ws));
            ws.addEventListener('error', err => reject(err));
            ws.addEventListener('message', this.onMessageReceived);
        });
    }

    private readonly onMessageReceived = (event: MessageEvent) => {
        const message = JSON.parse(event.data) as Message;

        if (this.messagesAwaitingReply.has(message.correlationId)) {
            this.messagesAwaitingReply.get(message.correlationId).
                                        resolve(message);
            this.messagesAwaitingReply.delete(message.correlationId);
        } else {
            this.messagesCallback(message);
        }
    }

    async send(message: Partial<Message>, awaitForReply: boolean = false):
➥ Promise<Message> {
        return new Promise<Message>(async (resolve, reject) => {
            if (awaitForReply) {
                this.messagesAwaitingReply.set(message.correlationId, { resolve,
➥ reject });  <-- Хранит сообщения, нуждающиеся в ответе
            }
            this.websocket.then(
                ws => ws.send(JSON.stringify(message)),
                () => this.messagesAwaitingReply.delete(message.correlationId)
            );
        });
    }
```

Присваивает обратные вызовы сообщениям WebSocket

Обрабатывает входящие сообщения

ПРИМЕЧАНИЕ Тип `Partial` рассмотрен в главе 5.

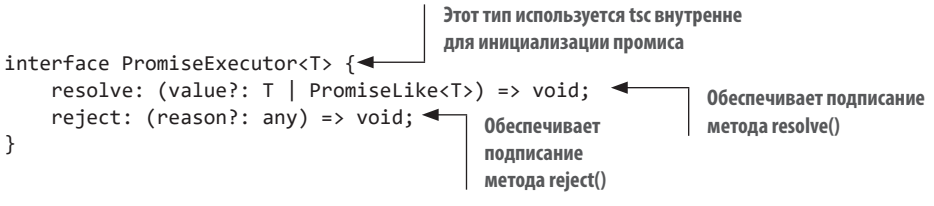
Метод `connect()` подключается к серверу и подписывается на стандартные сообщения WebSocket. Эти действия обернуты в `Promise`, поэтому если этот метод делает возврат, то мы можем быть уверены, что WebSocket-соединение установлено и все обработчики присвоены. Дополнительная выгода в том, что теперь с этими асинхронными функциями мы можем использовать ключевые слова `async` и `await`.

Метод `onMessageReceived()` обрабатывает сообщения, поступающие от сервера, являясь, по сути, маршрутизатором событий. В этом методе мы десериализуем сообщение и проверяем его ID корреляции. Если входящее сообщение является ответом на другое сообщение, то следующий код вернет `true`:

```
messagesAwaitingReply.has(message.correlationId)
```

Каждый раз, когда клиент отправляет сообщение, мы сохраняем его ID корреляции, отображенный в `PromiseExecutor`, в `messagesAwaitingReply`. `PromiseExecutor` знает, какой клиент ожидает ответа.

Листинг 10.26. Интерфейс PromiseExecutor из websocket-controller.ts



Для создания `Promise` интерфейс `PromiseConstructor` (см. объявление в lib.es2015.promise.d.ts) использует тип `PromiseLike`. В TypeScript есть ряд типов, оканчивающихся на `Like` (вроде `ArrayLike`). Они определяют подтипы, которые имеют меньше свойств, чем оригинальный тип. Промисы могут иметь различные сигнатуры конструктора. Например, все из них имеют `then()`, но могут иметь или не иметь `catch()`. `PromiseLike` сообщает tsc: «Я не знаю, какая у этого реализация, но по меньшей мере оно допускает `then`».

Когда клиент отправляет сообщение, требующее ответа, он использует метод `send()`, показанный в листинге 10.25, который хранит ссылку на сообщения клиентов, используя ID корреляции и объект типа `PromiseExecutor`:

```
this.messagesAwaitingReply.set(message.correlationId, { resolve, reject });
```

Ответ асинхронен, и мы не знаем, когда он поступит. В подобных сценариях вызывающий элемент JavaScript может создать `Promise`, передавая обратные вызовы `resolve` и `reject`, как объясняется в разделе A.10.2 приложения. Именно поэтому метод `send()` обертывает код, отправляющий сообщение, в `Promise`, и мы храним ссылки на объект, содержащий `resolve` и `reject` наряду с ID корреляции в `messageAwaitingReply`.

Для сообщений, ожидающих ответов, нам нужен способ их передачи в момент поступления. Мы можем использовать `Promise`, который может быть разрешен (или отклонен), используя обратные вызовы, которые мы передаем в конструктор как `Promise(resolve, reject)`. Чтобы сохранить ссылки на эту пару функций `resolve/reject` до момента получения ответа, мы создаем объект `PromiseExecutor`, который содержит именно два этих свойства (`resolve/reject`), откладывая их на потом. Другими словами, `PromiseExecutor` — это просто контейнер для двух обратных вызовов.

Интерфейс `PromiseExecutor` просто описывает тип объекта, который мы храним в карте `messageAwaitingReply`. Когда приходит ответ, метод `onMessageReceived()` находит по ID корреляции объект `PromiseExecutor`, вызывает `resolve()` и удаляет это сообщение из карты:

```
this.messagesAwaitingReply.get(message.correlationId).resolve(message);
this.messagesAwaitingReply.delete(message.correlationId);
```

Теперь, когда вы понимаете, как работает метод `send()`, вы должны суметь понять вызывающий его код. Следующий листинг показывает, как клиент может запросить длиннейшую цепочку.

Листинг 10.27. Запрос длиннейшей цепочки в websocket-controller.ts

```
async requestLongestChain(): Promise<Block[]> {
    const reply = await this.send(  <-- Вызывает метод send() и ожидает ответа
        {  <-- Первый аргумент в объекте Message
            type: MessageTypes.GetLongestChainRequest,
            correlationId: uuid()
        }, true);  <-- true означает "ожидаю ответа"
    return reply.payload;  <-- Возвращает из ответа полезную нагрузку
}
```

Класс `WebsocketController` имеет несколько других методов, которые работают с другими типами сообщений. Они реализованы похожими на `requestLongestChain()`. В файле client/lib/websocket-controller.ts вы увидите полный код `WebsocketController`. Посмотреть приложение в действии можно, выполнив `npm install`, а затем `npm start`. Откройте в браузере пару окон и попробуйте добыть несколько блоков.

ИТОГИ

- В блокчейне блок, содержащий одну и ту же транзакцию, могут одновременно добывать многие узлы. При этом правило длиннейшей цепочки помогает определить среди них победителя и достичь консенсуса среди всех остальных узлов.

- Если вы хотите разработать приложение, где клиентская и серверная стороны написаны в TypeScript, создайте два отдельных основанных на узлах проекта. Каждый проект будет иметь свой файл package.json и все необходимые сценарии конфигурации.
- Если нужно установить такую связь между клиентом и сервером, при которой каждая сторона сможет инициировать обмен данными, рассмотрите использование протокола WebSocket. Протокол HTTP основан на запросах, а WebSocket нет, что делает его удачным выбором для организации отправки данных от сервера клиенту без запросов.
- Node.js и большинство других технологий, используемых в бэкенде, поддерживают протокол WebSocket, и вы можете реализовать Node.js-сервер в TypeScript.

11 Разработка приложений Angular с помощью TypeScript

В этой главе:

- ✓ Знакомство с фреймворком Angular.
- ✓ Генерирование, сборка и запуск на сервере веб-приложения, написанного с помощью Angular и TypeScript.
- ✓ Реализация зависимостей в Angular.

В октябре 2014-го команда разработчиков из Google рассматривала вариант создания нового языка, AtScript, который бы расширял TS и использовался для разработки нового фреймворка Angular 2. В частности, AtScript должен был поддерживать декораторы, которых тогда еще в TS не было.

Затем один из инженеров предложил организовать встречу с командой TypeScript из Microsoft, чтобы обсудить их мнение насчет добавления декораторов в сам TS. Ребята из Microsoft согласились, и фреймворк Angular был написан на TypeScript, который также стал рекомендованным языком для написания приложений Angular. Сегодня более миллиона разработчиков используют Angular с TS для разработки веб-приложений, что дало невероятный прирост популярности TS.

На момент написания книги Angular имел 56 000 звезд и 1000 участников на GitHub. Отдадим дань уважения этому фреймворку и посмотрим, как можно использовать TS для написания приложений Angular. Эта глава вкратце познакомит вас со фреймворком, в то время как глава 12 будет посвящена рассмотрению кода блокчейн-приложения, написанного на Angular.

Сегодня главными игроками на рынке разработки веб-приложений (кроме суперпопулярного JQuery) являются Angular и React.js, при этом их постепенно догоняет Vue.js. Angular — это фреймворк, в то время как React.js — библиотека, которая поистине хорошо делает одно — отображает UI в DOM браузера. Глава 13 поможет вам начать разрабатывать веб-приложения при помощи React.js и TS. В главе 14 мы будет использовать React для разработки еще одной версии блокчейн-клиента. Глава 15 рассматривает основы Vue.js, а в главе 16 мы напишем еще один вариант блокчейн-клиента уже в нем.

> **ПРИМЕЧАНИЕ** Сложно предоставить подробный разбор разработки на Angular и TS в рамках одной главы. Если вы всерьез заинтересованы изучением этого фреймворка, то прочитайте другую нашу книгу — «Angular Development with TypeScript, second edition (Manning, 2018)»[1].

Разница между фреймворком и библиотекой в том, что фреймворк вынуждает вас писать приложения определенным способом, а библиотека предлагает конкретные возможности, которые вы можете использовать по собственному усмотрению. В этом смысле Angular определенно относится именно к фреймворкам, или даже более того, он является своеобразной платформой (фреймворком, допускающим расширение), которая имеет все необходимое для разработки веб-приложений:

- Поддержку внедрения зависимостей.
- Библиотеку современных компонентов UI — Angular Material.
- Роутер для организации навигации пользователя в приложении.
- Модуль для обмена данными с HTTP-серверами.
- Средства для разделения приложения на развертываемые модули с безотложной или ленивой загрузкой.
- Продвинутую поддержку форм.
- Библиотеку реактивных расширений (RxJS) для обработки потоков данных.
- Веб-сервер разработки, поддерживающий перезагрузку кода в реальном времени.
- Инструменты оптимизации и связывания для развертывания.
- Интерфейс командной строки для быстрой генерации приложения, библиотеки или более мелких артефактов вроде компонентов, модулей, служб и т. п.

---

[1] *Файн Я., Моисеев А.* Angular и TypeScript. Сайтостроение для профессионалов. — СПб.: Питер, 2018. — 464 с.: ил.

Давайте создадим и запустим простое веб-приложение, которое позволит нам продемонстрировать некоторые из возможностей Angular.

11.1. ГЕНЕРАЦИЯ И ЗАПУСК НОВОГО ПРИЛОЖЕНИЯ С ПОМОЩЬЮ ANGULAR CLI

CLI означает интерфейс командной строки, и Angular CLI является инструментом, который может сгенерировать и настроить новый проект Angular меньше чем за минуту. Для его установки на компьютер выполните следующую команду:

```
npm install @angular/cli -g
```

Теперь вы можете запускать CLI из терминала, используя команду `ng`, сопровождаемую параметрами. Эта команда может использоваться для генерации нового рабочего пространства, приложения, библиотеки, компонента, службы и др. Параметры для команды `ng` описаны в документации к Angular CLI (https://angular.io/cli). Вы также можете увидеть все, что доступно, выполнив `ng help` в терминале. Для получения помощи относительно конкретного параметра выполните `ng help`, сопровождаемую именем интересующего параметра, например `ng help new`.

> **ПРИМЕЧАНИЕ** В этой главе мы использовали Angular CLI 7.3. Чтобы увидеть, какая версия установлена у вас, выполните `ng version`.

Эта глава сопровождается четырьмя образцами проектов, которые были сгенерированы в Angular CLI. Несмотря на то что эти проекты готовы для обзора и выполнения, опишем процесс генерации и запуска проекта hello-world, чтобы вы могли попробовать осуществить его самостоятельно.

Для генерации нового минималистичного проекта выполните команду `ng new`, сопровождаемую его именем (например, `workspace`). Чтобы создать простой проект hello-world, выполните в терминале следующую команду:

```
ng new hello-world --minimal
```

Вам будет задан вопрос: `Would you like to add Angular routing? (y/N)` (Желаете ли вы добавить маршрутизацию Angular?). Ответьте `N`. Затем понадобится выбрать формат таблиц стилей. Чтобы по умолчанию использовать CSS, просто нажмите Ввод. Через секунду вы увидите имена сгенерированных файлов в новой директории hello-world, одним из которых будет package.json. Еще через 30 секунд будет выполнена команда `run npm`, которая установит все необходимые зависимости. На рис. 11.1 показано, как может выглядеть окно терминала в момент готовности проекта.

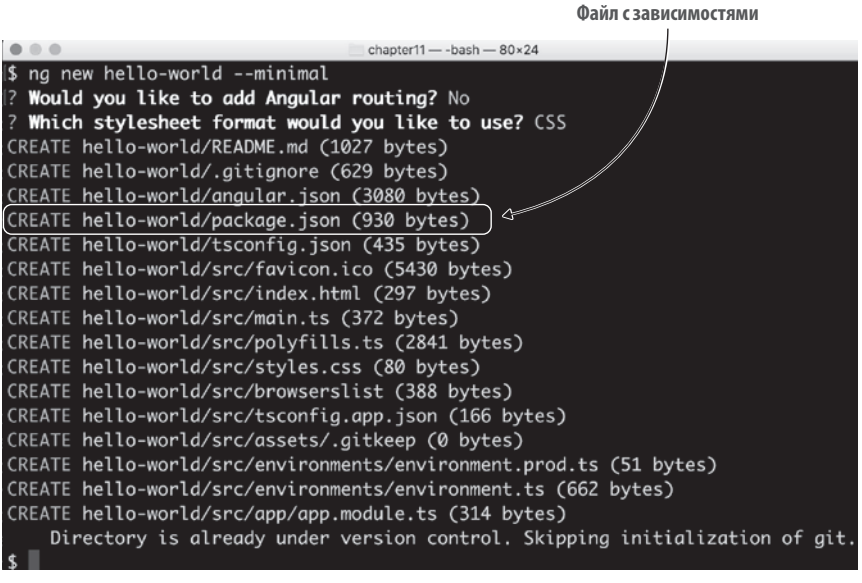


Рис. 11.1. Генерация минимального проекта

В терминале переключитесь на только что сгенерированную директорию hello-world и запустите приложение в браузере с помощью команды `ng serve -o` (`-o` означает: «Открыть браузер на хосте и на порте по умолчанию»):

```
cd hello-world
ng serve -o
```

Команда `ng serve -o` создает в памяти связки для этого приложения, запускает с ним веб-сервер и открывает браузер. На рис. 11.2 показан итоговый вывод консоли. На момент написания этого материала команда `ng serve` использует WebPack для связывания приложений и WebPack DevServer для запуска их на сервере. По умолчанию ваше приложение запускается по адресу localhost:4200.

Если вы читали главу 6, то узнаете связки WebPack и файлы карт кода. Код приложения расположен в части main.js, а код из Angular размещен в части под названием vendor.js. Не пугайтесь размера vendor.js (3.52 Мб), так как `ng serve` создает связки в памяти, не оптимизируя их. При запуске сборки в продакшене — `ng serve --prod` были бы произведены связки общим объемом чуть более 100 Кб.

Поздравляем! Ваше первое приложение Angular готово и запущено, и отображает страницу с сообщением «Welcome to hello-world!».

В момент написания эта страница выглядела так, как показано на рис. 11.3.

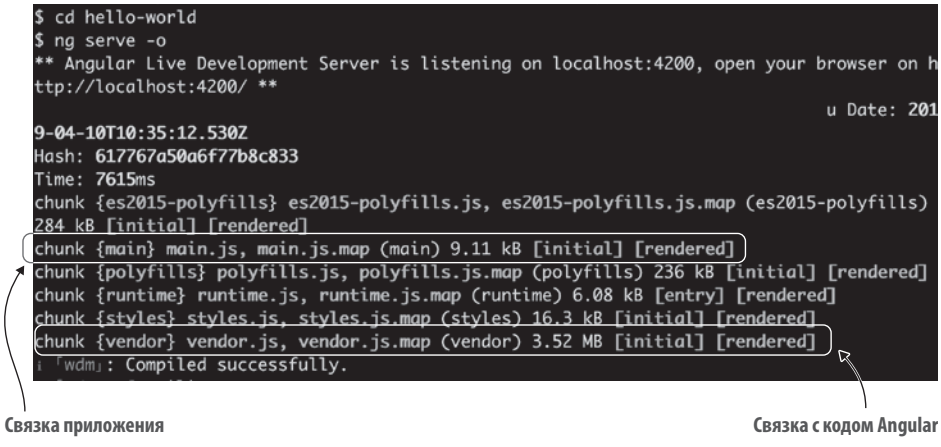


Рис. 11.2. Связки приложения созданы

Рис. 11.3. Выполнение hello-world

Несмотря на то что это не тот UI, который нужен вашему приложению, созданный проект содержит базовый TS-код — tsconfig.json, HTML-файл, отображающий этот UI, package.json со всеми необходимыми зависимостями, предварительно настроенный бандлер и некоторые другие файлы. Возможность получения связанного и рабочего приложения в течение всего пары минут, не затрачивая при этом времени на изучение Angular, весьма впечатляет. Но для разработки собственных приложений все-таки потребуется изучить этот фреймворк, а мы поможем вам сделать в этом направлении первые шаги.

Во-первых, закройте работающее приложение, нажав в терминале Ctrl-C. Затем откройте директорию hello-world в VS Code, который предлагает гораздо более удобную среду разработки, нежели терминал и простой текстовый редактор.

11.2. РАССМОТРЕНИЕ СГЕНЕРИРОВАННОГО ПРИЛОЖЕНИЯ

На рис. 11.4 показан скриншот из VS Code, демонстрирующий структуру сгенерированного проекта hello-world. Все эти файлы были созданы Angular CLI. Мы не станем затрагивать каждый, но опишем те из них, которые необходимы для понимания функционирования Angular-приложения и являются главными его составляющими. Эти файлы выделены на рис. 11.4 стрелочками.

Исходный код приложения расположен в директории src. Как минимум он будет иметь один компонент (app.component.ts) и один модуль (app.module.ts). Содержимое app.component.ts показано в следующем листинге.

Листинг 11.1. app.component.ts: компонент высшего уровня

```
import { Component } from '@angular/core';   <-- Импортирует декоратор Component

@Component({   <-- Декорирует класс с помощью @Component
    selector: 'app-root',                    <-- В других шаблонах вы можете ссылаться
    template: `   <-- Шаблон (UI) этого компонента     на этот компонент как на <app-root>
        <div style="text-align:center">
            <h1>
                Welcome to {{title}}!   <-- Привязывает значение свойства title класса
            </h1>
            <img width="300" src="...">
        </div>
        <h2>Here are some links to help you start: </h2>
        <!-- We removed the ul tag due to book space constraints -->
    `,
    styles: []   <-- Здесь размещается CSS
})
export class AppComponent {   <-- Декорированный класс component
    title = 'hello-world';   <-- Объявляет и инициализирует свойство title класса
}
```

> **ПРИМЕЧАНИЕ** Мы удалили ссылки, показанные на рис. 11.3, из HTML-части кода, чтобы сделать важную часть кода более наглядной.

Компонент Angular — это класс, декорированный при помощи `@Component()` (мы рассматривали декораторы в разделе 5.1). UI любого компонента объявляется в объекте, который мы передаем в декоратор `@Componet()`, а именно в его свойствах `selector`, `template`, `styles` или других.

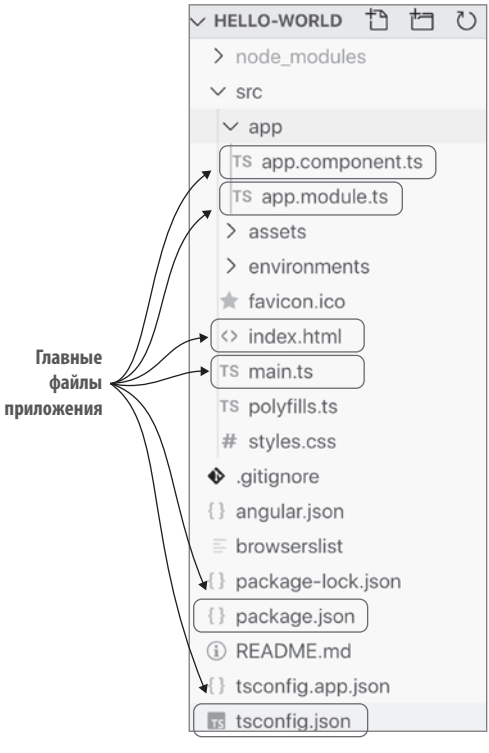


Рис. 11.4. Взгляд на рабочее пространство hello-world в VS Code

Свойство `selector` содержит значение, которое может использоваться в HTML-файлах, требующих наличия этого компонента. В листинге 11.1 CLI в качестве селектора для компонента приложения сгенерировал значение `app-root`, и если вы откроете файл index.html, то увидите раздел `<body>`, включающий открывающиеся и закрывающиеся теги, которые этому селектору соответствуют:

```
<body>
    <app-root></app-root>
</body>
```

Элементы UI любого компонента размещены в свойстве `template` декоратора `@Component()`. Обратите внимание на обратные кавычки, позволяющие создавать многострочные строки, что оказывается очень удобно при форматировании HTML.

ПРИМЕЧАНИЕ Angular позволяет вам отделять от TS-кода HTML. Если вы предпочитаете хранить HTML в отдельном файле, то вместо `template` используйте свойство `templateURL`. Например, `templateUrl: "app.component.html"`.

Взглянем на строку в листинге 11.1, которая гласит `Welcome to {{title}}!`. Двойные фигурные скобки представляют интерполяцию — вложение выражений в текст. Интерполяция также позволяет привязывать значение к строке. Привязывание используется для синхронизации значений членов TS класса с UI. Так откуда берется `title`? Это свойство класса `AppComponent`.

Значением свойства `title` является `hello-world`, что объясняет, почему UI на рис. 11.2 отобразил «Welcome to hello-world!». Измените значение этого свойства в процессе выполнения приложения, и UI сразу же будет обновлен.

> **ПРИМЕЧАНИЕ** Двойные фигурные скобки используются для привязки в шаблоне значений переменной к строке. С другой стороны, для привязки значений к свойству компонента используются квадратные скобки: `<CustomerComponent [name]=LastName>`. Здесь мы привязываем значение переменной `lastName` к свойству `name` класса `CustomerComponent`. Вы увидите больше примеров синтаксиса привязки свойств в листингах 11.26 и 11.32.

В листинге 11.1 свойство `styles` указывает на пустой массив. Если бы мы хотели добавить CSS-стили, то могли бы добавить их встроенными или указать имена одного или более CSS-файлов в свойстве `styleUrls`, например `styleUrls: [app.component.css]`.

Но объявления класса компонента недостаточно, поскольку ваше приложение должно иметь как минимум один модуль Angular (не стоит путать с модулями ECMAScript). Модуль Angular — это TS-класс, оформленный декоратором `@NgModule()`. Он подобен реестру компонентов, служб и, возможно, других взаимосвязанных модулей. Как правило, код класса пуст, а список членов модуля указан в свойствах декоратора.

Давайте взглянем на код, сгенерированный в файле app.module.ts.

Листинг 11.2. Файл src/app/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

@NgModule({
    declarations: [
        AppComponent
    ],
    imports: [
        BrowserModule
    ],
    providers: [],
```

Для веб-приложения необходим BrowserModule

Нам нужна реализация декоратора NgModule()

Это единственный компонент нашего приложения

Применяет декоратор NgModule()

Объявляет все компоненты, принадлежащие этому модулю

При необходимости импортирует другие модули

Объявляет поставщиков сервисов Angular при их наличии

```
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Указывает корневой компонент, который должен быть загружен

Декоратор `@NgModule()` требует от вас перечисления всех компонентов и модулей, используемых в приложении. В свойстве `declaration` у нас есть только один компонент, но если бы у нас их было несколько (например, `CustomerComponent` и `OrderComponent`), то нам бы пришлось их также перечислить:

```
declarations: [ AppComponent, CustomerComponent, OrderComponent ]
```

Конечно же, если вы упоминаете класс, интерфейс, переменную или имя функции, то их необходимо импортировать в верхнюю часть файла модуля, как мы делали для `AppComponent`. Кстати, вы заметили ключевое слово `export` в листинге 11.1? Если бы его там не было, то мы не смогли бы импортировать `AppComponent` в файл app.module.ts, как и в любой другой.

Свойство `imports` является местом для перечисления необходимых модулей. Сам Angular разбит на модули, и ваше реальное приложение, скорее всего, тоже будет на них разделено. В листинге 11.1 `imports` содержит только `BrowserModule`, который должен быть включен в корневой модуль любого приложения, работающего в браузере. В следующем листинге изображен другой пример использования свойства `imports`, показывающий, что еще может быть в него включено.

Листинг 11.3. Импортирование других модулей

```
imports: [ BrowserModule,      ← Модуль Angular для веб-приложений
           HttpClientModule,   ← Модуль Angular для совершения HTTP-вызовов
           FormsModule,        ← Модуль Angular для поддержки форм
           ShippingModule,     ← Модуль приложения, реализующий доставку
           BillingModule ]     ← Модуль приложения, реализующий биллинг
```

> **ПРИМЕЧАНИЕ** Кроме перечисления модулей в свойстве `imports` декоратора `@NgModule()`, нужно добавить инструкцию импорта ES6 для каждого из этих модулей, чтобы указать на файлы, где они реализованы.

В следующем разделе мы рассмотрим сервисы и внедрение зависимостей. Помимо этого, мы обсудим поставщиков сервисов, которые могут быть перечислены в свойстве `providers` модуля.

Свойство `bootstrap` называет компонент высшего уровня (корневой компонент), который модуль должен загружать первым. Корневой компонент может использовать дочерние, которые Angular распознает и тоже загрузит. Сами же эти дочерние компоненты могут иметь свои дочерние компоненты — их тоже Angular обнаружит и все загрузит. Модуль Angular в листинге 11.2 в разделе `declaration` имеет всего один компонент, поэтому он также перечислен в `bootstrap`, но если

бы компонентов было больше, то вам бы понадобилось указать один из них для инициализации при загрузке модуля.

Где же код, который изначально загружает модуль с его `AppComponent` (показанным ранее в листинге 11.1)? Он располагается в сгенерированном CLI-файле main.ts.

Листинг 11.4. Файл src/main.ts

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {   <-- Проверяет переменную среды продакшена
    enableProdMode();
}

platformBrowserDynamic()   <-- Создает точку входа в приложение
    .bootstrapModule(AppModule)   <-- Инициализирует корневой модуль
    .catch(err => console.error(err));   <-- Перехватывает возможные ошибки
```

Во-первых, код в main.ts считывает один из файлов в директории `environments`, чтобы проверить значение логической переменной `environment.production`. Не углубляясь в подробности, мы просто отметим, что это значение влияет на то, сколько раз детектор изменений Angular передаст дерево компонентов приложения, чтобы увидеть, что нужно обновить в UI. Детектор изменений мониторит каждую переменную, привязанную к UI, и передает движку отображения сигнал о том, что нужно обновить.

Во-вторых, API `platformBrowserDynamic()` создает *платформу*, являющуюся точкой входа в приложение. Затем он инициализирует модуль, который, в свою очередь, загружает корневой и все дочерние компоненты, необходимые для отображения этого модуля. Он также отобразит и другие модули, которые перечислены в свойстве `imports`, а также создаст инъектор, который знает, как внедрить сервисы, перечисленные в свойстве `providers` декоратора `@NgModule()`.

Хорошо, мы закончили с TS-кодом, создали связки, и вы могли подумать, что файл index.html, показанный на рис. 11.4, будет использовать связку, не так ли? А вот и нет. Такова была изначальная версия этого файла, когда он включал только селектор нашего корневого компонента, как можно увидеть в следующем листинге.

Листинг 11.5. src/index.html: HTML файл, загружающий приложение

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
```

```
    <title>HelloWorld</title>
    <base href="/">

    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
    <app-root></app-root>  ◄——  Корневой компонент приложения
</body>
</html>
```

В этом файле вы не увидите теги `<script>`, указывающие на JS-связки, показанные в листинге 11.2. Но когда вы запустите приложение при помощи команды `ng serve`, Angular добавит в него эти теги, и вы сможете увидеть их во время выполнения в инструментах разработчика Chrome, как показано на рис. 11.5.

ПРИМЕЧАНИЕ Команда `ng serve` собирает и пересобирает связки в памяти, чтобы ускорить процесс разработки. Но если вы хотите увидеть действительные файлы, выполните команду `ng build`, которая создаст файл index.html и все связки в директории dist.

ПРИМЕЧАНИЕ Файл angular.json, сгенерированный CLI, содержит все конфигурации проекта, и в нем вы можете изменять директорию вывода так же, как и многие другие опции.

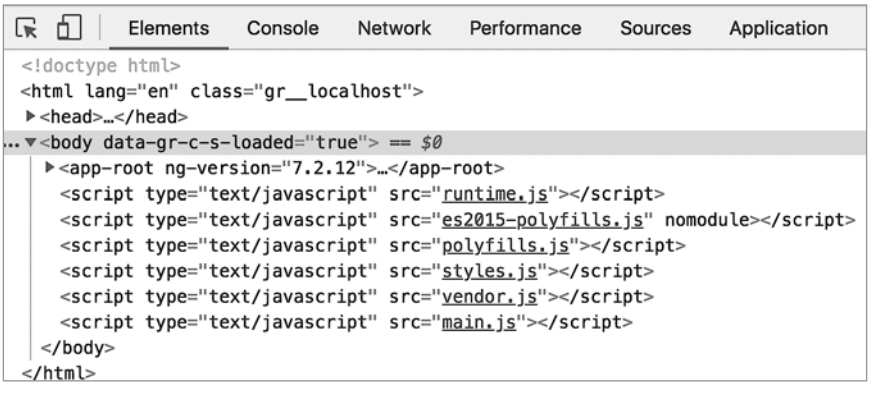


Рис. 11.5. Исполняемая версия HTML-документа имеет теги <script>

Это был высокоуровневый обзор проекта, сгенерированного CLI-командой `ng new hello-world --minimal`. Если бы мы не использовали опцию `--minimal`, CLI дополнительно сгенерировал бы некоторый шаблонный код для модульного и сквозного тестирования.

Вы можете использовать этот простой сгенерированный проект как основу для приложения этой главы, которое потребует больше компонентов и сервисов.

Angular CLI же призван помочь вам с генерацией шаблонного кода для различных артефактов приложения вроде компонентов и сервисов. Теперь узнаем, какую роль играют сервисы в Angular-приложении.

11.3. СЕРВИСЫ ANGULAR И ВНЕДРЕНИЕ ЗАВИСИМОСТЕЙ

Если компонент является классом с UI, тогда сервис будет просто классом, реализующим бизнес-логику вашего приложения. Вы можете создать один сервис, который вычисляет стоимость доставки, и другой, который будет инкапсулировать весь обмен HTTP-данными с сервером. Общее у этих сервисов то, что они не имеют UI. Angular же может инстанцировать и *внедрить* сервис в компоненты вашего приложения или в другой сервис.

ЧТО ТАКОЕ ВНЕДРЕНИЕ ЗАВИСИМОСТЕЙ

Если вам приходилось писать функцию, получающую в качестве аргумента объект, значит, вы уже писали программу, которая этот объект инстанцирует и внедряет в функцию.

Представьте пункт выдачи, отгружающий товары. Приложение, которое отслеживает отправленные товары, может создавать объект товара и вызывать функцию, создающую и сохраняющую запись о его отправке.

```
var product = new Product();
createShipment(product);
```

Функция `createShipment()` зависит от существования экземпляра объекта `Product`. Другими словами, у этой функции есть зависимость: `Product`. Но при этом сама функция не знает, как создавать `Product`. Вызывающий сценарий должен каким-то образом создать и передать (*внедрить*) этот объект в функцию в качестве аргумента.

Технически вы отделяете создание объекта `Product` от его использования. Но обе приведенные выше строки кода располагаются в одном сценарии, следовательно, это разделение условное, и если вам понадобится заместить `Product` на `MockProduct`, то в этом простом примере понадобится лишь небольшое изменение кода.

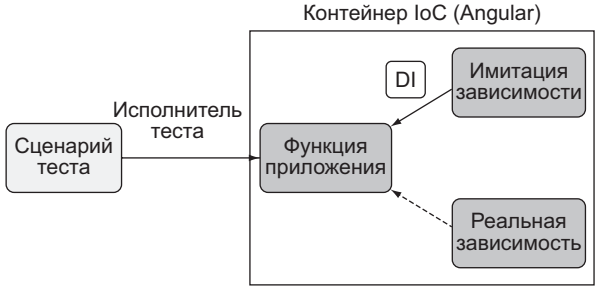
А что, если бы функция `createShipment()` имела три зависимости (например, товар, компанию-курьера и пункт отправки), каждая из которых имела бы еще и свои зависимости? В этом случае создание другого набора объектов для функции `createShipment()` потребовало бы внесения в код намного большего

числа изменений. А что, если бы можно было попросить кого-нибудь создать экземпляры зависимостей (с их собственными зависимостями) за вас?

Именно этим и занимается шаблон внедрения зависимостей (DI): если объект A зависит от объекта, идентифицированного токеном (уникальным ID) B, то объект A для инстанцирования объекта, на который указывает B, не будет явно использовать оператор `new`. Вместо этого он получит B внедрением из операционной среды.

Объект A просто должен объявить: «Мне нужен объект, известный как B. Не мог бы кто-нибудь его передать?» Объект A не запрашивает конкретный тип объекта (вроде `Product`), но скорее делегирует фреймворку ответственность за внедрение чего-либо в токен B. Выглядит это так, как будто объект A не хочет контролировать создание экземпляров и готов передать контроль этого процесса фреймворку. Здесь мы говорим о принципе *инверсии контроля* (IoC).

Еще одно удачное применение DI — это написание модульных тестов, где реальные сервисы нужно заменить на тестовые макеты. В Angular вы можете с легкостью настраивать, какие объекты (макеты или реальные) должны быть внедрены в тестовые сценарии, как показано на следующем изображении:



Внедрение макета сервиса в модульный тест

Фреймворк Angular реализует шаблон DI и предлагает вам простой способ замещения одного объекта другим.

Давайте попросим Angular сгенерировать класс `ProductService` в нашем проекте hello-world. Для этого используем команду CLI `ng generate`, которая может генерировать сервисы, компоненты, модули и т. д. Конкретно для генерации сервиса нужно использовать команду `ng generate service` (или `ng g s`), сопровождаемую именем сервиса. Следующая команда создаст файл product.service.ts в директории src/app:

```
ng generate service product --skip-tests
```

> **ПРИМЕЧАНИЕ** Чтобы увидеть все доступные аргументы и опции команды `ng generate`, выполните в терминале `ng help --generate`.

Если бы мы не указали опцию `--skip-tests`, то CLI дополнительно сгенерировал бы файл с шаблонным кодом для тестирования сервиса продуктов. В следующем листинге показано содержимое файла product.service.ts.

Листинг 11.6. Сгенерированный файл product.service.ts

```
import { Injectable } from '@angular/core';

@Injectable({                                  ← Этот декоратор отмечает класс как внедряемый
    providedIn: 'root'                         ← Экземпляр сервиса должен быть доступен
})                                               для всех членов корневого модуля
export class ProductService {

    constructor() { }
}
```

Декоратор `@injectable()` даст Angular команду дополнительно создать метаданные, необходимые для инстанцирования и внедрения сервиса. Свойство `provideIn` позволяет вам указывать, где этот сервис должен быть доступен. Значение `root` означает, что мы хотим, чтобы этот сервис был *предоставлен* на уровне приложения и являлся одиночкой, то есть все другие компоненты и сервисы будут использовать один экземпляр объекта `ProductService`. Если ваше приложение состоит из нескольких функциональных модулей, то вы можете ограничить доступность сервиса одним из них, например `provideIn: ShippingModule`.

В качестве альтернативы использованию свойства `provideIn` можно указать поставщика свойства `providers` в `@NgModule`:

```
@NgModule({
    ...
    providers: [ProductService]
})
```

Предположим, у вас есть `ProductComponent`, получающий детали товара при помощи класса `ProductService`. В отсутствие DI `ProductComponent` должен знать, *как* инстанцировать класс `ProductService`. Это может быть сделано несколькими способами вроде использования оператора `new`, вызова `getInstance()` для объекта-одиночки или активации фабричной функции `createProductService()`. В любом случае `ProductComponent` становится *плотно связанным* с `ProductService`, так как замещение `ProductService` на другую реализацию этого сервиса требует изменения кода в `ProductComponent`.

Если вы хотите повторно задействовать `ProductComponent` в другом приложении, использующем для получения описания товара другой сервис, то должны

изменить код, возможно, на `productService = new AnotherProductService()`. DI позволяет вам отделить компоненты приложения и сервисы, избавив их от необходимости знать, как создавать свои зависимости.

Документация Angular использует принцип *токена*, являющегося произвольным ключом, представляющим объект для внедрения. Вы отображаете токены в значения для DI, указывая *поставщиков*: поставщик — это инструкция для Angular, сообщающая, *как* создать экземпляр объекта для будущего внедрения в целевой компонент или другой сервис.

Мы уже упоминали, что поставщик может быть определен в объявлении модуля (если вам нужен сервис-одиночка), но вы также можете указать его на уровне компонента, как показано в листинге 11.7, где `ProductComponent` получает `ProductService` внедрением.

Angular инстанцирует и внедряет любой класс, использованный в аргументах конструктора. Если вы объявите поставщика в декораторе `@Component()`, то компонент получит свой собственный экземпляр сервиса, который будет уничтожен вместе с уничтожением компонента.

Листинг 11.7. ProductService, внедренный в ProductComponent

```
@Component({
    providers: [ProductService]
})
class ProductComponent {
    product: Product;

    constructor(productService: ProductService) {

        this.product = productService.getProduct();
    }
}
```

Определяет токен ProductService в качестве поставщика для внедрения

Внедряет объект, представленный токеном ProductService

Использует API внедренного объекта, предполагая, что getProduct() существует в сервисе

Зачастую имя токена совпадает с типом внедряемого объекта. Предыдущий фрагмент кода использует сокращение, `[ProductService]`, чтобы дать команду Angular предоставить токен `ProductService` посредством инстанцирования класса с тем же именем. Длинная версия выглядела бы так: `providers: [{provide: ProductService, useClass:ProductService}]`. Таким образом вы говорите Angular: «Если ты увидишь класс с конструктором, использующим токен `ProductService`, внедри экземпляр класса `ProductService`».

`ProductComponent` не требуется знать, какую конкретную реализацию типа `ProductService` использовать, — он использует любой объект, указанный в качестве поставщика. Ссылка на объект `ProductService` будет внедрена через аргумент конструктора и не будет необходимости явно инстанцировать `ProductService` в `ProductComponent`. Просто используйте его как в предыдущем

примере кода, который вызывает служебный метод `getProduct()` для экземпляра `ProductService`, магически созданного Angular.

Если вам нужно повторно использовать тот же `ProductComponent` с другой реализацией типа `ProductService`, измените строку поставщиков так: `providers: [{provide: ProductService, useClass: AnotherProductService}]`. Теперь Angular инстанцирует `AnotherProductService`, но код `ProductComponent`, использующий `ProductService`, не потребует изменения. В этом примере использование DI повышает повторное использование `ProductComponent` и устраняет его тесную связь с `ProductService`.

11.4. ПРИЛОЖЕНИЕ С ВНЕДРЕНИЕМ PRODUCTSERVICE

Эта глава включает в себя пример проекта di-products, который показывает, как вы можете внедрять (и использовать) `ProductService` в `ProductComponent`. В следующем листинге показан код `ProductService`.

Листинг 11.8. Файл product.service.ts из проекта di-products

```
import { Injectable } from '@angular/core';
import { Product } from './product';

@Injectable({
    providedIn: 'root'  ◄── Создает экземпляр-одиночку ProductService
})
export class ProductService {

    getProduct(): Product {  ◄── Возвращает жестко закодированные данные товара
        return { id: 0,
                 title: "iPhone XI",
                 price: 1049.99,
                 description: "The latest iPhone" };
    }
}
```

В этом сервисе у нас есть метод `getProduct()`, возвращающий жестко закодированные данные типа `Product`, показанного в следующем листинге. Однако в реальных приложениях для получения этих данных мы бы делали HTTP-запрос к серверу.

Листинг 11.9. Файл product.ts

```
export interface Product {
    id: number,
    title: string,
    price: number,
    description: string
}
```

ПРИМЕЧАНИЕ Мы объявили тип `Product` как интерфейс, что обеспечивает проверку типов `getProduct()`, но не оставляет следов в скомпилированном JS-коде. Мы могли объявить тип `Product` как класс, но это привело бы к генерации JS-кода (вроде класса или функции, в зависимости от значения опции компилятора target). Для объявления пользовательских типов лучше по возможности использовать интерфейсы, а не классы.

Теперь давайте взглянем на `component product`, который мы сгенерировали, используя следующую команду CLI:

```
ng generate component product --t --s --skip-tests
```

Опция `--t` указывает, что мы не хотим генерировать отдельный HTML-файл для шаблона компонента. Опция `--s` указывает, что вместо генерации отдельного CSS-файла мы будем использовать встроенные стили. В свою очередь, `--skip-tests` генерирует файл без шаблонного кода для модульного тестирования.

В следующем листинге показан `ProductComponent` после добавления шаблона и внедрения `ProductService`.

Листинг 11.10. product.component.ts: компонент товара

```
import {Component} from '@angular/core';
import {ProductService} from "../product.service";
import {Product} from "../product";

@Component({
    selector: 'di-product-page',
    template: `<div>
    <h1>Product Details</h1>
    <h2>Title: {{product.title}}</h2>  <-- Привязывает к UI title
    <h2>Description: {{product.description}}</h2>  <-- Привязывает к UI описание
    <h2>Price: \${{product.price}}</h2>  <-- Привязывает к UI цены
</div>`
})

export class ProductComponent {
    product: Product;  <-- Свойства этого объекта привязаны к UI

    constructor( productService: ProductService) {  <-- Внедряет ProductService

        this.product = productService.getProduct();  <-- Использует API ProductService
    }
}
```

Когда Angular инстанцирует `ProductComponent`, он также внедрит в этот компонент `ProductService`, так как он является аргументом конструктора. Затем конструктор вызовет `getProduct()`, свойство `product` компонента будет заполнено, и UI будет обновлен с использованием привязок.

ПРИМЕЧАНИЕ Просим вас не ругать нас за вызов `getProduct()` из конструктора компонента. В реальном проекте мы бы использовали для этого специальный обратный вызов `ngOnInit()`, но здесь мы хотели показать вам максимально простой код.

В завершение наш `AppComponent` определяет `ProductComponent` как потомка, используя в шаблоне его селектор.

Листинг 11.11. app.component.ts: компонент высшего уровня

```
import { Component } from '@angular/core';

@Component({
    selector: 'app-root',
    template: `<h1> Basic Dependency Injection Sample</h1>
               <di-product-page></di-product-page>`
})
export class AppComponent {}
```

Добавляет ProductComponent в шаблон

Соберите связку и запустите dev-сервер, используя команду `ng serve -o`, и браузер отобразит UI, как показано на рис. 11.6.

ПРИМЕЧАНИЕ В маловероятном случае, в котором вам не понравится UI нашего компонента, показанный на рис. 11.6, вы можете добавить в его декоратор `@Component()` свойство `styles` и использовать любые нужные вам стили.

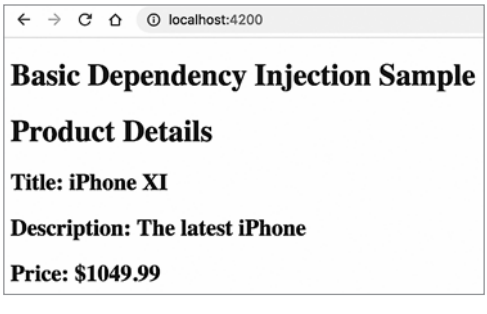


Рис. 11.6. Отображение данных товара

УПРАВЛЕНИЕ СОСТОЯНИЯМИ В ANGULAR

Управление состояниями является одной из важнейших составляющих любого приложения и само по себе заслуживает отдельной главы, но у нас места хватает лишь на врезку. Мы поместили ее в наше обсуждение DI, потому что в Angular DI широко используется для реализации управления состояниями.

В веб-приложении один компонент способен изменять значение одной или нескольких переменных, используемых в другом. Это может происходить вследствие действий пользователя или в результате получения новых данных, сгенерированных сервером. Например, вы находитесь на Facebook, и верхняя панель инструментов (один компонент) показывает, что у вас есть три непрочитанных сообщения. Когда вы щелкаете по цифре 3, открывается мессенджер (еще один компонент), который показывает эти непрочитанные сообщения.

Что-то помимо этих двух компонентов (панели инструментов и мессенджера) хранит и поддерживает счетчик сообщений в процессе сеанса пользователя. Это пример *управления состоянием*, которое играет очень важную роль в любом приложении.

Внедряемые сервисы Angular (совместно с RxJS) предлагают вам простой способ реализации управления состояниями. Если вы создадите сервис AppState и объявите его поставщика в декораторе `@NgModule()`, Angular создаст одиночку AppState, и вы сможете внедрить его в любые компоненты или сервисы, которым нужен доступ к текущему состоянию приложения.

Сервис AppState может иметь свойство `messageCounter`, а `MessangerComponent` может увеличивать его каждый раз, когда поступает новое сообщение. В итоге `ToolbarComponent` получал бы текущее значение `AppState.messageCounter` и отображал его в UI. Таким способом AppState становится *единственным источником истины* для хранения и предоставления значений состояния приложения. Более того, когда один компонент обновляет счетчик сообщений, AppState может разослать его новое значение другим компонентам, заинтересованным в получении информации о новом состоянии.

Несмотря на то что внедряемые сервисы предлагают чистое решение для управления состоянием, некоторые разработчики для реализации этого процесса в Angular предпочитают использовать сторонние библиотеки (вроде NGRX или NGXS). Эти библиотеки могут требовать от вас написания большого количества шаблонного кода, и вам следует подумать дважды, прежде чем принимать решение о реализации управления состоянием в вашем приложении. Если реализовать его плохо, то в итоге приложение может получиться склонным к ошибкам и затратным в обслуживании.

Яков Файн разместил видео «Angular: When ngrx is an overkill» (Angular: когда ngrx оказывается излишней), в котором сравнивает использование одиночного сервиса с использованием библиотеки NGRX для реализации управления состоянием в простом приложении. Посмотреть это видео вы можете на YouTube по адресу http://mng.bz/6w5A.

11.5. ПРОГРАММИРОВАНИЕ ЧЕРЕЗ АБСТРАКЦИИ В TYPESCRIPT

В разделе 3.2.3 мы рекомендовали вам программировать через интерфейсы (то есть абстракции). Поскольку Angular DI позволяет вам замещать внедряемые объекты, было бы неплохо объявить интерфейс `ProductSevice` и указать его в качестве поставщика. Точка внедрения выглядела бы как `constructor (productService: ProductService)`, далее вы бы написали несколько конкретных классов, реализующих этот интерфейс, и переключали их в объявлении поставщика по мере необходимости.

Вы могли бы сделать это в Java, C# и других объектно-ориентированных языках. В TypeScript же проблема в том, что после компиляции кода в JS интерфейсы удаляются, так как JS их не поддерживает. Другими словами, если бы `ProductService` был объявлен как интерфейс, конструктор `constructor (productService: ProductService)` превратился бы в `constructor (productService)` и Angular ничего бы о `ProductService` не узнал.

Но есть и хорошие новости: TS поддерживает абстрактные классы, которые могут иметь ряд реализованных методов и ряд абстрактных — тех, что были объявлены, но не реализованы (подробности в разделе 3.1.5). В этом случае вам понадобится реализовать конкретные классы, расширяющие абстрактные, а также реализовать все абстрактные методы. Например, вы могли бы использовать классы как в следующем листинге.

Листинг 11.12. Объявление абстрактного класса и двух потомков

```
export abstract class ProductService{   <-- Объявляет абстрактный класс
    abstract getProduct(): Product;     <-- Объявляет абстрактный метод
}

export class MockProductService extends ProductService{   <-- Создает первую конкретную реализацию абстрактного класса
    getProduct(): Product {
        return new Product('Samsung Galaxy S10');
    }
}

export class RealProductService extends ProductService{   <-- Создает вторую конкретную реализацию абстрактного класса
    getProduct(): Product {
        return new Product('iPhone XII');
    }
}
```

Хорошо также то, что вы можете использовать имя абстрактного класса в конструкторах, и в процессе генерации JS-кода Angular будет использовать определенный конкретный класс, исходя из объявления поставщика. Использование

классов ProductService, MockProductService и RealProductService, объявленных в листинге 11.12, позволит вам написать что-то вроде следующего:

Листинг 11.13. Использование абстрактного класса в роли поставщика

```
// Фрагмент из app.module.ts
@NgModule({
    providers: [{provide: ProductService, useClass: RealProductService}],
    ...
})
export class AppModule { }

// Фрагмент из product.component.ts
@Component({...})
export class ProductComponent {
    constructor(productService: ProductService) {...};
    }
    ...
}
```

Отображает конкретный тип в абстрактный токен

Использует абстрактный токен в качестве точки внедрения

Здесь мы используем абстракцию ProductService в объявлении поставщика и в качестве аргумента конструктора. Это не относится к листингу 11.8, в котором ProductService был конкретной реализацией определенной функциональности. Вы можете заместить поставщиков так же, как это описывалось ранее, или переключиться с одной конкретной реализации сервиса на другую.

Если бы вы не стали использовать абстрактные классы, то пришлось бы быть очень осторожными при объявлении классов ProductService и MockProductService, чтобы они имели в точности одинаковый API getProducts(). При использовании же подхода с абстрактными классами компилятор TS выдаст ошибку, если вы попытаетесь реализовать конкретный класс, но упустите реализацию одного из абстрактных методов. Программируйте через абстракции!

> **ПРИМЕЧАНИЕ** Есть и другой способ использовать DI и быть уверенными, что несколько классов имеют одинаковые API. В TS класс может реализовывать другой класс, например, так: `class MockProductService implements ProductService`. Этот синтаксис позволяет анализатору типов гарантировать реализацию классом MockProductService всех публичных методов, определенных в ProductService, и использовать любой из этих классов с DI.

11.6. НАЧАЛО РАБОТЫ С HTTP-ЗАПРОСАМИ

Приложения Angular могут обмениваться данными с любым веб-сервером, поддерживающим HTTP, и в этом разделе мы покажем, как можно начать делать HTTP-запросы. Это поможет вам понять код блокчейн-приложения, представленного в следующей главе.

Браузерные веб-приложения выполняют HTTP-запросы асинхронно, поэтому UI остается восприимчивым постоянно. Пользователь может продолжать работать с приложением в то время, как HTTP-запросы обрабатываются сервером. В Angular асинхронный HTTP реализуется с использованием специального *наблюдаемого* объекта, предлагаемого библиотекой RxJS, поставляемой вместе с Angular.

Если ваше приложение требует HTTP-сообщения, вам нужно добавить `HttpClientModule` в раздел imports декоратора `@NgModule()`. После этого вы можете использовать внедряемый сервис `HttpClient` для вызова `get()`, `post()`, `delete()` и других запросов, каждый из которых возвращает объект `Observable`.

В контексте клиент-серверного взаимодействия вы можете представить `Observable` как поток данных, которые могут быть переданы сервером вашему веб-приложению. Этот принцип проще понять, если использовать его в WebSocket-соединениях, где сервер передает данные в поток через открытый сокет постоянно. В случае же с HTTP вы всегда получаете обратно только один набор результатов, но можете воспринимать его как поток одного фрагмента данных.

> **ПРИМЕЧАНИЕ** Яков Файн опубликовал серию постов, посвященных RxJS и наблюдаемым потокам, которая доступна на его сайте по ссылке http://mng.bz/omBp.

Давайте посмотрим, как веб-клиент может сделать запрос к конечной точке сервера /product/123, чтобы извлечь `Product` с `ID 123`. В следующем листинге изображен один из способов вызова метода `get()` сервиса `HttpClient` с передачей URL в виде `string`.

Листинг 11.14. Создание запроса HTTP GET

```
interface Product {  ◄── Определяет тип Product
        id: number,
        title: string
}
...
class ProductService {
constructor(private httpClient: HttpClient) { }  ◄── Внедряет сервис HttpClient

ngOnInit() {  ◄── Этот метод обратного вызова активируется Angular
    this.httpClient.get<Product>('/product/123')  ◄── Объявляет запрос get()
        .subscribe(
            data => console.log(`id: ${data.id} title: ${data.title}`),
            (err: HttpErrorResponse) => console.log(`Got error: ${err}`)
        );                                          Выводит ошибку в случае ее возникновения
    }                                                Подписывается на результат get()
}
```

Сервис `HttpClient` внедряется в конструктор, и поскольку мы добавили квалификатор `private`, `httpClient` становится свойством объекта `ProductService`,

инстанцированного Angular. Мы поместили код, создающий HTTP-запрос, внутрь так называемого хук-метода `ngOnInit()`, который вызывается Angular, когда компонент инстанцирован и все его свойства инициализированы.

В методе `get()` мы не указывали полный URL (вроде http://localhost:8000/product/123). Мы предполагаем, что приложение Angular сделает запрос к тому же серверу, где оно развернуто, следовательно, основная часть URL может быть опущена. Обратите внимание, что в `get<Product>()` мы используем утверждение типа `<Product>` (эквивалент `as Product`), чтобы указать тип данных, ожидаемый в теле HTTP-ответа. Это утверждение типа сообщает статическому анализатору типов что-то вроде следующего: «Уважаемый TypeScript, тебе тяжеловато будет вывести тип данных, получаемых от сервера, поэтому давай я тебе помогу — им будет тип `Product`».

Возвращаемый результат всегда будет объектом `Observable`, который имеет метод `subscribe()`. В качестве его аргументов мы указали два обратных вызова:

- Первый будет вызван при получении данных, которые он отобразит в консоли.
- Второй же будет вызван, если запрос вернет ошибку.

Методы `post()`, `put()` и `delete()` используются схожим образом. Вы вызываете один из них и подписываетесь на его результаты.

> **ПРИМЕЧАНИЕ** Мы уже говорили, что каждый внедряемый сервис требует объявления поставщика, но поставщики для `HttpClient` объявляются внутри `HttpClientModule`, который включен в раздел `imports` декоратора `@NgModule`, следовательно, вам не нужно явно объявлять их в приложении.

По умолчанию `HttpClient` ожидает данные в формате JSON, которые затем автоматически преобразуются в JS-объекты. Если вы ожидаете данные в другом формате, используйте опцию `responseType`. Например, можно считать произвольный текст из файла, как показано в следующем листинге.

Листинг 11.15. Определение string в качестве типа возвращаемых данных

```
let someData: string;

this.httpClient
    .get<string>('/my_data_file.txt', {responseType: 'text'})
    .subscribe(
        data => someData = data,
        (err: HttpErrorResponse) => console.log(`Got error: ${err}`)
      );
```

Определение string в качестве типа для тела ответа

Присваивает полученные данные переменной

Выводит ошибку в случае ее появления

Теперь давайте посмотрим, как мы можем считать данные из файла JSON при помощи `HttpClient`. Эта глава содержит проект read-file, демонстрирующий использование `HttpClient.get()` для считывания файла, содержащего данные о товаре в формате JSON. Это приложение имеет директорию data, в которой содержится файл products.json, показанный в следующем листинге.

Листинг 11.16. Файл data/products.json

```
[
    { "id": 0, "title": "First Product", "price": 24.99 },
    { "id": 1, "title": "Second Product", "price": 64.99 },
    { "id": 2, "title": "Third Product", "price": 74.99}
]
```

Директория data содержит ресурсы проекта (файл products.json) и должна быть включена в связки этого проекта. Поэтому мы добавим ее в свойство приложения `assets` в файле angular.json.

Листинг 11.17. Фрагмент из angular.json

```
"assets": [
    "src/favicon.ico",
    "src/assets",
    "src/data"
]
```

Предустановленные ресурсы, сгенерированные Angular CLI

Имя директории ресурсов, добавленной нами в проект

Как правило, при использовании сервиса `HttpClient` вы будете указывать URL сервера. Но в нашем примере приложение URL будет указывать на локальный файл data/products.json. Наше приложение будет считывать этот файл и отображать товары, как показано на рис. 11.7.

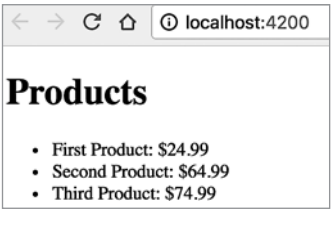


Рис. 11.7. Отображение содержимого products.json

`ApplicationComponent` будет использовать `HttpClient.get()` для формирования HTTP-запроса `GET`, и мы объявим интерфейс `Product`, определяя структуру ожидаемых данных о товаре.

Листинг 11.18. src/product.ts: пользовательский тип Product

```
export interface Product {
    id: string;
    title: string;
    price: number;
}
```

Файл app.component.ts показан в листинге 11.19, и именно в нем мы реализуем более простой способ подписки на ответы `HttpClient`. На этот раз вместо явного метода `subscribe()` мы используем фильтр `(pipe) async`.

ПРИМЕЧАНИЕ Фильтры в Angular — это специальные функции преобразования, которые могут использоваться в шаблоне компонента и представляются в виде вертикальной черты, сопровождаемой именем фильтра. Например, фильтр `currency` преобразует число в валюту (`123.5521 | currency` отобразит `$123.55` (по умолчанию валюта обозначается знаком доллара)).

Листинг 11.19. app.component.ts: компонент высшего уровня

```
import {HttpClient} from '@angular/common/http';
import {Observable} from 'rxjs';
import {Component, OnInit} from "@angular/core";
import {Product} from "./product";

@Component({
    selector: 'app-root',
    template: `<h1>Products</h1>
    <ul>
        <li *ngFor="let product of products$ | async">
            {{product.title }}: {{product.price | currency}}
        </li>
    </ul>
    `})

export class AppComponent implements OnInit{
    products$: Observable<Product[]>;

    constructor(private httpClient: HttpClient) {}

    ngOnInit() {
        this.products$ = this.httpClient
                         .get<Product[]>('/data/products.json');
    }
}
```

Импортирует Observable из библиотеки RxJS

Перебирает наблюдаемые товары и автоматически подписывается на них при помощи фильтра async

Отображает заголовок товара и его цену в формате валюты

Объявляет типизированный наблюдаемый объект для товаров

Внедряет сервис HttpClient

Создает HTTP-запрос GET, указывая тип ожидаемых данных

Наблюдаемый объект, возвращаемый методом `get()`, будет развернут в шаблон фильтром `async`, после чего заголовок и цена каждого продукта будут отображены следующим кодом:

```
<li *ngFor="let product of products$ | async">
    {{product.title }}: {{product.price | currency}} 1((CO17-1))
</li>
```

`*ngFor` является структурной директивой Angular, которая производит итерацию по каждой единице, отправленной наблюдаемым объектом `products$`, и отображает элемент `<li>`. Каждый элемент покажет заголовок и цену товара, используя привязку. Знак доллара в конце `products$` является просто условием для именования переменных, представляющих наблюдаемые объекты.

Чтобы увидеть это приложение в действии, выполните сначала команду `npm install` в директории client, а следом команду:

```
ng serve -o
```

11.7. НАЧАЛО РАБОТЫ С ФОРМАМИ

HTML предоставляет основные возможности для отображения форм, проверки введенных значений и отправки данных на сервер. Но HTML-формы могут быть недостаточно функциональны для реальных приложений, которым нужна возможность программной обработки введенных данных, применения пользовательских правил проверки, отображения понятных пользователю сообщений об ошибках, преобразования формата введенных данных и выбора способа отправки данных на сервер.

Angular предлагает для обработки форм два API:

- *Управляемый шаблоном* — при использовании этого API формы полностью программируются в шаблоне компонента при помощи директив, а объект модели создается Angular неявно. Но так как при определении формы вы ограничены HTML-синтаксисом, этот способ подходит только для простых форм.
- *Реактивный* — при использовании этого API вы явно создаете объект модели в коде TS, а затем связываете элементы HTML-шаблона со свойствами этой модели при помощи специальных директив. Для явного создания объекта формы модели вы используете классы `FormControl`, `FormGroup` и `FormArray`.

Для нестандартных форм реактивный подход является лучшим решением. В этом разделе мы вкратце познакомим вас с использованием реактивных форм, которые будут также использоваться в нашем блокчейн-приложении.

Для их активации вам нужно добавить `ReactiveFormsModule` из `@angular/forms` в список imports декоратора `@NgModule()` следующим образом:

Листинг 11.20. Добавление поддержки реактивных форм

```
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
    ...
    imports: [
        ...
        ReactiveFormsModule  ◄──  Импорт модуля поддержки реактивных форм
    ],
    ...
})
```

Теперь давайте взглянем на создание модели формы, представляющей структуру данных, служащую для их хранения. Ее можно построить из классов `FormControl`, `FormGroup` и `FormArray`. Например, следующий листинг объявляет свойство

класса типа FormGroup и инициализирует его с новым объектом, который будет содержать экземпляры элементов управления вашей формой.

Листинг 11.21. Создание объекта модели формы

```
myFormModel: FormGroup;

    constructor() {
        this.myFormModel = new FormGroup({   <-- Создает экземпляр модели формы
            username: new FormControl(''),   |  Добавляет в модель формы
            ssn: new FormControl('')         |  элементы ее управления
        });
    }
```

FormControl является элементарной единицей формы, которая обычно соответствует одному элементу <input>, но может также представлять и более сложные компоненты UI вроде календаря или слайдера. Экземпляр FormControl хранит текущее значение HTML-элемента, которому оно соответствует, статус валидности этого элемента и информацию о его изменениях.

Вот как вы можете создать элемент управления и передать его начальное значение в качестве первого аргумента конструктора:

```
city = new FormControl('New York');
```

Вы также можете создать FormControl и прикрепить один или более встроенных пользовательских валидаторов, которые могут быть присоединены к элементу управления формой или ко всей форме. Следующий листинг показывает, как вы можете добавить два встроенных валидатора в элемент управления формой.

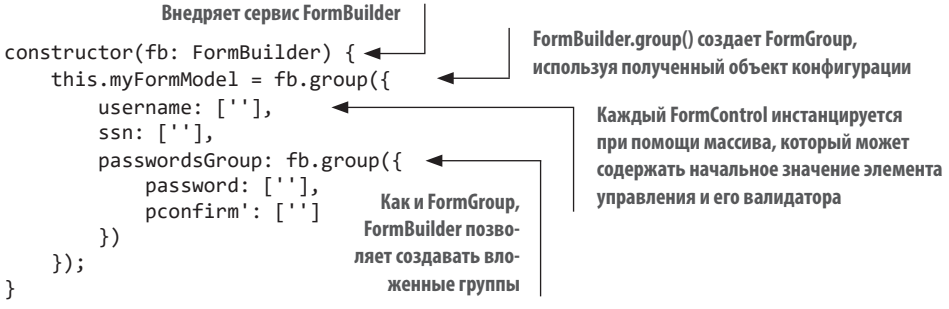
Листинг 11.22. Добавление валидаторов в элемент управления формой

```
                                          Создает элемент управления формой
                                          с начальным значением New York
city = new FormControl('New York',  <--   |
               [Validators.required,  <--  Добавляет необходимый валидатор
                Validators.minLength(2)]);  <--  Добавляет валидатор minLength
```

FormGroup является коллекцией объектов FormControl и представляет либо всю форму, либо ее часть. FormGroup объединяет значения и валидность каждого FormControl в группу. Если один из элементов управления в группе оказывается недействительным, вся группа также становится недействительной.

Внедряемый сервис FormBuilder является одним из способов создания моделей форм. Его API более лаконичен и избавляет вас от повторяющегося инстанцирования новых объектов FormControl как в листинге 11.21. В следующем листинге Angular внедряет объект FormBuilder, используемый для объявления модели формы.

Листинг 11.23. Создание formModel при помощи FormBuilder

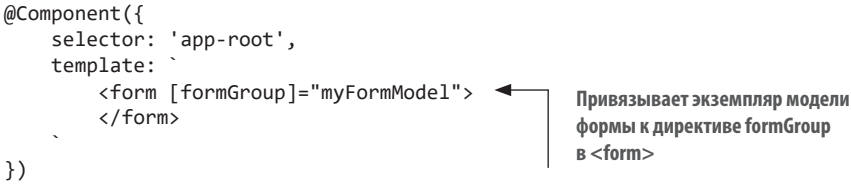



```
constructor(fb: FormBuilder) {
    this.myFormModel = fb.group({
        username: [''],
        ssn: [''],
        passwordsGroup: fb.group({
            password: [''],
            pconfirm': ['']
        })
    });
}
```

Метод `FormBuilder.group()` в качестве последнего аргумента принимает объект с дополнительными параметрами конфигурации. При необходимости вы можете использовать его для определения валидаторов уровня группы.

Реактивный подход требует использовать в шаблонах компонентов директивы. К этим директивам добавляется префикс `form`, например `formGroup` (со строчной `f`), как показано в следующем листинге.

Листинг 11.24. Привязка FormGroup к тегу HTML-формы




```
@Component({
    selector: 'app-root',
    template: `
        <form [formGroup]="myFormModel">
        </form>
    `
})
class AppComponent {
    myFormModel = new FormGroup({
                    username: new FormControl(''),
                    ssn: new FormControl('')
                  });
}
```

Реактивные директивы `formGroup` и `formControl` привязывают элементы DOM вроде `<form>` и `<input>` к объекту модели (вроде `myFormModel`) при помощи синтаксиса привязки свойств с квадратными скобками:

```
<form [formGroup]="myFormModel">
    ...
</form>
```

Директивами, связывающими элементы DOM со свойствами модели TS по имени, являются `formGroupName`, `formControlName` и `formArrayName`. Они могут использоваться только внутри HTML-элемента, отмеченного директивой `formGroup`.

Директива `formGroup` привязывает экземпляр класса `formGroup`, представляющего модель всей формы, к элементу DOM формы высшего уровня, как правило, `<form>`. В шаблоне компонента используйте `formGroup` (со строчной `f`), а в TypeScript создайте экземпляр класса `FormGroup` (с прописной `F`).

Директива `formControlName` должна использоваться в области директивы `formGroup`. Она связывает отдельный экземпляр `FormControl` с элементом DOM. Давайте продолжим добавление кода в пример модели `dateRange` из предыдущего раздела. Компонент и модель формы остаются прежними. Для завершения шаблона вам нужно лишь добавить HTML-элементы с директивой `formControlName`.

Листинг 11.25. Завершенный шаблон формы

```
<form [formGroup]="myFormModel">
    <div formGroupName="dateRange">
        <input type="date" formControlName="from">
        <input type="date" formControlName="to">
    </div>
</form>
```

from — это имя свойства во вложенной группе модели dateRange

to — это имя свойства во вложенной группе модели dateRange

Как и в директиве `formGroupName`, вы указываете имя того `FormControl`, который хотите связать с элементом DOM. Опять же, это будут имена, которые вы выбрали в процессе определения модели формы.

Директива `formControl` используется с отдельными элементами управления формой или формами с единичным элементом управления. Это удобно, когда вы не хотите создавать модель формы при помощи `formGroup`, но при этом хотите использовать возможности API Forms вроде валидации и реактивного поведения, обеспечиваемого свойством `FormControl.valueChanges`, имеющим тип `Observable` (это означает, что вы можете подписаться на `valueChanges` и получать данные полей формы каждый раз, когда пользователь вводит в них символ).

Следующий фрагмент кода ищет данные о погоде в городе, введенном в форму, и выводит результат в консоль:

Листинг 11.26. Компонент погоды, использующий FormControl

```
@Component({
    ...
    template: `<input type="text" [formControl]="weatherControl">`
})
class FormComponent {
  weatherControl: FormControl = new FormControl();

  constructor() {
      this.weatherControl.valueChanges
          .pipe(
```

Использует formControl с привязкой свойств

Вместо определения модели формы создает самостоятельный экземпляр FormControl

Использует наблюдаемый объект valueChanges для получения значения из формы

```
            switchMap(city => this.getWeather(city))
        )
        .subscribe(weather => console.log(weather));
  }
}
```

Использует оператор RxJS для переключения на другой наблюдаемый объект, возвращаемый getWeather()

Подписывается на valueChanges и выводит данные о погоде, полученные из этого наблюдаемого объекта

В библиотеке RxJS присутствуют десятки операторов, которые можно применить к единице данных, передаваемой наблюдаемым объектом, до ее передачи в метод `subscribe()`. Не углубляясь в детали, мы просто отметим, что в предыдущем фрагменте кода метод `getWeather()` делает HTTP-запрос на сервер погоды и возвращает наблюдаемый объект. Оператор `switchMap` получает данные из наблюдаемого объекта `valueChanges` и передает их в `getWeather()`, который также возвращает наблюдаемый объект.

В главе 12 мы будем использовать реактивный API Forms в классе `AppComponent` для обработки форм с транзакциями в блокчейн:

```
this.transactionForm = fb.group({
    sender    : ['', Validators.required],
    recipient : ['', Validators.required],
    amount    : ['', Validators.required]
});
```

Этот код отобразит форму с тремя элементами управления вводом: `sender`, `recipient` и `amount`. Каждый из этих элементов в качестве начального значения получит пустую строку с прикрепленным к ней `Validators.required`.

11.8. ОСНОВЫ МАРШРУТИЗАЦИИ

В одностраничном приложении (SPA) веб-страница не будет перезагружаться, но ее составляющие могут изменяться. Мы хотим добавить в подобное приложение навигацию, чтобы оно изменялось в области содержимого страницы в ответ на действия пользователя. Маршрутизатор Angular позволяет настроить и реализовать подобную навигацию без выполнения полной перезагрузки страницы.

Посадочная страница SPA будет иметь некоторые части, которые остаются на ней постоянно, в то время как другие компоненты будут отображаться на основе действий пользователя или других событий. На рис. 11.8 показан пример страницы, где панель навигации расположена сверху, поисковая панель — слева, а футер будет отображаться на странице постоянно. Большая же площадь, отмеченная тегом `<router-outlet>`, служит для отображения различных компонентов по одному. Изначально эта область вывода маршрутизатора (`router outlet`) может отображать `HomeComponent`, а когда пользователь щелкнет по ссылке, маршрутизатор может показать там `ProductComponent`.

Каждое приложение имеет один объект маршрутизатора, и для установки навигации вам нужно настроить маршруты этого приложения. Angular включает множество классов, поддерживающих навигацию, например `Router`, `Route`, `Routes`, `ActivatedRoutes` и др. Вы можете настроить маршруты в массиве объектов типа `Route`.

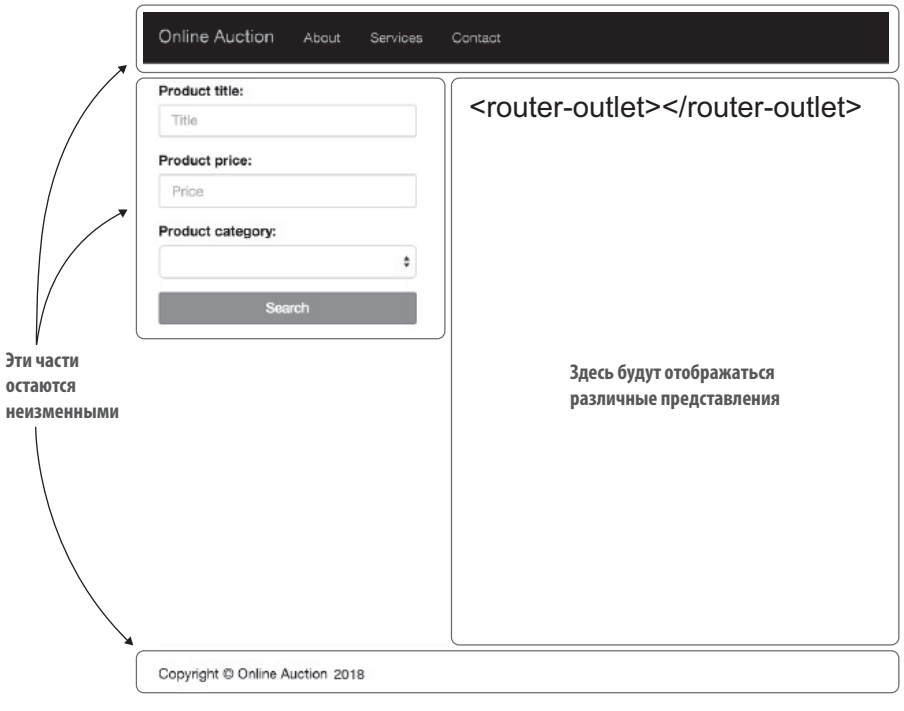


Рис. 11.8. Страница с областью router-outlet

Листинг 11.27. Пример настройки маршрутов

```
const routes: Routes = [
    {path: '',        component: HomeComponent},
    {path: 'product', component: ProductDetailComponent}
];
```

Пустой путь указывает, что по умолчанию отображается HomeComponent

Если URL содержит сегмент product — отображает ProductDetailComponent

Так как настройка маршрутов выполняется на уровне модуля, необходимо сообщить модулю приложения о маршрутах в декораторе `@NgModule`. Если вы объявляете маршруты для корневого модуля, используйте метод `forRoot()`, как показано в следующем листинге.

Листинг 11.28. Предоставление корневому модулю данных о маршрутах

```
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule } from '@angular/router';
...
@NgModule({
    imports: [BrowserModule,
              RouterModule.forRoot(routes)],
      ...
})
```

Создать модуль маршрутизатора и сервис для корневого модуля приложения

Давайте рассмотрим простое приложение, расположенное в директории router. Мы сгенерировали его при помощи команды `ng new router --minimal`. Когда нам был задан вопрос: «Would you like to add Angular routing?», мы выбрали «Yes», и CLI сгенерировал файл app-routing.module.ts.

У `AppComponent` в верхней части страницы есть две ссылки — `Home` и `Product`. Приложение отображает либо `HomeComponent`, либо `ProductDetailComponent`, в зависимости от того, на какой ссылке щелкает пользователь. `HomeComponent` отображает текст «Home Component», а `ProductDetailComponent` отображает «Product Detail Component». Изначально веб-страница показывает `HomeComponent`, как изображено на рис. 11.9.

Когда пользователь щелкает по ссылке `Product`, маршрутизатор должен отобразить `ProductDetailComponent`, как показано на рис. 11.10. Посмотреть URL для этих маршрутов вы можете на рис. 11.9 и 11.10.

Рис. 11.9. Отображение HomeComponent

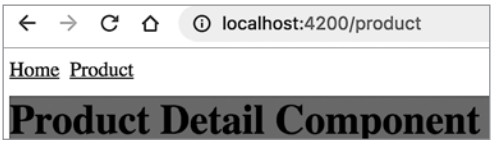


Рис. 11.10. Отображение ProductDetailComponent

Главная цель этого элементарного приложения — знакомство с маршрутизатором, поэтому его компоненты очень просты. В следующем листинге показан код `HomeComponent`.

Листинг 11.29. home.component.ts: класс HomeComponent

```
import {Component} from '@angular/core';

@Component({
    selector: 'home',
    template: '<h1 class="home">Home Component</h1>',
    styles: ['.home {background: red}']})
export class HomeComponent {}
```

Отображает этот компонент на красном фоне

Как видно из следующего листинга, код `ProductDetailComponent` выглядит аналогичным образом, но при этом в качестве фона использует синий цвет.

Листинг 11.30. product-detail.component.ts: класс ProductDetailComponent

```
import {Component} from '@angular/core';

@Component({
    selector: 'product',
    template: '<h1 class="product">Product Detail Component</h1>',
    styles: ['.product {background: cyan}']}) ◄— Отображает этот компонент на синем фоне
export class ProductDetailComponent {}
```

Angular CLI сгенерировал отдельный модуль для маршрутизации в файле app-routing.module.ts. Корневой модуль импортирует настроенный `RouterModule` из этого файла, показанного в следующем листинге. Мы передаем в метод `forRoot()` объект конфигурации с объявленными маршрутами. В данном случае в интерфейсе `Routes` определены всего два свойства: `path` и `component`.

Листинг 11.31. app-routing.module.ts: модуль с настроенными маршрутами

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home.component';
import { ProductDetailComponent } from './product-detail.component';

const routes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'product', component: ProductDetailComponent }
];

@NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule] )
})
export class AppRoutingModule { }
```

HomeComponent отображается в путь, содержащий пустую строку, что делает его маршрутом по умолчанию

Если URL содержит сегмент product, отображается в области вывода маршрутизатора ProductDetailComponent

Делает маршруты доступными в RouterModule

Экспортирует настроенный RouterModule, чтобы он мог быть импортирован корневым модулем

Следующий шаг — создание корневого компонента, который будет содержать ссылки для навигации между представлениями `Home` и `Product`.

Листинг 11.32. app.component.ts: компонент высшего уровня

```
import {Component} from '@angular/core';

@Component({
    selector: 'app-root',
    template: `
        <a [routerLink]="['/']">Home</a>
        <a [routerLink]="['/product']">Product Details</a>
        <router-outlet></router-outlet>
    `
})
export class AppComponent {}
```

Создает ссылку, которая привязывает routerLink к пустому пути

Создает ссылку, которая привязывает routerLink к пути /product

<router-outlet> определяет область страницы, где маршрутизатор будет отображать компоненты (по одному)

Квадратные скобки вокруг `routerLink` означают привязку свойства, в то время как скобки в той же строке справа представляют массив с одним элементом (например, `['/']`). Во втором теге привязки свойство `routerLink` привязано к компоненту, настроенному для пути `/product`.

Путь предоставляется в виде массива, потому что он может включать параметры, передаваемые в процессе навигации. Например, `['/product', 123]` может давать маршрутизатору команду следовать к компоненту, который отобразит информацию о товаре с ID 123. Соответствующие компоненты будут отображаться в области, отмеченной `<router-outlet>`, которая в данном приложении расположена под тегами привязки. Ни один из компонентов не знает о конфигурации маршрутизатора, поскольку она произведена на уровне модуля, как показано в следующем листинге.

Листинг 11.33. app.module.ts: корневой модуль

```
...
@NgModule({
    declarations: [
        AppComponent, HomeComponent, ProductDetailComponent
    ],
    imports: [
        BrowserModule,
        AppRoutingModule
    ],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Объявляет компоненты, принадлежащие модулю

Импортирует модуль с предварительно настроенными маршрутами

Для запуска приложения, описанного в этом разделе, установите зависимости, выполнив команду `npm install` в директории маршрутизатора. Затем команда `ng serve -o` запустит сервер и откроет браузер на localhost:4200. Браузер, в свою очередь, отобразит окно, показанное ранее на рис. 11.8.

Мы только что продемонстрировали вам очень простое приложение, использующее маршрутизатор Angular. Фактически же он предлагает гораздо больше возможностей:

- Передача параметров в процессе навигации.
- Подписка на изменение параметров родительского компонента.
- Защита маршрутов: применение бизнес-логики, которая может помешать пользователю следовать по маршрутам.
- Ленивая загрузка модулей в процессе навигации.
- Возможность определять в компоненте более одной области вывода маршрутизатора.

Angular — это очень серьезное решение для разработки одностраничных приложений, а маршрутизатор играет главную роль в навигации на стороне клиента.

На этом наше краткое знакомство с фреймворком Angular завершается. Оно не сделает вас экспертом, но по крайней мере вы уже будете готовы читать и понимать код новой версии блокчейн-клиента, представленного в главе 12.

Из-за нехватки места мы не объясняли принципы реактивного программирования, поддерживаемые включенной в Angular библиотекой RxJS. Мы также не показали Angular Material — набор современных компонентов UI. Эти темы рассматриваются в книге «Angular и TypeScript. Сайтостроение для профессионалов» (СПб.: Питер, 2018).

ИТОГИ

- Angular — это фреймворк, имеющий все необходимое для разработки одностраничных приложений. Он включает маршрутизатор и поддерживает внедрение зависимостей, работу с формами и многое другое.
- С помощью Angular CLI вы можете сгенерировать свое первое Angular-приложение буквально за одну-две минуты. Это приложение будет полностью настроено и готово к запуску.
- Angular использует декораторы TS для разных целей, например для объявления компонентов, внедряемых сервисов, а также свойств ввода/вывода.
- Angular CLI имеет встроенный инструмент, позволяющий вам собирать оптимизированные связки для продакшена и неоптимизированные для режима разработки.
- Сам Angular был написан в TypeScript, который также является рекомендованным языком для разработки веб-приложений с помощью этого фреймворка.

12 Разработка клиента блокчейна на Angular

В этой главе:

- ✓ Код веб-блокчейн-клиента в Angular.
- ✓ Запуск клиента Angular, сообщающегося с WebSocket-сервером.

В этой главе мы рассмотрим новую версию блокчейн-приложения, клиентская часть которого будет написана в Angular. Исходный код располагается в двух директориях: client и server. Но если в главе 10 эти директории являлись частью одного проекта, то на этот раз они являются двумя различными проектами и имеют собственные файлы package.json. В реальных приложениях фронтенд- и бэкенд-части обычно представлены в виде раздельных проектов.

Код сервера обмена сообщениями тот же, что и в главе 10, и функциональность этой версии приложения та же. Единственное отличие в том, что реализация фронтенда была полностью переписана на Angular. Давайте посмотрим это приложение в действии.

ПРИМЕЧАНИЕ Чтобы освежить в памяти функциональность блокчейн-клиента и сервера обмена сообщениями, вернитесь к главе 10.

12.1. ЗАПУСК БЛОКЧЕЙН-ПРИЛОЖЕНИЯ ANGULAR

Код этого приложения состоит из сервера обмена сообщениями и веб-клиента. Чтобы запустить сервер, откройте терминал в директории server, выполните `npm`

`install` для установки зависимостей сервера, а затем `npm start`. Вы увидите сообщение «Listening on http://localhost:3000» (Прослушивание на http://localhost:3000). Оставьте сервер запущенным и переходите к запуску клиента.

Чтобы запустить клиент Angular, откройте другое окно терминала в директории client, выполните `npm install` для установки Angular и его зависимостей, а затем выполните `npm start`. В клиентском файле package.json команда start является псевдонимом для уже знакомой нам команды `ng serve`. Она соберет связки так же, как и во всех приложениях, рассмотренных в главе 11, и вы сможете открыть браузер на localhost:4200.

В этот момент у вас будет запущено два сервера: dev-сервер, установленный Angular CLI, и WebSocket-сервер обмена сообщениями, выполняемый под Node.js, как изображено на рис. 12.1.

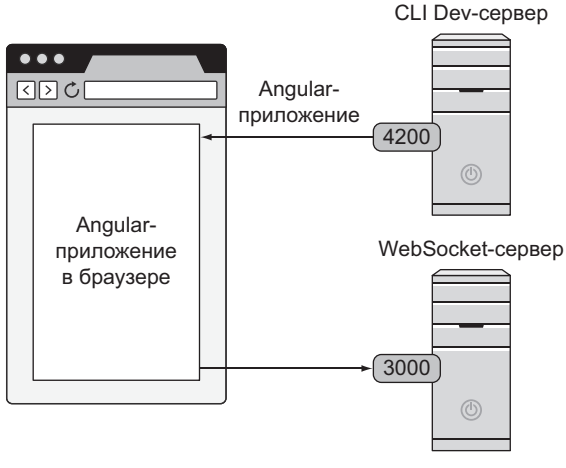


Рис. 12.1. Одно приложение, два сервера

Если бы вместо WebSocket-сервера вы запустили HTTP-сервер, то пришлось бы настраивать прокси, чтобы обойти ограничения правила единого домена. Вы можете подробнее узнать о проксировании бэкенд-сервера в документации к Angular здесь: http://mng.bz/nvl2.

> **ПРИМЕЧАНИЕ** Если вы захотите развернуть это приложение в продакшене, то понадобится WebServer, который станет хостом для связок блокчейн-клиента, и тогда можно запустить WebSocket-сервер на том же порте. Вы можете собирать оптимизированные связки для развертывания, выполняя команду `ng build --prod`. Процесс развертывания описан в документации Angular: https://angular.io/ guide/deployment.

Приложению потребуется некоторое время на генерацию первичного блока, а затем вы увидите знакомое окно, как показано на рис. 12.2.

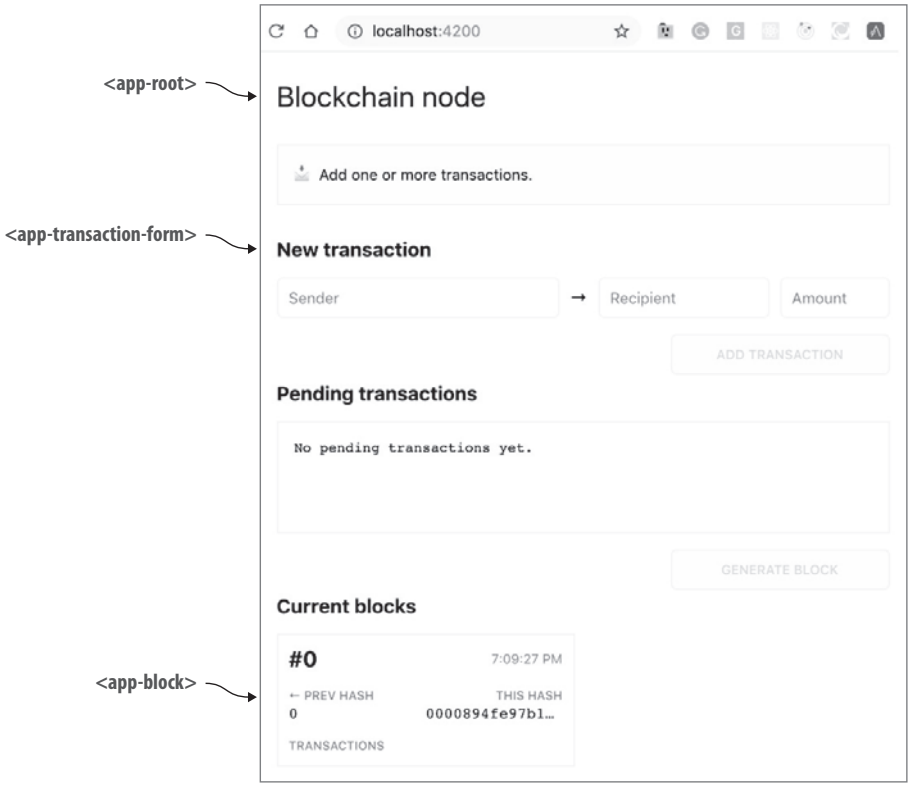


Рис. 12.2. Запуск блокчейн-клиента, написанного на Angular

У этого приложения три компонента:

- `AppComponent` — компонент высшего уровня. Его селектор — `app-root`.
- `TransactionFormComponent` — компонент, отображающий транзакции. Его селектор — `app-transaction-form`. Этот компонент является формой, содержащей три поля ввода и кнопку `ADD TRANSACTION`.
- `BlockComponent` — компонент, отображающий данные блока. Его селектор — `app-block`.

Структура проекта Angular показана на рис. 12.3, где стрелки указывают на файлы или директории, в которых вы найдете исходный код блокчейн-клиента.

Файлы, представляющие `BlockComponent`, расположены в директории block. Файлы, относящиеся к `TranscationFormComponent`, можно найти в директории transaction-form. Директория shared содержит повторно используемые сервисы `BlockchainNodeService`, `CryptoService` и `WebsocketService`.

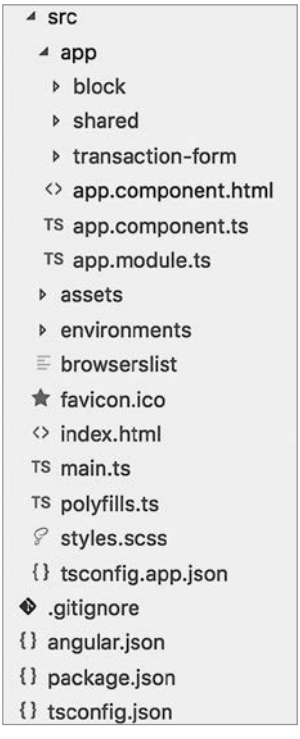


Рис. 12.3. Структура проекта Angular

12.2. ОБЗОР APPCOMPONENT

Мы не станем рассматривать код, относящийся к функциональности блокчейна, так как уже делали это в главах 8–10. Рассмотрим только код, показывающий, как все сделано в Angular.

Код для корневого компонента расположен в двух файлах: app.component.html и app.component.ts. В следующем листинге показан шаблон компонента высшего уровня.

Листинг 12.1. Шаблон AppComponent: app.component.html

```
<main>
    <h1>Blockchain node</h1>
    <aside><p>{{ statusLine }}</p></aside>
    <section>
        <app-transaction-form></app-transaction-form>
    </section>
    <section>
        <h2>Pending transactions</h2>
        <pre class="pending-transactions list">{{ formattedTransactions }}</pre>
        <div class="pending-transactions form">
```

Дочерний компонент, в котором пользователь вводит транзакции

```
            <button type="button"
                class="ripple"
                (click)="generateBlock()"
                    [disabled]="node.noPendingTransactions || node.isMining">
                GENERATE BLOCK
            </button>
        </div>
        <div class="clear"></div>
    </section>
    <section>
        <h2>Current blocks</h2>
        <div class="blocks">
            <div class="blocks ribbon">
                <app-block
                    *ngFor="let blk of node.chain; let i = index"
                            [block]="blk"
                            [index]="i">
                </app-block>
            </div>
            <div class="blocks overlay"></div>
        </div>
    </section>
</main>
```

Добавляет обработчик события click для кнопки, генерирующей блоки

Привязывает атрибут кнопки disabled (отключен)

Дочерний компонент, где отображаются блоки

Перебирает существующие в цепочке блоки

Привязывает индекс итератора к свойству input, в BlockComponent

Привязывает объект block к свойству block в BlockComponent

В этом шаблоне мы используем привязку три раза:

- В листинге 12.1 свойство `disable` кнопки `GENERATE BLOCK` может быть либо `true`, либо `false`, в зависимости от значения выражения `node.noPendingTransactions || node.isMining`.
- Свойство `block` компонента `<app-block>` получает значение из текущего `blk` во время выполнения директивой `ngFor` итерации по массиву `node.chain`, который является свойством класса `AppComponent` (показанного в листингах 12.2, 12.3 и 12.4).
- Свойство `index` компонента `<app-block>` получает значение из переменной `index`, предоставленной директивой `ngFor`. Это значение представляет текущее значение итератора цикла.

Строка `(click)="generateBlock()"` объявляет обработчик событий для события `click`. В шаблонах Angular круглые скобки используются для определения обработчика событий.

Шаблон в листинге 12.1 включен в декоратор `@Component()` TS-класса `AppComponent`. Его первая часть приведена в следующем листинге.

Листинг 12.2. Первая часть app.component.ts

```
import {Component} from '@angular/core';
import {Message, MessageTypes} from './shared/messages';
import {Block, BlockchainNodeService, formatTransactions, Transaction,
➥ WebsocketService}
```

```
➥ WebsocketService}
            from './shared/services';

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
})
export class AppComponent {

constructor(private readonly server: WebsocketService,
            readonly node: BlockchainNodeService) {
    this.server.messageReceived.subscribe(message =>
                this.handleServerMessages(message));
    this.initializeBlockchain();
}

private async initializeBlockchain() {
    const blocks = await this.server.requestLongestChain();
    if (blocks.length > 0) {
        this.node.initializeWith(blocks);
    } else {
        await this.node.initializeWithGenesisBlock();
    }
}
```

Использует в инструкции import директорию services (не file)

Внедряет два сервиса

Подписывается на сообщения сервисов

Создает экземпляр блокчейна

Вернитесь к шаблону компонента приложения в листинге 12.1. Если пользователь щелкнет по кнопке `GENERATE BLOCK`, он вызовет метод `generateBlock()`, объявленный в TS-классе `AppComponent`. В следующем листинге показано несколько методов из этого класса.

Листинг 12.3. Вторая часть of app.component.ts

```
get statusLine(): string {
    return (
        this.node.chainIsEmpty          ? '⌛ Initializing the blockchain...' :
        this.node.isMining              ? '⌛ Mining a new block...' :
        this.node.noPendingTransactions ? '✉ Add one or more transactions.' :
                                          '✓ Ready to mine a new block.'
    );
  }

  get formattedTransactions() {
    return this.node.hasPendingTransactions
        ? formatTransactions(this.node.pendingTransactions)
        : 'No pending transactions yet.';
  }

  async generateBlock(): Promise<void> {
        this.server.requestNewBlock(this.node.pendingTransactions)
    const miningProcessIsDone =
➥ this.node.mineBlockWith(this.node.pendingTransactions);

    const newBlock = await miningProcessIsDone;
```

Обработчик события click для кнопки GENERATE BLOCK

```
    this.addBlock(newBlock);
  };

private async addBlock(block: Block, notifyOthers = true): Promise<void> {
    try {
        await this.node.addBlock(block);
        if (notifyOthers) {
            this.server.announceNewBlock(block);
        }
        } catch (error) {
            console.log(error.message);
    }
}
```

Эта функция пытается добавить новый блок в блокчейн

Новый блок был принят блокчейном

Новый блок был отвергнут блокчейном

Обратите внимание, как использование ключевых слов `async` и `await` позволяет нам написать код, который выглядит так, как будто выполняется синхронно, несмотря на то что каждый вызов функции, имеющий перед собой `await`, выполняется асинхронно.

ОРГАНИЗАЦИЯ ИМПОРТОВ С ПОМОЩЬЮ INDEX.TS

Обратите внимание, что класс `AppComponent` импортирует несколько классов, просто указывая имя директории, вместо того чтобы содержать несколько инструкций, указывающих на разные файлы. Это возможно благодаря тому, что мы ввели специальный TS-файл index.ts в директорию shared/services. Содержимое этого файла показано в следующем фрагменте кода:

```
export * from './blockchain-node.service';
export * from './crypto.service';
export * from './websocket.service';
```

Здесь мы повторно экспортируем все члены, экспортированные из трех файлов, перечисленных в этом файле. Если директория включает файл с именем index.ts, вы можете упростить инструкции импорта, просто использовав имя директории, и tsc найдет члены для импорта в файлах, включенных в index.ts:

```
import {Block, BlockchainNodeService, formatTransactions, Transaction,
        ➥ WebsocketService}
       from './shared/services';
```

Если бы этого файла не было, нам бы пришлось писать пять инструкций импорта, указывающих на разные файлы.

Третья часть кода app.component.ts, приведенная в следующем листинге, показывает методы в классе `AppComponent`, которые обрабатывают сообщения сервера, переданные через WebSocket. Они обрабатывают запросы длиннейшей цепочки и новых блоков. Вся эта функциональность объяснялась в главе 10.

Листинг 12.4. Третья часть app.component.ts

```
handleServerMessages(message: Message) {
        switch (message.type) {  <-- Обрабатывает сообщения WebSocket-сервера
            case MessageTypes.GetLongestChainRequest: return
➥ this.handleGetLongestChainRequest(message);
            case MessageTypes.NewBlockRequest : return
➥ this.handleNewBlockRequest(message);
            case MessageTypes.NewBlockAnnouncement : return
➥ this.handleNewBlockAnnouncement(message);
            default: {
                console.log(`Received message of unknown type:
                            "${message.type}"`);
            }
        }
    }

    private handleGetLongestChainRequest(message: Message): void {
        this.server.send({                                  Обрабатывает запросы
            type: MessageTypes.GetLongestChainResponse,     длиннейшей цепочки
            correlationId: message.correlationId,
            payload: this.node.chain
        });
    }

    private async handleNewBlockRequest(message: Message): Promise<void> {
        const transactions = message.payload as Transaction[];
        const newBlock = await this.node.mineBlockWith(transactions);
        this.addBlock(newBlock);
    }

    private async handleNewBlockAnnouncement(message: Message): Promise<void> {
        const newBlock = message.payload as Block;
        this.addBlock(newBlock, false);                     Обрабатывает анонсирова-
    }                                                       ние нового блока
}
```

Вернитесь к шаблону `AppComponent` из листинга 12.1, и вы найдете ссылку на дочерний компонент `<app-transaction-form>`. Этот компонент мы рассмотрим следующим.

12.3. РАССМОТРЕНИЕ TRANSACTION FORM COMPONENT

Шаблон `AppComponent` служит хостом для двух дочерних компонентов: `TransactionFormComponent` и `BlockComponent`. Шаблон `TransactionFormComponent` — это форма с тремя элементами контроля, содержащая кнопку `ADD TRANSACTION`, как показано в листинге ниже. Мы взяли стандартный HTML-тег `<form>` и добавили в него директиву `[formGroup]= "transactionForm"`, чтобы включить поддержку API реактивных форм, предлагаемых Angular.

Листинг 12.5. transaction-form.component.html: UI TransactionFormComponent

```
<h2>New transaction</h2>
<form class="add-transaction-form"
        [formGroup]="transactionForm"
        (ngSubmit)="enqueueTransaction()">
    <input type="text"
            name="sender"
            autocomplete="off"
            placeholder="Sender"
            formControlName="sender">
    <span class="hidden-xs">•</span>

    <input type="text"
            name="recipient"
            autocomplete="off"
            placeholder="Recipient"
            formControlName="recipient">
<input type="number"
            name="amount"
            autocomplete="off"
            placeholder="Amount"
            formControlName="amount">

    <button type="submit"
            class="ripple"
            [disabled]="transactionForm.invalid || node.isMining">
            ADD TRANSACTION
    </button>
</form>
```

Привязывает свойство класса transactionForm к директиве Angular formGroup

При щелчке по кнопке Submit вызывает enqueueTransaction()

Имя соответствующего свойства в модели формы

Использует привязку свойств для условного отключения кнопки Submit

В разделе 11.7 мы коротко познакомили вас с реактивными формами Angular, и листинг 12.5 также использует директивы этого API. Обратите внимание, что мы начали с привязки объекта модели `transactionForm` (определенной в классе `BlockComponent`) к атрибуту `formGroup`. Помимо этого, каждый элемент управления формой имеет атрибут `formControlName`, отвечающий за одноименное свойство объекта `transactionForm`. Код `TransactionFormComponent` показан в следующем листинге.

Листинг 12.6. transaction-form.component.ts: класс TransactionFormComponent

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { BlockchainNodeService } from '../shared/services';

@Component({
    selector: 'app-transaction-form',
    templateUrl: './transaction-form.component.html'
})
export class TransactionFormComponent {
    readonly transactionForm: FormGroup;

    constructor(readonly node: BlockchainNodeService,
                fb: FormBuilder) {
```

Внедряет сервисы

```
        this.transactionForm = fb.group({
            sender : ['', Validators.required],
            recipient: ['', Validators.required],
            amount : ['', Validators.required]
        });
    }

    enqueueTransaction() {
        if (this.transactionForm.valid) {
            this.node.addTransaction(this.transactionForm.value);
            this.transactionForm.reset();
        }
    }
}
```

Объявляет объект модели transaction- Form

У каждого элемента управления формой есть необходимый валидатор, но нет начального значения

Этот метод вызывается, когда вы добавляете в список ожидания новую транзакцию

И снова попросим вас вернуться к шаблону `AppComponent` в листинге 12.1, где вы увидите цикл `*ngFor`, отображающий дочерние компоненты `<app-block>`. Его мы рассмотрим следующим.

12.4. ОБЗОР BLOCK COMPONENT

`BlockComponent` отвечает за отображение одного блока, и его шаблон показан в следующем листинге. Этот шаблон достаточно прост. Он содержит ряд тегов `<div>` и `<span>`, а свойства `Block` вставлены при помощи привязки, представленной в виде фигурных скобок.

Листинг 12.7. block.component.html: UI BlockComponent

```
<div class="block">
    <div class="block header">
        <span class="block index">#{{ index }}</span>
        <span class="block timestamp">{{ block.timestamp |
➥ date:'mediumTime' }}</span>
    </div>
    <div class="block hashes">
        <div class="block hash">
            <div class="block label">• PREV HASH</div>
            <div class="block hash-value">{{ block.previousHash }}</div>
        </div>
        <div class="block hash">
            <div class="block label">THIS HASH</div>
            <div class="block hash-value">{{ block.hash }}</div>
        </div>
    </div>
    <div>
        <div class="block label">TRANSACTIONS</div>
        <pre class="block transactions">{{ formattedTransactions }}</pre>
    </div>
</div>
```

Вставляет в шаблон значения свойства Block

Вставляет форматированные транзакции

В верхней части шаблона для форматирования даты мы использовали фильтр `date, block.timestamp | date: 'mediumTime'`, который отобразит дату в виде `h:mm:ss`. Отображенный блок показан на рис. 12.4. Вы можете прочитать о фильтре `date` в документации Angular по ссылке https://angular.io/api/common/DatePipe.

Рис. 12.4. Блок, отображенный в браузере

В следующем листинге показан класс `BlockComponent`. Он является компонентом представления, который просто получает от своего родителя значения и отображает их. Логика приложения здесь не применяется.

Листинг 12.8. block.component.ts: класс BlockComponent

```
import { Component, Input } from '@angular/core';
import { Block, formatTransactions } from '../shared/services';

@Component({
    selector: 'app-block',
    templateUrl: './block.component.html'
})

export class BlockComponent {
    @Input() index: number;   <-- Получает от родительского компонента index
    @Input() block: Block;    <-- Получает от родительского компонента Block

    get formattedTransactions(): string {
        return formatTransactions(this.block.transactions);
    }                          <-- Форматирует транзакции, используя функцию
}                                  из файла blockchain-node.service.ts
```

И еще раз вернемся к листингу 12.1, где шаблон родительского компонента использует директиву `*ngFor`, чтобы выполнить цикл по всем блокам в блокчейне и передать данные каждому экземпляру `BlockComponent`, как еще раз показано здесь.

Листинг 12.9. Фрагмент из app.component.html

```
<app-block
    *ngFor="let blk of node.chain; let i = index" [block]="blk"
    [index]="i">
</app-block>1
```

Экземпляр объекта блока (`blk`) и текущий индекс (`i`) передаются в экземпляр `BlockComponent` посредством привязок через свойства `@input()`. Предыдущий фрагмент кода может показаться немного запутывающим, поэтому давайте представим, что родительский компонент должен отображать только один блок. В следующем листинге показано, как подобный родитель может передать данные в `BlockComponent`.

Листинг 12.10. Родитель, передающий данные потомку

```
@Component({
    selector: 'app-parent',
    template: ` Meet my child
    <app-block
            [block]="blk"  <-- Привязывает значение blk к свойству потомка block
            [index]="blockNumber"> <-- Привязывает значение blockNumber к свойству потомка index
    </app-block>
    `
})

export class ParentComponent {
    blk: Block =                                    <-- Инициализирует в родителе значения
        { hash: "00005b1692f26",
            nonce: 2634,
            previousHash: "0000734b922d",
            timestamp: 25342683;
            transactions: ["John to Mary $100",
                           "Alex to Nina $400"];
        };
    blockNumber: 123;
}
```

ПРИМЕЧАНИЕ Потомок может передавать данные родителю посредством свойств, отмеченных декоратором `@Output()`. Вы можете прочитать об этом в блоге Якова Файна «*Angular 2: Component communication with events vs callbacks*» по ссылке http://mng.bz/vlQ4.

До сих пор мы рассматривали компоненты (классы с UI) нашего блокчейн-приложения. Теперь же давайте рассмотрим сервисы (классы с логикой приложения).

12.5. ОБЗОР СЕРВИСОВ

В главе 10 блокчейн-приложение имело класс `WebsocketController`, который был инстанцирован при помощи ключевого слова `new`. Здесь же аналогичная функциональность обернута в сервис, инстанцированный и внедренный Angular. В этом проекте сервисы располагаются в директории shared/services.

В следующем листинге показан фрагмент `WebsocketService`, отвечающий за весь обмен данными с WebSocket-сервером.

Листинг 12.11. Фрагмент из websocket.service.ts

```
interface PromiseExecutor<T> {
    resolve: (value?: T | PromiseLike<T>) => void;
    reject: (reason?: any) => void;
}

@Injectable({
    providedIn: 'root'
})
export class WebsocketService {
    private websocket: Promise<WebSocket>;
    private readonly messagesAwaitingReply = new Map<UUID,
➥ PromiseExecutor<Message>>();
    private readonly _messageReceived = new Subject<Message>();

    get messageReceived(): Observable<Message> {
        return this._messageReceived.asObservable();
    }

constructor(private readonly crypto: CryptoService) {
    this.websocket = this.connect();
}

private get url(): string {
    const protocol = window.location.protocol === 'https:' ? 'wss' : 'ws';
    const hostname =    environment.wsHostname;
    return `${protocol}://${hostname}`;
}

private connect(): Promise<WebSocket> {
    return new Promise((resolve, reject) => {
        const ws = new WebSocket(this.url);
        ws.addEventListener('open', () => resolve(ws));
        ws.addEventListener('error', err => reject(err));
        ws.addEventListener('message', this.onMessageReceived);
    });
}
```

PromiseExecutor знает, какой клиент ожидает ответа

Это сервис-одиночка, доступный для всех компонентов и других сервисов

Создает экземпляр RxJS Subject

Получает наблюдаемую часть Subject

Подключается к WebSocket-серверу

Получает URL сервера из переменной среды

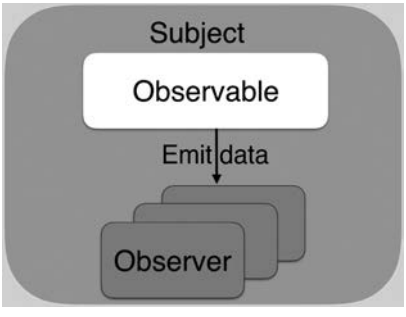
ПРИМЕЧАНИЕ Тип `PromiseExecutor` был добавлен в листинге 10.26.

Объект `WebsocketService` внедряется в `AppComponent`:

```
export class AppComponent {
    constructor(private readonly server: WebsocketService,
            readonly node: BlockchainNodeService) {
this.server.messageReceived.subscribe(message =>
                    this.handleServerMessages(message));
  ...
    }
  ...
}
```

RXJS: OBSERVABLE, OBSERVER И SUBJECT

Библиотека RxJS предлагает различные способы обработки потоков данных. Если у вас есть экземпляр объекта `Observable`, то вы можете вызвать для него `subscribe()`, при условии, что экземпляр `Observer` знает, что делать с полученными данными. Каждый раз, когда `Observable` отправляет новые данные, `Observer` будет их обрабатывать.



Рассылка с помощью RxJS Subject

`RxJS Subject` же инкапсулирует `Observable` и `Observer`. Один `Subject` может включать множество наблюдателей, каждый из которых будет представлять подписчика. Чтобы разослать данные по всем подписчикам, мы вызываем для `Subject` метод `next(someData)`. Чтобы подписаться на получение данных, мы вызываем для него же метод `subscribe()`.

Следующий фрагмент кода создает экземпляр `Subject` и двух подписчиков. В последней строке код отправляет 123, и это значение будет получено каждым подписчиком.

```
const mySubject = new Subject();  <-- Создает Subject
...
const subscription1 = mySubject.subscribe(...);?  <-- Создает первого подписчика
const subscription2 = mySubject.subscribe(...);?  <-- Создает второго подписчика
...
?mySubject.next(123);  <-- Рассылает подписчикам 123
```

Если вы хотите ограничить возможности части кода, чтобы она могла только подписываться на субъект, но не отправлять ему данные, пропишите в этой части только наблюдаемый сегмент `Subject`, используя метод `asObservable()`, как показано в геттере `messageReceived` листинга 12.11.

`AppComponent` подписывается на получаемые от сервера сообщения и обрабатывает их. Помимо этого, он также отправляет серверу свои сообщения, касающиеся запросов длиннейшей цепочки или анонсирования нового блока.

Сервис `WebsocketService` получает URL WebSocket-сервера из переменной среды `environment.wsHostname`, которая определена в каждом файле директории `environments` проекта. Поскольку наш клиент выполняется в режиме разработки (`ng serve`), он использует настройку из файла `environment.ts`, показанного в следующем листинге.

Листинг 12.12. Файл environments/environment.ts

```
export const environment = {
    production: false,  <-- Код выполняется в режиме разработки
    wsHostname: 'localhost:3000'  <-- URL для dev-сервера WebSocket
};
```

Если вы запустили клиент командой `ng serve --prod` или собрали файлы командой `ng build --prod`, ваше приложение будет использовать файл environment.prod.ts, который может иметь другое значение в `wsHostname`.

Директория shared/services также содержит следующие файлы:

- blockchain-node.service.ts — код, создающий блок. В приложении главы 10 эта функциональность была реализована в файле blockchain-node.ts.
- crypto.service.ts — содержит метод `sha256()`, который занимается вычислением хеш-значения.

На этом завершается рассмотрение кода Angular-версии клиентского блокчейн-приложения.

> **ПРИМЕЧАНИЕ** Если вам нравится синтаксис Angular для разработки клиентской стороны веб-приложений, обратите внимание на фреймворк для серверной стороны под названием NestJS (https://github.com/nestjs). Он выполняется в среде Node.js, и его синтаксис покажется знакомым для любого Angular-разработчика. Nest.js поддерживает TS, имеет встроенный инструмент CLI и даже модуль TypeORM, позволяющий работать с относительными базами данных с серверной стороны TS.

ИТОГИ

- В режиме разработки мы, как правило, запускаем приложения Angular с двумя веб-серверами: один обслуживает данные, а второй обслуживает само веб-приложение. Последний поставляется с Angular.
- Обычно весь обмен данными с сервером реализуется во внедряемых в компоненты сервисах. В нашем блокчейн-клиенте веб-клиент реализует связь с WebSocket-сервером в TS-классе `WebSocketService`.
- RxJS-класс `Subject` предлагает возможность рассылки, которую в нашем блокчейн-приложении мы используем для отправки сообщений нескольким узлам блокчейна.

13 Разработка приложений React.js с помощью TypeScript

В этой главе:

- ✓ Знакомство с библиотекой React.js.
- ✓ Использование свойств и состояния в компонентах React.
- ✓ Взаимодействие компонентов.

Библиотека React.js (она же React) была создана Facebook-инженером Джорданом Уолком (Jordan Walke) в 2013 году. Сегодня она уже имеет 1300 участников и 140 000 звезд на GitHub. Согласно опросу разработчиков StackOverflow, проведенному в 2019 году, React является второй по популярности JS-библиотекой (jQuery сохраняет лидирующую позицию). React — это не фреймворк, а именно библиотека, которая отвечает за отображение представлений в браузере (вспомните о букве V в шаблоне проектирования MVC). В этой главе мы покажем вам, как начать разрабатывать приложения в React, используя TS.

Главной составляющей этой библиотеки являются компоненты, а UI веб-приложения состоит из элементов, имеющих отношения по принципу «родитель-потомок». Но там, где Angular берет под контроль весь корневой элемент веб-страницы, React позволяет вам контролировать менее масштабный элемент страницы (вроде `<div>`), даже если остальная ее часть реализована с помощью другого фреймворка или в чистом JavaScript.

Вы можете разрабатывать React-приложения либо в JS, либо в TS и развертывать их, используя инструменты вроде Babel или WebPack (описанные в главе 6).

Не углубляясь в теорию, давайте начнем с написания простейшей версии приложения Hello World при помощи React и JS. На TS же мы переключимся в разделе 13.2.

13.1. РАЗРАБОТКА ПРОСТЕЙШЕЙ ВЕБ-СТРАНИЦЫ ПРИ ПОМОЩИ REACT

В этом разделе мы покажем вам две версии простой веб-страницы, написанной на React и JavaScript. Каждая из них будет отображать `"Hello World"`, но первая версия будет использовать React без дополнительных инструментов, а вторая задействует Babel.

Реальные React-приложения — это проекты с настроенными зависимостями, инструментами и процессом сборки, но во избежание усложнения наша первая веб-страница будет иметь всего один HTML-файл, загружающий библиотеку React из CDN. Эта версия страницы Hello World расположена в файле hello-world-simplest/index.html, а ее содержимое показано в следующем листинге.

Листинг 13.1. hello-world-simplest/index.html: приложение Hello World

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">                                  Загружает пакет React из CDN
        <script
➥ crossorigin src="https://unpkg.com/react@16/umd/react.development.js">
          </script>
        <script
➥ crossorigin src="https://unpkg.com/react-dom@16/umd/
➥ react-dom.development.js">  <-- Загружает пакет ReactDOM из CDN
          </script>
    </head>
    <body>
        <div id="root"></div>  <-- Добавляет <div> с id «root»

        <script>                                                Создает элемент <h1>
            const element = React.createElement('h1',           при помощи функции
                                                 null,          createElement
                                Текст элемента <h1> --> 'Hello World');
              ReactDOM.render(element,                                  Мы не передаем данные
                              document.getElementById('root'));         (объект свойств) в элемент <h1>
          </script>
    </body>                                                 Отображает <h1>
</html>                                                     внутри <div>
```

Процессы объявления содержимого страницы (`React.createElement()`) и ее отображения в DOM браузера (`ReactDOM.render()`) разделены. Первый

поддерживается API объекта React, а последний выполняет ReactDOM. Именно поэтому мы загрузили эти два пакета в раздел `<head>` страницы.

В React элементы UI представлены в виде дерева компонентов, которое всегда имеет один корневой компонент. Эта веб-страница содержит `<div>` с `ID root`, который служит в роли такого элемента для содержимого, отображаемого React. В сценарии листинга 13.1 мы подготавливаем этот элемент для отображения при помощи `React.createElement()`, а затем вызываем `reactDOM.render()`, которая находит элемент с ID root и отображает его в этом элементе.

> **ПРИМЕЧАНИЕ** В Chrome щелкните правой кнопкой на странице Hello World и выберите опцию Inspect. Она откроет инструменты разработчика, показав `<div>` с элементом `<h1>` внутри.

Метод `createElement()` имеет три аргумента: имя HTML-элемента, его *свойства* (неизменяемые данные для передачи элементу) и содержимое. В данном случае нам не понадобилось передавать свойства (атрибуты) и мы использовали `null`. Назначение свойств мы поясним далее в разделе 13.4.3. Содержимое `h1` — это `"Hello World"`, но он может содержать дочерние элементы (вроде `ul` с вложенными элементами `li`), которые могут быть созданы посредством вложенных вызовов `createElement()`.

Откройте файл index.html в браузере, и он отобразит текст `"Hello World"`, как показано на рис. 13.1.

Рис. 13.1. Отображение файла hello-world- simplest/index.html

Вызов `createElement()` для страницы, имеющей только один элемент, — это абсолютно нормально, но для страницы с десятками элементов это уже будет нудно и раздражающе. React позволяет вам вкладывать разметку UI в код JS, что выглядит как HTML, но в JSX. Мы это рассмотрим чуть позже во врезке «JSX и TSX».

Давайте посмотрим, как бы выглядела наша страница Hello World, если бы мы использовали JSX. Обратите внимание на строку `const myElement = <h1>HelloWorld</h1>` следующего листинга — мы используем ее вместо вызова `createElement()`.

Листинг 13.2. index_jsx.html: JSX версия Hello World

```
<!DOCTYPE html>
    <head>
        <meta charset="utf-8">
        <script
            src="https://unpkg.com/react@16/umd/react.development.js"></script>
        <script
            src="https://unpkg.com/react-dom@16/umd/
➥ react-dom.development.js"></script>

        <script src="https://unpkg.com/babel-standalone/
➥ babel.min.js"></script>  <-- Добавляет Babel из CDN
    </head>
    <body>
        <div id="root"></div>
          <script type="text/babel">   <-- Сценарий имеет тип text/babel
          const myElement = <h1>Hello World</h1>;   <-- Присваивает переменной JSX значение
          ReactDOM.render(   <-- Инициирует отображение myElement в <div>
                myElement,
                document.getElementById('root')
          );
          console.log(myElement);   <-- Мониторит отображенный объект JavaScript
        </script>

    </body>
</html>
```

Это приложение отображает ту же страницу, что вы видели на рис. 13.1, но написано иначе. В JS-код вложена строка `<h1>HelloWorld!</h1>`, которая выглядит как HTML, но в действительности является JSX. Браузеры не могут это считать, поэтому нам нужен инструмент для преобразования JSX в рабочий JavaScript. Здесь и появляется Babel!

Раздел `<head>` в листинге 13.2 содержит дополнительный тег `<script>`, загружающий Babel из CDN. Мы также изменили тип сценария на text/babel, что заставляет браузеры игнорировать его, но дает команду Babel трансформировать содержимое этого тега `<script>` в JavaScript.

> **ПРИМЕЧАНИЕ** В реальности мы не стали бы использовать CDN для добавления Babel в Node-проект (как мы делали в листинге 13.2), но такой вариант вполне подходит для демонстрации. В Node-приложениях Babel устанавливался бы в проекте локально и являлся бы частью процесса сборки.

На рис. 13.2 приведен скриншот с открытой консолью браузера. Babel преобразовал JSX-значение в объект JS, который был отображен внутри `<div>`, и мы вывели этот объект в консоли.

Теперь, когда у вас есть понимание работы с простыми страницами React, мы перейдем на Node-проекты и проекты, основанные на компонентах

приложения. Давайте рассмотрим некоторые инструменты, используемые React-разработчиками в реальных проектах.

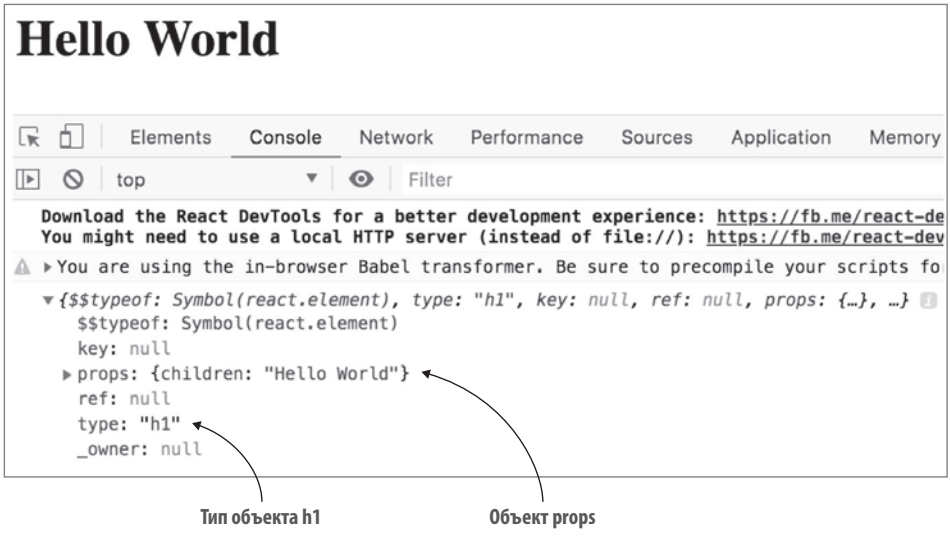


Рис. 13.2. Отображенный объект JavaScript

13.2. ГЕНЕРАЦИЯ И ЗАПУСК НОВОГО ПРИЛОЖЕНИЯ С ПОМОЩЬЮ CREATE REACT APP

Если вы хотите создать приложение React, включающее транспилятор и бандлер, то нужно добавить в него файлы конфигурации. Этот процесс автоматизируется интерфейсом командной строки (CLI) под названием Create React App (подробнее на www.npmjs.com/package/create-react-app). Этот инструмент генерирует все необходимые файлы конфигурации для Babel и WebPack, а вы можете сосредоточиться на написании самого приложения, не тратя время на настройку его инструментов. Для глобальной установки `create-react-app` на компьютер выполните в терминале следующую команду:

```
npm install create-react-app -g
```

Теперь вы можете сгенерировать JS-или TS-версию приложения. Для создания TS-приложения выполните команду `create-react-app`, сопровождаемую его именем и опцией `--typescript`:

```
create-react-app hello-world --typescript
```

Примерно через минуту все необходимые файлы будут сгенерированы в директории hello-world, а также будут установлены зависимости проекта. В частности, будут добавлены следующие пакеты React:

- *react* — библиотека JS для создания пользовательских интерфейсов.
- *react-dom* — пакет React для работы с DOM.
- *react-scripts* — сценарии и конфигурации, используемые инструментом Create React App. Для поддержки TypeScript нужна версия react-сценариев 2.1 или новее.

Помимо перечисленных выше пакетов, CLI устанавливает WebPack, Babel, TypeScript, их файлы определений типов и другие зависимости.

Для запуска сгенерированного веб-приложения переключитесь на директорию hello-world и выполните `npm start`, которая, в свою очередь, выполнит `react-scripts start`. WebPack свяжет приложение, а `webpack-dev-server` определит его на localhost:3000, как показано на рис. 13.3. Эта функциональность предоставляется WebPack DevServer.

> **ПРИМЕЧАНИЕ** Для связывания WebPack использует настройки конфигурации из файла webpack.config.js, расположенного в директории node_modules/react-scripts/config.

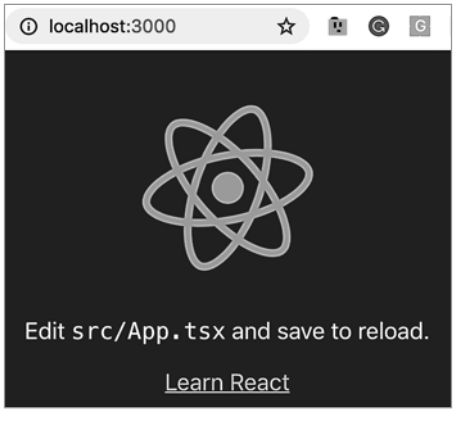


Рис. 13.3. Запуск приложения hello-world

UI сгенерированного приложения предлагает нам отредактировать файл src/App.tsx, являющийся основным TS-файлом этого приложения. Откройте эту директорию в VS Code, и вы увидите файлы проекта, как показано на рис. 13.4.

Исходный код вашего приложения расположен в директории src, а директория public отведена для ресурсов приложения, которые не должны быть включены

в его связки. Например, если в приложении есть тысячи изображений и ему нужно динамически обращаться к их путям, тогда они помещаются в директорию public вместе с другими файлами, не требующими обработки перед развертыванием.

Файл index.html содержит элемент `<div id="root"></div>`, который служит в качестве контейнера для сгенерированного приложения React. В нем вы не найдете теги `<script>`, нужные для загрузки кода библиотеки React. Они будут добавлены в процессе сборки, когда связки приложения уже будут готовы.

Рис. 13.4. Сгенерированные файлы и директории

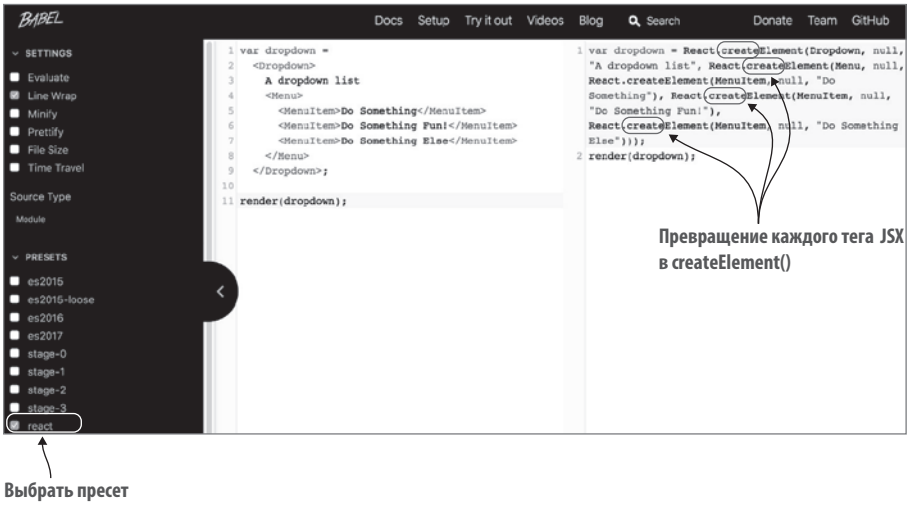
ПРИМЕЧАНИЕ Запустите приложение и откройте вкладку Elements в инструментах разработчика, чтобы увидеть исполняемое содержимое index.html.

ПРИМЕЧАНИЕ Файл serviceWorker.ts сгенерирован просто на случай, если вы захотите разработать прогрессивное веб-приложение (PWA), которое может запускаться офлайн при помощи кэшированных ресурсов. В наших примерах приложений мы это использовать не будем.

JSX И TSX

Спецификация JSX (https://facebook.github.io/jsx) приводит такое определение: «JSX — это XML-подобное синтаксическое расширение ECMAScript, не имеющее определенной семантики. Оно не предназначено для реализации в движках или браузерах».

Расшифровывается JSX как JavaScript XML. Оно определяет набор XML-тегов, которые могут быть вложены в код JS. Для отображения в браузере эти теги могут быть считаны и преобразованы в обычные HTML-теги, в React для этого предусмотрен парсер. В главе 6 мы показали компилятор Babel REPL (https://babeljs.io/repl), а на следующем рисунке показан скриншот этого компилятора с примером JSX.



Парсинг JSX в Babel

Слева мы выбрали пресет React и вставили пример кода из спецификации JSX. Этот пресет указывает, что мы хотим преобразовать каждый JSX-тег в вызов `React.createElement()`. Образец кода должен отображать выпадающее меню, содержащее три элемента. Справа вы можете видеть, как JSX был преобразован в JavaScript.

Каждое приложение React имеет минимум один компонент — корневой. Наше сгенерированное приложение как раз и имеет только корневой компонент App. Файл с кодом функции App имеет расширение .tsx, что говорит компилятору TS, что он содержит JSX. Но tsc недостаточно наличия одного только расширения .tsx для его обработки. Вам нужно включить поддержку JSX, добавив опцию компилятора jsx. Для этого откройте файл tsconfig.json, и вы найдете в нем следующую строку:

```
"jsx": "preserve"
```

Опция jsx влияет только на стадию генерации, проверка типов остается незатронутой. Значение preserve дает tsc команду скопировать jsx-часть в выводимый файл, изменив его расширение на .jsx, так как его будет считывать еще один процесс (например, Babel). Если бы это значение было react, tsc преобразовал бы JSX-теги в вызовы `React.createElement()`, как можно видеть на предыдущем изображении справа.

Как вы заметили, некоторые файлы имеют необычное расширение: .tsx. Если бы мы писали код в JS, то CLI сгенерировал бы файл с расширением .jsx (не .tsx). JSX и TSX объясняются во врезке «JSX и TSX».

Компонент React может быть объявлен как функция или класс. Функциональный (основанный на функции) компонент реализуется как функция и структурируется следующим образом (типы опущены):

Листинг 13.3. Функциональный компонент

```
const MyComponent = (props) => {   <-- props используется для передачи данных компонентам

    return (
        <div>...</div>   <-- Возвращает JSX компонента
    )

    // Здесь можно разместить другие функции.
}

export default MyComponent;
```

Разработчики, предпочитающие работать с классами, могут создать компоненты, основанные на них, которые будут реализованы в качестве подклассов `React.Component`. В следующем примере показано, как они структурируются:

Листинг 13.4. Компонент, основанный на классе

```
class MyComponent extends React.Component {   <-- Класс должен наследовать от React.Component

    render() {   <-- Метод render() вызывается React
        return (   <-- Возвращает JSX для отображения
            <div>...</div>
        );
    }

    // Здесь можно разместить и другие методы.
}

export default MyComponent;
```

Функциональный компонент просто возвращает JSX, но компонент, основанный на классе, должен включать метод `render()`, возвращающий JSX. Мы

предпочитаем использовать функциональный вариант, который имеет ряд преимуществ перед классовым:

- Для написания функции требуется меньше кода и нет необходимости наследовать код компонента от класса.
- Функциональный компонент генерирует меньше кода в процессе компиляции Babel, и минификаторы кода лучше выводят код, сокращая имена переменных более эффективно, поскольку все они локальны для функции в отличие от членов класса, которые считаются публичным API и не могут быть переименованы.
- Функциональные компоненты не нуждаются в ссылке `this`.
- Функции легче тестировать, чем классы; утверждения просто отображают props в возвращаемый JSX.

ПРИМЕЧАНИЕ Основанные на классах компоненты следует применять, только если вы вынуждены использовать версию React старше 16.8. В таких старых версиях только основанные на классах компоненты будут поддерживать методы состояния и жизненного цикла.

Если вы используете текущую версию Create React App с опцией `--typescript`, тогда сгенерированный файл App.tsx будет уже включать шаблонный код для функционального компонента (функцию типа `React.FC`), как показано в следующем листинге.

Листинг 13.5. Файл App.tsx

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

const App: React.FC = () => {
    return (
        <div className="App">
            <header className="App-header">
                <img src={logo} className="App-logo" alt="logo" />
                <p>
                    Edit <code>src/App.tsx</code> and save to reload.
                </p>
                <a
                    className="App-link"
                    href="https://reactjs.org"
                    target="_blank"
                    rel="noopener noreferrer"
                >
                    Learn React
                </a>
            </header>
```

Импортирует библиотеку React

Определяет функциональный компонент

Возвращает шаблон компонента в виде выражения JSX (это не строка)

В JSX вместо CSS селектора «class» используйте className, чтобы избежать конфликтов с ключевым словом JS «class»

```
        </div>
    );
}

export default App;
```

Экспортирует объявление компонента App, чтобы он мог использоваться в других модулях

ПРИМЕЧАНИЕ Мы использовали Create react App версии 3.0. Более старые версии этого инструмента генерировали бы компонент `App`, основанный на классе.

Сгенерированная функция App возвращает разметку (или шаблон), который React использует для отображения UI этого компонента, показанный ранее на рис. 13.3. В процессе сборки Babel преобразует эту разметку в чистый объект JS — `JSX.element` с контейнером `<div>`, который обновит виртуальную DOM и DOM браузера (виртуальную DOM мы рассмотрим в разделе 13.5). Этот компонент App не имел отдельного места для хранения его данных (своего состояния), поэтому мы добавим его в следующем разделе.

13.3. УПРАВЛЕНИЕ СОСТОЯНИЕМ КОМПОНЕНТА

Состояние компонента является хранилищем данных, которые этот компонент должен отображать. Эти данные сохраняются в `state` компонента, даже если React повторно отображает компонент. Если у вас есть компонент `Search`, его состояние может хранить последний поисковый критерий и последний поисковый результат. Каждый раз, когда код обновляет состояние компонента, React обновляет его UI для отражения изменений, вызванных действиями пользователя (например, кликами по кнопкам или набором текста в полях ввода), а также другими событиями.

ПРИМЕЧАНИЕ Не стоит путать индивидуальное состояние компонента с состоянием приложения. Состояние приложения хранит данные, которые могут поступать от нескольких компонентов, функций или классов.

Как же вы можете определять и обновлять состояние компонента? Это зависит от того, как этот компонент был создан изначально. Мы собираемся на минуту вернуться к компонентам, основанным на классах, чтобы вы могли понять разницу в работе с состоянием в функциональных и классовых компонентах. Затем мы вернемся к рекомендуемым нами функциональным компонентам.

13.3.1. Добавление состояния в компоненты, основанные на классе

Если вам нужно работать с подобным компонентом, вы можете определить тип, представляющий состояние, создать и инициализировать объект этого типа, а затем при необходимости обновить его при помощи вызова `this.setState(…)`.

Давайте рассмотрим простой, основанный на классе компонент, имеющий объект состояния с двумя свойствами: имя пользователя и изображение. Для получения изображений мы будем использовать сайт Lorem Picsum, который возвращает случайные изображения обозначенного размера. Например, если вы введете URL https://picsum.photos/600/150, то браузер покажет случайное изображение размером 600 × 150 пикселов. В следующем листинге показан подобный основанный на классе компонент с объектом состояния, имеющим два свойства.

Листинг 13.6. Основанный на классе компонент с состоянием

```
interface State {  ◄—— Определяет тип для состояния компонента
    userName: string;
    imageUrl: string;
}
+
export default class App extends Component {

    state: State = { userName: 'John',  ◄—— Инициализирует объект State
             imageUrl: 'https://picsum.photos/600/150' };
    render() {
        return (
            <div>
                <h1>{this.state.userName}</h1>  ◄—— Отображает здесь userName
                <img src={this.state.imageUrl} alt=""/> ◄—— Отображает здесь imageUrl
            </div
        );
    }
}
```

Глядя на код метода `render()`, вы, вероятно, можете догадаться, что этот компонент будет отображать `"John"` и изображение. Обратите внимание, что мы вложили значения свойств состояния в JSX, поместив их в фигурные скобки: `{this.state.userName}`.

Любой основанный на классе компонент наследуется от класса `Component`, имеющего свойство `state` и метод `setState()`. Если нужно изменить значение любого свойства состояния, следует сделать это при помощи упомянутого метода:

```
this.setState({userName: "Mary"});
```

Вызывая `setState()`, вы даете React понять, что может потребоваться обновление UI. Если вы обновите состояние напрямую (например, при помощи `this.state.userName='Mary'`), React не станет вызывать для обновления UI метод `render()`. Как вы могли догадаться, свойство `state` объявлено в базовом классе `Component`.

В разделе 13.2 мы перечислили преимущества функциональных компонентов над классовыми и с этого момента больше не будем использовать последние. В функциональных компонентах мы управляем состоянием при помощи *хуков*, которые были добавлены в React версии 16.8.

13.3.2. Использование хуков для управления состоянием

Как правило, хуки позволяют «присоединять» поведение к функциональному компоненту без необходимости написания классов, создания оберток или использования наследования. В этом случае вы как бы говорите функциональному компоненту: «Я хочу, чтобы ты имел дополнительную функциональность, оставаясь при этом простой функцией».

Имена хуков должны начинаться со слова `use` — так Babel обнаруживает их и отличает от обычных функций. Например, `useState()` является хуком для управления состоянием компонента, а `useEffect()` используется для добавления побочного эффекта поведения (вроде получения данных с сервера). В этом разделе мы сосредоточимся на хуке `useState()`, используя все тот же пример из предыдущего раздела: компонент, чье состояние представлено именем пользователя и URL изображения. Но на этот раз компонент будет функциональный.

Хук `useState()` может создавать примитивное значение или сложный объект и хранить его между вызовами компонента. Следующая строка показывает, как вы можете определить состояние для имени пользователя:

```
const [userName, setUserName] = useState('John');
```

Функция `useState()` возвращает пару: значение текущего состояния и функцию, позволяющую его обновлять. Помните ли вы синтаксис для деструктуризации массива, появившийся в ECMAScript 6? (Если нет, загляните в раздел A.8.2 приложения). Предыдущая строка означает, что хук `useState()` получает в качестве начального значения строку `'John'` и возвращает массив, а мы используем деструктуризацию для получения двух элементов этого массива и задействования их в роли двух переменных: `userName` и `setUserName`. Если вы захотите обновить значение `userName` с `'John'` на `'Mary'` и заставить React обновить UI (при необходимости), можете сделать так:

```
setUserName('Mary');
```

> **ПРИМЕЧАНИЕ** В своей IDE зажмите Cmd (для Mac) или Ctrl (для Windows) и щелкните по `useState()`. Таким образом вы откроете определение типов для этой функции, где объявлено, что она возвращает значение состояния и функцию для его обновления. `useState()` не является чистой функцией, поскольку где-то внутри React она хранит состояние компонента, в связи с чем мы зовем ее функцией с побочными эффектами.

В следующем листинге показан функциональный компонент, который хранит состояние в двух примитивах: `userName` и `imageUrl`, отображая их значения при помощи JSX:

Листинг 13.7. Функциональный компонент, использующийся для хранения состояния примитивы

```
import React, {useState} from 'react';

const App: React.FC = () => {

    const [userName, setUserName] = useState('John');
    const [imageUrl, setImageUrl] = useState('https://picsum.photos/600/150');

    return (
        <div>
            <h1>{userName}</h1>
            <img src={imageUrl} alt=""/>
        </div>
    );
}

export default App;
```

Импортирует хук useState

Определяет состояние userName

Определяет состояние imageUrl

Отображает значение переменной состояния userName

Отображает значение переменной состояния imageUrl

Теперь давайте перепишем предыдущий компонент, чтобы вместо двух примитивов он объявлял свое состояние в виде объекта с двумя свойствами: `userName` и `imageUrl`. Следующий листинг объявляет интерфейс `State` и использует хук `useState()` для работы с объектом, имеющим тип `State`.

Листинг 13.8. Использование объекта для хранения состояния

```
import React, {useState} from 'react';

interface State {
    userName: string;
    imageUrl: string;
}

const App: React.FC = () => {

    const [state, setState] = useState<State>({
        userName: 'John',
        imageUrl: 'https://picsum.photos/600/150'
    });

    return (
        <div>
            <h1>{state.userName}</h1>
            <img src={state.imageUrl} alt=""/>
        </div>
    );
}

export default App;
```

Определяет тип для состояния компонента

Определяет и инициализирует объект состояния

Отображает значение свойства состояния userName

Отображает значение свойства состояния imageUrl

Обратите внимание, что `useState()` является обобщенной функцией, и в процессе ее вызова мы передали конкретный тип `State`.

Исходный код этого приложения расположен в директории hello-world. Выполните команду `npm start`, и браузер отобразит окно, похожее на рис. 13.5 (изображение может отличаться).

Рис. 13.5. Отображение имени пользователя и изображения

Имя пользователя и изображение расположены слишком близко к левой границе окна, но это легко исправить с помощью CSS. Сгенерированное приложение, показанное в листинге 13.5, имеет отдельный файл app.css с CSS-селекторами, примененными к компоненту посредством атрибута `className` (вы не можете использовать `class`, так как он будет конфликтовать с ключевым словом JavaScript `class`. На этот раз мы добавим `margin` (отступ), объявив объект JS со стилями и использовав его в JSX. В следующем листинге мы добавили переменную `myStyles` и использовали ее в JSX компонента.

Листинг 13.9. Добавление стилей в компонент

```
const App: React.FC = () => {

    const [state, setState] = useState<State>({
        userName: 'John',
        imageUrl: 'https://picsum.photos/600/150'
    });

    const myStyles = {margin: 40};    ◄—— Объявляет стили

    return (
        <div style ={myStyles}>  ◄—— Применяет стили
            <h1>{state.userName}</h1>
            <img src={state.imageUrl} alt=""/>
        </div>
    );
}
```

Обратите внимание, что свойство `style` строго типизировано, что поможет нам в проверке свойств CSS. Это одно из преимуществ JSX перед простым HTML — JSX будет преобразован в JS, и TS добавит строгую типизацию в HTML и CSS-элементы через файлы определений.

С добавленным отступом браузер будет отображать `<div>`, оставляя вокруг него дополнительно 40 пикселов пространства, как показано на рис. 13.6.

Рис. 13.6. Добавление отступа

Наше первое приложение на React работает и выглядит неплохо. Оно использует функциональный компонент, который хранит жестко закодированные данные в объекте состояния и отображает данные при помощи JSX. Это хорошее начало, и в следующем разделе мы начнем написание нового приложения с уже более богатой функциональностью.

13.4. РАЗРАБОТКА МЕТЕОПРИЛОЖЕНИЯ

В этом разделе мы разработаем приложение, в котором пользователь сможет вводить название города и получать текущие данные о погоде. Создавать это приложение мы будем поэтапно:

1. Добавим HTML из компонента `App`, куда пользователь сможет вводить название города.
2. Добавим код для получения реальных данных с метеосервера, и компонент `App` будет их отображать.

3. Создадим еще один компонент — `WeatherInfo`, который будет потомком `App`. `App` будет извлекать метеоданные и передавать их `WeatherInfo` для отображения.

Мы будем получать данные из метеослужбы http://openweathermap.org, которая предоставляет API для совершения запросов о погоде во множестве городов по всему миру. В ответ на запрос эта служба возвращает информацию в виде строки формата JSON. Например, для получения данных о текущей температуре воздуха в Лондоне по Фаренгейту (`units=imperial`) URL будет выглядеть так: `http://api.openweathermap.org/data/2.5/find?q=London&units=imperial&appid=12345`. (Создатели этой метеослужбы требуют, чтобы вы получили ID приложения, что достаточно просто сделать. Если вы хотите запустить наше метеоприложение, запросите ID и замените `12345` в предыдущем URL на его полученное в ответ значение.)

Образец кода для этой главы включает и это приложение, расположенное в директории weather, которая была сгенерирована с использованием следующей команды:

```
create-react-app weather --typescript
```

Затем мы заменили JSX-код в файле app.tsx на простую HTML-форму, в которой пользователь мог бы вводить имя города и нажимать на кнопку `Get Weather`. Кроме этого, введенный город представляет состояние этого компонента, и компонент `App` будет обновлять его состояние при вводе пользователем названия.

13.4.1. Добавление хука состояния в компонент App

Первая версия нашего компонента `App` определяет его состояние при помощи хука `useState()` следующим образом:

```
const [city, setCity] = useState('');
```

Значение переменной `city` должно обновляться функцией `setCity()`. Наш хук `useState()` инициализирует переменную `city` с пустой строкой, поэтому TS выведет тип `city` как `string`. В листинге 13.10 показан компонент `App` с объявленным состоянием и формой, определенной в разделе `JSX`. Этот код также имеет обработчик событий, `handleChange()`, который вызывается каждый раз, когда пользователь вводит или обновляет символы в поле ввода.

Листинг 13.10. Файл App.tsx в метеоприложении

```
import React, { useState, ChangeEvent } from 'react';

const App: React.FC = () => {
```

```
    const [city, setCity] = useState('');

    const handleChange = (event: ChangeEvent<HTMLInputElement>) => {
        setCity(event.target.value);
    }

    return (
        <div>
            <form>
                <input type="text" placeholder="Enter city"
                    onChange = {handleChange} />
                <button type="submit">Get weather</button>
                <h2>City: {city}</h2>
            </form>
        </div>
    );
}

export default App;
```

Объявляет состояние города

Объявляет функцию для обработки событий поля ввода

Обновляет состояние вызовом setCity()

Присваивает обработчик атрибуту onChange

Отображает текущее значение состояния

Поле ввода определяет обработчик событий: `onChange = {handleChange}`. Обратите внимание, что мы не вызывали здесь `handleClick()`, а просто передали имя этой функции. `onChange` в React ведет себя как `onInput` и срабатывает, как только содержимое поля ввода изменяется. Когда пользователь вводит (или изменяет) символ в этом поле, вызывается функция `handleChange()`, которая обновляет состояние и тем самым вызывает обновление UI.

> **ПРИМЕЧАНИЕ** Нигде не описано, какие типы событий React вы можете использовать с конкретными элементами JSX. Чтобы избежать использования в качестве аргумента в функциях обработки событий `event: any`, откройте файл index.d.ts в директории node_modules/@types/react и найдите в нем «Event Handler Types». Это должно помочь вам выяснить, что подходящим типом для события `onChange` является обобщенный тип `ChangeEvent<T>`, который получает в качестве параметра тип конкретного элемента, например `ChangeEvent<HTMLInputElement>`.

Чтобы продемонстрировать обновление состояния, мы добавили элемент `<h2>`, который показывает его текущее значение: `<h2>Entered city: {city}</h2>`. Обратите внимание, что для повторного отображения текущего значения `city` нам не понадобилось писать jQuery-подобный код для нахождения ссылки на этот элемент `<h2>` и изменения его значения напрямую. Вызов `setCity(event.target.value)` вынуждает React обновить соответствующий узел в DOM.

В большинстве случаев, если вам нужно обновить состояние функционального компонента, делать это следует только при помощи подходящей функции `setXXX()`, возвращаемой хуком `useState()`. Вызывая `setXXX()`, вы даете React понять, что может потребоваться обновление UI. Если вы обновите состояние

напрямую (например, указав `city="London"`), React не обновит UI. Он может пакетировать обновление UI перед сверкой виртуальной DOM с DOM браузера. На рис. 13.7 показан скриншот, сделанный после того, как пользователь набрал в поле ввода *Londo*.

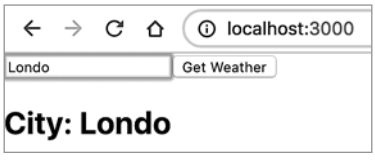


Рис. 13.7. После набора пользователем Londo

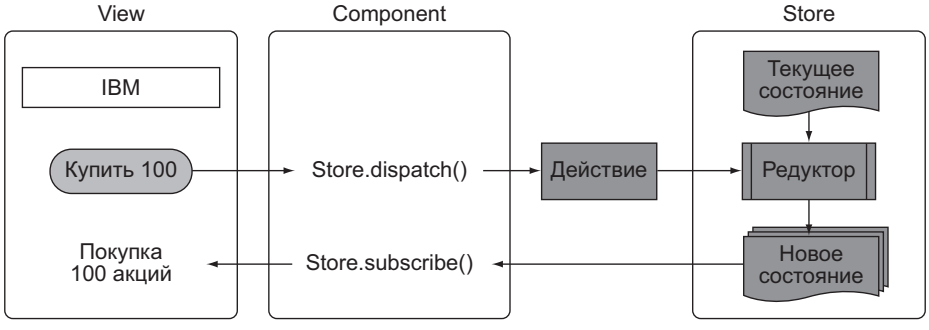
REDUX И УПРАВЛЕНИЕ СОСТОЯНИЕМ ПРИЛОЖЕНИЯ

Хук `useState()` в функциональных компонентах (или метод `setState()` в компонентах на основе классов) используется для хранения внутренних данных компонента и их синхронизации с UI, но при этом приложение в целом может также нуждаться в хранении и обслуживании данных, используемых множеством компонентов или относящихся к текущему состоянию UI (например, пользователь выбрал компонент `X` в компоненте `Y`). В приложениях React наиболее популярной JS-библиотекой для управления состояниями является Redux (вы также можете предпочесть использовать другую известную библиотеку — MobX). Redux основана на следующих принципах:

- *Единый источник истины* — используется одно хранилище данных, содержащее состояние вашего приложения.
- *Состояние только для чтения* — когда действие отправлено, функция-редуктор дублирует текущее состояние и обновляет дублированный объект на основе этого действия.
- *Изменения состояния производятся чистыми функциями* — функции-редукторы получают действие и объект текущего состояния, а возвращают новое состояние.

В Redux поток данных однонаправленный:

1. Компонент приложения отправляет действие в хранилище.
2. Редуктор (чистая функция) получает объект текущего состояния, а затем дублирует, обновляет и возвращает его.
3. Компонент подписывается на хранилище, получает объект нового состояния и обновляет UI согласно ему.



Поток данных Redux

Приведенное изображение показывает однонаправленный поток данных в Redux.

Пользователь щелкает по кнопке, чтобы купить 100 акций IBM. Функция обработки клика вызывает метод `dispatch()`, генерируя действие, которое является JS-объектом и имеет свойство `type`, описывающее, что произошло в приложении (например, пользователь хочет купить акции IBM). Помимо свойства `type`, объект действия может при необходимости иметь другое свойство с полезными данными, хранящимися в состоянии приложения:

```
{
    type: 'BUY_STOCK',  ◄—— Тип действия
    stock: {symbol: 'IBM', quantity: 100}  ◄—— Полезные данные действия
}
```

Этот объект только описывает действие и предоставляет полезную нагрузку, но ничего не знает о том, как должно изменяться состояние. А кто же знает? Знает об этом редуктор — чистая функция, определяющая, как должно изменяться состояние приложения. При этом редуктор никогда не изменяет текущее состояние, но создает новую его версию и возвращает на нее новую ссылку. Как видно из предыдущего рисунка, компонент подписывается на изменения и соответствующим образом обновляет UI.

Редуктор не реализует логику приложения, требующую взаимодействия с внешними сервисами (например, размещение заказа). Эти функции просто обновляют и возвращают состояние приложения на основе действия и его полезной нагрузки, если таковая присутствует. Реализация логики приложения потребовала бы взаимодействия со средой за пределами редуктора, порождая побочные эффекты, но чистые функции побочных эффектов иметь не могут. Более подробно об этом можете прочитать в документации Redux на GitHub http://mng.bz/Q0lv.

ПРИМЕЧАНИЕ Чтобы увидеть, что React обновляет в DOM только узел `<h2>`, запустите приложение (командой `npm start`), открыв инструменты разработчика на вкладке Elements. Расширьте дерево DOM, чтобы содержимое элемента `<h2>` стало видимым, и начните печатать текст в поле ввода. Вы увидите, что браузер изменяет только содержимое элемента `<h2>`, при этом остальные элементы остаются неизменными.

Работа с состоянием компонента является внутренней функцией компонента. Но в определенный момент компоненту может потребоваться начать работать с внешними данными, и именно тогда в дело вступит хук `useEffect()`.

13.4.2. Получение данных при помощи хука useEffect в компоненте App

Вы научились хранить название города в состоянии компонента `App`, но наша конечная цель — узнать погоду в заданном городе, получив данные с внешнего сервера. Говоря на языке функционального программирования, нам нужно написать функцию с *побочными эффектами*. В отличие от *чистых функций*, такие функции используют внешние данные, и каждый вызов может производить разные результаты, даже если аргументы функции остаются теми же.

В функциональных компонентах React для реализации функциональности с побочными эффектами мы будем использовать хук `useEffect()`. React по умолчанию автоматически вызывает функцию обратного вызова, переданную в `useEffect()`, после каждого отображения DOM. Давайте добавим в компонент App из листинга 13.10 следующую функцию:

```
useEffect(() => console.log("useEffect() was invoked"));
```

Если вы запустите приложение с открытой консолью браузера, то будете видеть сообщение `"useEffect() was invoked"` (была вызвана `useEffect()`) при наборе каждого символа в поле ввода и обновлении UI. Каждый компонент React проходит через ряд событий жизненного цикла, и если вам нужно, чтобы код выполнялся после добавления компонента в DOM или при каждом его повторном отображении, то хук `useEffect()` как раз подойдет для размещения такого кода. Но если же вы хотите, чтобы код в `useEffect()` выполнялся только один раз после начального отображения, определите в качестве второго аргумента пустой массив:

```
useEffect(() => console.log("useEffect() was invoked"), []);
```

Код в предыдущем хуке будет выполнен один раз после отображения компонента, что делает его удачным местом для выполнения начального получения данных.

Давайте предположим, что вы живете в Лондоне и хотите видеть погоду в этом городе сразу после запуска приложения. Начните с инициализации состояния `city` как `"London"`:

```
const [city, setCity] = useState('London');
```

Теперь вам нужно написать функцию, получающую данные для указанного города. URL будет включать следующие статические части (замените `12345` на ваш ID приложения):

```
const baseUrl = 'http://api.openweathermap.org/data/2.5/weather?q=';
const suffix = "&units=imperial&appid=12345";
```

Между этими строками вам нужно поместить название города, и в итоге полный URL схематично будет выглядеть так:

```
baseUrl + 'London' + suffix
```

Для совершения Ajax-запросов мы будем использовать API Fetch браузера (см. документацию к Mozilla по ссылке http://mng.bz/Xp4a). Функция `fetch()` возвращает `Promise`, и в методе `getWeather()` мы будем использовать ключевые слова `async/await` (см. раздел A.10.4 приложения).

Листинг 13.11. Получение данных о погоде

```
const getWeather = async (city: string) => {
    const response = await fetch(baseUrl + city + suffix);
    const jsonWeather = await response.json();
    console.log(jsonWeather);
}
```

Делает асинхронный вызов сервера погоды

Преобразует ответ в формат JSON

Выводит в консоль данные о погоде в формате JSON

> **ПРИМЕЧАНИЕ** Мы предпочитаем использовать для асинхронного кода ключевые слова `async/await`, но использование промисов с цепными вызовами `.then()` здесь также подойдет.

Когда вы используете стандартный метод браузера `fetch()`, получение данных становится двухэтапным процессом: сначала вы получаете ответ, а затем вызываете функцию `json()` для объекта ответа, чтобы получить фактические данные.

> **ПРИМЕЧАНИЕ** JavaScript-разработчики зачастую для обработки HTTP-запросов используют сторонние библиотеки. Одной из наиболее популярных является основанная на промисах библиотека Axios (www.npmjs.com/package/axios).

Теперь вы можете использовать эту функцию для начального получения данных в `useEffect()`:

```
useEffect( () => getWeather(city), []);
```

Если вы хотите, чтобы код в `useEffect()` выполнялся только при изменении конкретной переменной состояния, тогда можете прикрепить хук к ней. Например, можно указать, что `useEffect()` должен выполняться только при обновлении `city`:

```
useEffect(() => console.log("useEffect() was invoked"),
                ['city']);
```

Текущая версия компонента `App` показана в следующем листинге.

Листинг 13.12. Получение данных о погоде в Лондоне в useEffect()

```
import React, { useState, useEffect, ChangeEvent } from 'react';

const baseUrl = 'http://api.openweathermap.org/data/2.5/weather?q=';
const suffix = "&units=imperial&appid=12345";

const App: React.FC = () => {

    const [city, setCity] = useState('London');

    const getWeather = async (city: string) => {
        const response = await fetch(baseUrl + city + suffix);
        const jsonWeather = await response.json();
        console.log(jsonWeather);
      }

      useEffect( { () => getWeather(city) }, []);

    const handleChange = (event: ChangeEvent<HTMLInputElement>) => {
      setCity( event.target.value );
}

    return (
        <div>
            <form>
                <input type="text" placeholder="Enter city"
                       onInput = {handleChange} />
                <button type="submit">Get Weather</button>
                <h2>City: {city}</h2>
            </form>
        </div>
    );
}

export default App;
```

Асинхронно получает данные о погоде для указанного города

Пустой массив означает, что этот хук должен быть запущен всего один раз

Обновляет состояние

Вторым аргументом `useEffect()` является пустой массив, поэтому `getWeather()` будет вызван всего один раз после первичного отображения компонента `App`.

Запустите это приложение при открытой консоли браузера, и оно выведет извлеченный JSON с данными о погоде в Лондоне, как показано на рис. 13.8.

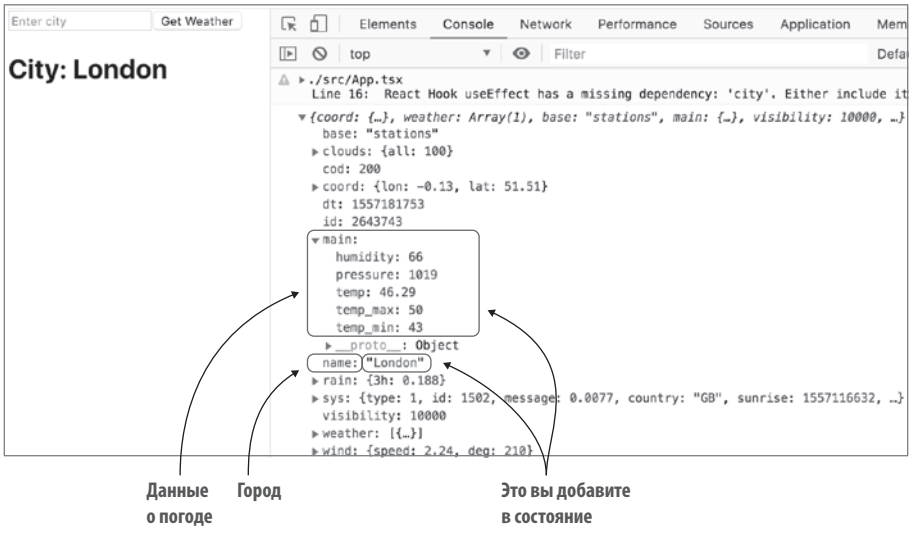


Рис. 13.8. Данные о погоде в Лондоне, представленные в консоли

ПРИМЕЧАНИЕ Если вы запустите это приложение, консоль браузера покажет следующее предупреждение: «React Hook useEffect has a missing dependency: 'city'. Either include it or remove the dependency array react-hooks/exhaustive-deps» (В хуке `useEffect` потеряна зависимость: `'city'`. Либо добавьте ее, либо удалите массив зависимостей react-hooks/exhaustive-deps). Именно поэтому внутри упомянутого хука мы используем переменную состояния `city`, которая является зависимостью и должна быть перечислена в массиве. Это не ошибка, и для простоты мы сохраним код как есть, но вам стоит помнить об этом при проектировании собственных хуков.

ПРИМЕЧАНИЕ Для более глубокого понимания хука `useEffect()` прочитайте руководство «Complete Guide to useEffect», написанное Дэном Абрамовым (*Dan Abramov*) и доступное в его блоге Overreacted по ссылке https://overreacted.io/a-complete-guide-to-useeffect.

Начальное получение данных для предустановленного города завершено, и идея хранить эти данные в состоянии компонента кажется удачной. Давайте определим новый тип `Weather` для хранения содержимого свойств `name` и `main`, оба из которых отмечены на рис. 13.8.

Листинг 13.13. Файл weather.ts

```
export interface Weather {
    city: string;          <-- Это свойство соответствует свойству name, см. рис. 13.8
    humidity: number;      |
    pressure: number;      |
    temp: number;          |  Эти значения поступают из свойства main, см. рис. 13.8
    temp_max: number;      |
    temp_min: number;      |
}
```

В компоненте App мы добавим новую переменную состояния weather и функцию для ее обновления:

```
const [weather, setWeather] = useState<Weather | null>(null);
```

Обратите внимание, что хук usestate() позволяет вам использовать обобщенный параметр для лучшей безопасности типов.

Теперь нужно обновить функцию getWeather(), чтобы она сохраняла извлеченные данные о погоде и название города в состояние компонента.

Листинг 13.14. Сохранение состояния в getWeather

```
async function getWeather(location: string) {
    const response = await fetch(baseUrl + location + suffix);
    if (response.status === 200){
        const jsonWeather = await response.json();
        const cityTemp: Weather = jsonWeather.main;
        cityTemp.city=jsonWeather.name;
        setWeather(cityTemp);
    } else {
          setWeather(null);
    }
}
```

Хранит содержимое свойства main

Хранит название города

Сохраняет данные о погоде в состояние компонента

Извлечение информации о погоде провалилось

Этот код берет объект jsonWeather.main и название города из jsonWeather.name и помещает их в переменную состояния weather.

До сих пор наша функция getWeather() вызывалась хуком useEffect() для начального извлечения данных о погоде в Лондоне. Следующим шагом будет добавление кода для вызова getWeather() в ответ на ввод пользователем любого другого города и клик по кнопке Get Weather. Как вы видели в листинге 13.12, эта кнопка является частью формы (ее тип submit), поэтому мы добавим к тегу <form> обработчик событий. Функция handleSubmit() и первая версия JSX показаны в следующем листинге.

Листинг 13.15. Обработка клика по кнопке

```
const handleSubmit = (event: FormEvent) => {
        event.preventDefault();
        getWeather(city);
    }

return (

    <div>
        <form onSubmit = {handleSubmit}>
            <input type="text" placeholder="Enter city"
```

Когда происходит клик по кнопке Submit, отправляется FormEvent

Предотвращает предустановленное поведение кнопки отправки формы

Вызывает для введенного города getWeather()

Прикрепляет к форме обработчик событий

```
                    onInput = {handleChange} />
          <button type="submit">Get Weather</button>
          <h2>City: {city}</h2>
          {weather && <h2>Temperature: {weather.temp}F</h2>}
        </form>
    </div>
);
```

Показывает полученную температуру

В React обработчики событий получают экземпляры `SyntheticEvent`, являющегося расширенной версией нативных событий браузера (подробности по ссылке https://reactjs.org/docs/events.html). `SyntheticEvent` имеет тот же интерфейс, что и нативные события браузера (вроде `preventDefault()`, но события работают идентично во всех браузерах (в отличие от нативных событий).

Чтобы передать значение параметра в `getWeather(city)`, нам не нужно искать ссылку на поле `<input>` в UI. Состояние `city` компонента было обновлено, когда пользователь напечатал название города, следовательно, значение переменной `city` уже отображено в поле `<input>`. На рис. 13.9 показан скриншот страницы после того, как пользователь ввел Miami и щелкнул по кнопке Get Weather.

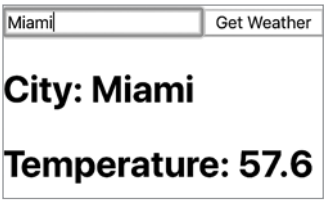


Рис. 13.9. В Майами жарко

> **ПРИМЕЧАНИЕ** В нашей книге «Angular и TypeScript. Сайтостроение для профессионалов» (СПб.: Питер, 2018)» мы также использовали сервис погоды. Вы можете найти версию этого приложения для Angular на GitHub по адресу http://mng.bz/yzMd.

А что, если пользователь введет несуществующий или неподдерживаемый на openweathermap.org город? Сервер вернет ошибку 404, и нам стоит добавить соответствующую обработку ошибок. До сих пор в случае, когда состояние weather оказывается ложным, за предотвращение отображения температуры отвечала эта строка:

```
{ weather && <h2>Temperature: {weather.temp}F</h2> }
```

В следующей версии этого приложения для проверки получения погодных данных в указанном городе мы создадим защиту типа.

Сейчас же давайте сделаем передышку и подытожим, что мы сделали в процессе разработки нашего метеоприложения:

1. Получили `Id` приложения с openweathermap.org.
2. Сгенерировали новое приложение и заместили JSX простой `<form>`.
3. Объявили состояние `city` при помощи хука `useState()`.

4. Добавили функцию `handleChange()`, обновляющую `city` при каждом изменении в поле ввода.
5. Добавили хук `useEffect()`, который будет вызываться один раз при запуске приложения.
6. Сделали так, чтобы `useEffect()` вызывал функцию `getWeather()`, использующую API `fetch()` для получения данных о погоде в Лондоне.
7. Объявили состояние `weather` для хранения полученной температуры и влажности, которые являются частью свойств полученного объекта погоды.
8. Добавили обработчик событий `handleSubmit()` для вызова `getWeather()` после того, как пользователь вводит название города и щелкает по кнопке Get Weather.
9. Изменили функцию `getWeather()`, чтобы сохранить полученные погодные данные в состоянии weather.
10. Отобразили полученную температуру на веб-странице под формой.

Все это хорошо, но не стоит размещать всю логику приложения в компоненте `App`. В следующем разделе мы создадим отдельный компонент, который будет отвечать за отображение данных о погоде.

13.4.3. Использование свойств

Приложение React — это дерево компонентов, и вам нужно решить, какие из них будут служить *контейнерами*, а какие будут отвечать за *представление*. Компонент-контейнер (иначе называемый умным компонентом) содержит логику приложения, общается с внешними поставщиками данных и передает данные своим потомкам. Как правило, контейнеры хранят состояние и содержат мало или совсем не содержат разметки.

Компонент представления (он же глупый компонент) просто получает данные от своего родителя и отображает их. Типичный компонент представления не имеет состояния и содержит большое количество разметки. Он получает данные для отображения посредством своего JS-объекта `props` (свойств).

ПРИМЕЧАНИЕ В разделе 14.4 мы рассмотрим компоненты UI блокчейн-приложения версии React. Там вы увидите один компонент-контейнер и три компонента представления.

ПРИМЕЧАНИЕ Если вы используете библиотеку для управления состоянием всего приложения (например, Redux), тогда с этой библиотекой будут взаимодействовать только контейнеры.

В нашем метеоприложении `App` является контейнером, знающим, как получать погодные данные от внешнего сервера. До сих пор этот компонент `App` отвечал также и за отображение полученной температуры, как показано на рис. 13.9. Но нам следует делегировать функциональность отображения погоды отдельному компоненту представления, например `WeatherInfo`. Любое приложение состоит из нескольких компонентов, и присутствие отдельного компонента `WeatherInfo` позволит нам продемонстрировать, как родительский компонент может отправлять данные своему потомку. Кроме того, наличие отдельного компонента `WeatherInfo`, который знает только, как отображать данные, полученные через `props`, делает его повторно используемым.

Компонент `App` (родитель) будет содержать компонент `WeatherInfo` (потомка) и должен будет передать ему полученные погодные данные. Передача данных компоненту React работает аналогично передаче данных HTML-элементам.

Мы начнем знакомство с `props`, используя в качестве примера JSX-элементы. Любой JSX-элемент может быть отображен по-разному, в зависимости от получаемых им данных. Например, JSX красной кнопки disabled может выглядеть так:

```
<button className="red" disabled />
```

Этот код дает команду React создать элемент `button` и передать ему конкретные значения через атрибуты `className` и `disabled`. React, в свою очередь, начнет с преобразования предыдущего JSX в вызов `createElement()`:

```
React.createElement("button", {
    className: "red",
    disabled: true
});
```

Затем предыдущий код произведет JS-объект:

```
{
    type: 'button',
    props: { className: "red", disabled: true }
}
```

Как вы можете видеть, `props` содержит данные, переданные в элемент React. React использует `props` для обмена данными между родительским и дочерним компонентами (предыдущая кнопка также была частью родительского компонента).

Предположим, что вы создали пользовательский компонент `Order` и добавили его в родительский JSX. Вы также могли бы передать ему данные через `props`. Например, компонент `Order` может нуждаться в получении значений таких свойств, как `operation`, `product` и `price`:

```
<Order operation="buy" product="Bicycle" price={187.50} />
```

Схожим образом мы добавим компонент `WeatherInfo` в JSX-компонент `App`, передав полученные погодные данные. Кроме того, мы обещали добавить пользовательскую защиту типа, чтобы гарантировать, что компонент `WeatherInfo` не будет ничего отображать, если данные о погоде в городе окажутся недоступны. В следующем листинге показан код из компонента `App`, определяющий и использующий защиту типа под названием `has`.

Листинг 13.16. Добавление защиты типа has

```
const has = (value: any): value is boolean => !!value;   ← Объявляет защиту типа has

...

return (
    <>   ← Пустой JSX-тег может использоваться в качестве контейнера
        <form onSubmit = {handleSubmit}>
            <input type="text" placeholder="Enter city"
                   onInput = {handleChange} />
            <button type="submit">Get Weather</button>
        </form>
        {has(weather) ? (   ← Применяет защиту типа has
            <WeatherInfo weather={weather} />   ← Передает погодные данные в WeatherInfo и отображает их
        ) : (
            <h2>No weather available</h2>   ← Отображает текстовое сообщение вместо WeatherInfo
        )}
    </>   ← Закрывает пустой JSX-тег
);
```

Как вы можете видеть, компонент `App` служит хостом для формы и компонента `WeatherInfo`, который должен отображать погодные данные. Все JSX-теги должны быть обернуты в один тег-контейнер. Ранее мы использовали в качестве родителя тег `<div>`. В листинге 13.16 мы используем вместо него пустой тег, являющийся сокращением для специального тега-контейнера `<React.Fragment>`, не добавляющего дополнительный узел в DOM.

В разделе 2.3 мы представили пользовательские защиты типов в TS. В листинге 13.16 мы объявили защиту типа `has` как функцию, чей возвращаемый тип является предикатом типа:

```
const has = (value: any): value is boolean => !!value;
```

Он получает значение типа any и применяет к нему JS-оператор двойного восклицания для проверки переданного значения на верность. Теперь выражение `has(weather)` будет проверять, была ли получена информация о погоде (как вы можете видеть в JSX листинга 13.17). Полученные погодные данные передаются компоненту `WeatherInfo` через его `props weather`:

```
<WeatherInfo weather={weather} />
```

Рассмотрим создание компонента `WeatherInfo`, получающего и отображающего погодные данные. В VS Code мы создали новый файл weather-info.tsx и объявили в нем компонент `WeatherInfo`. Как и в функциональном компоненте `App`, для `WeatherInfo` мы использовали нотацию стрелочной функции, но на этот раз наш компонент принимает явный объект `props`. Наведите курсор на FC, и вы увидите его объявление, как показано на рис. 13.10.

Обобщенный тип с параметром по умолчанию.

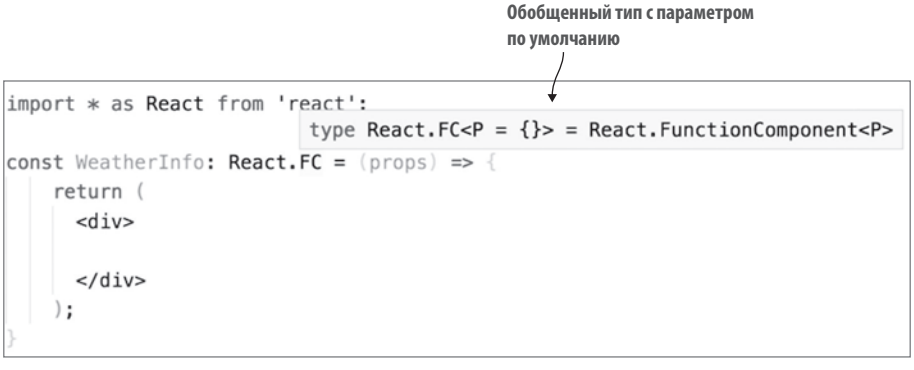


Рис. 13.10. Обобщенный тип с параметром по умолчанию

`React.FC` — это обобщенный тип, получающий в качестве параметра типа `P` (для `props`). Так почему же компилятор TS не жаловался, когда мы объявили компонент `App` без использования нотации обобщения и конкретного типа? Весь трюк заключается в `P = {}`. Так вы можете объявлять обобщенный тип со значением по умолчанию (см. врезку под названием «Значения по умолчанию для обобщенных типов» в разделе 4.2.2). Наш компонент `App` не использовал `props`, и по умолчанию React предположил, что `props` был пустым объектом.

Каждый компонент имеет свойство под названием `props`, которое может быть произвольным JS-объектом с относящимися к вашему приложению свойствами. В JavaScript обобщенные типы позволяют вам дать компоненту понять, что он получит `props`, содержащий `Weather`, как показано в листинге 13.17.

Компонент `WeatherInfo` является обобщенной функцией с одним параметром — `<P>`, как показано на рис. 13.10, а в качестве его аргумента мы использовали тип `{weather:Weather}`. Для отображения данных мы могли обратиться к каждому свойству объекта `weather`, используя точечную нотацию (например, `weather.city`), но быстрейшим способом извлечения значений из этих свойств в локальные переменные будет деструктуризация (рассмотренная в разделе A.8 приложения):

```
const {city, humidity, pressure, temp, temp_max, temp_min} = weather;
```

Листинг 13.17. WeatherInfo.tsx: функциональный компонент WeatherInfo

```
import * as React from 'react';
import {Weather} from './weather';

const WeatherInfo: React.FC<{weather: Weather} >=
    ({ weather }) => {

        const {city, humidity, pressure, temp, temp_max, temp_min} = weather;

        return (
            <div>
                <h2>City: {city}</h2>
                <h2>Temperature: {temp}</h2>
                <h2>Max temperature: {temp_max}</h2>
                <h2>Min temperature: {temp_min}</h2>
                <h2>Humidity: {humidity}</h2>
                <h2>Pressure: {pressure}</h2>
            </div>
        );
}

export default WeatherInfo;
```

Наш компонент — это обобщенная функция с аргументом типа Weather

Стрелочная функция имеет один аргумент—объект weather

Деструктуризует объект weather

Отображает город

Отображает погодные данные

Теперь все эти переменные можно использовать в JSX, возвращаемом этим компонентом (например, `<h2>City: {city}</h2>`).

Чтобы визуализировать `props` и состояние каждого компонента, установите расширение Chrome под названием React Developer Tools (инструменты разработчика React) и запустите приложение React с открытыми инструментами разработчика Chrome. Вы увидите дополнительную вкладку React, которая показывает отображенные элементы слева, а `props` и состояние (если присутствует) каждого компонента — справа (рис. 13.11). Наш компонент `WeatherInfo` не имеет состояния, в противном случае вы также увидели бы и содержимое состояния.

ПЕРЕДАЧА РАЗМЕТКИ В ДОЧЕРНИЙ КОМПОНЕНТ

`props` может использоваться не только для передачи в дочерний компонент данных, но также и для передачи JSX-фрагментов. Если вы поместите JSX-фрагмент между открывающимся и закрывающимся тегами компонента, как показано в следующем отрывке кода, тогда это содержимое будет храниться в свойстве `props.children` и вы сможете отобразить его при необходимости.

```
<WeatherInfo weather = {weather} >
    <strong>Hello from the parent!</strong>
</WeatherInfo>
```

Передает эту разметку в WeatherInfo

Здесь компонент App передает HTML-элемент `<strong>Hello from the parent! </strong>` в дочерний компонент `WeatherInfo`. По сути, он мог передать любой другой компонент React тем же способом.

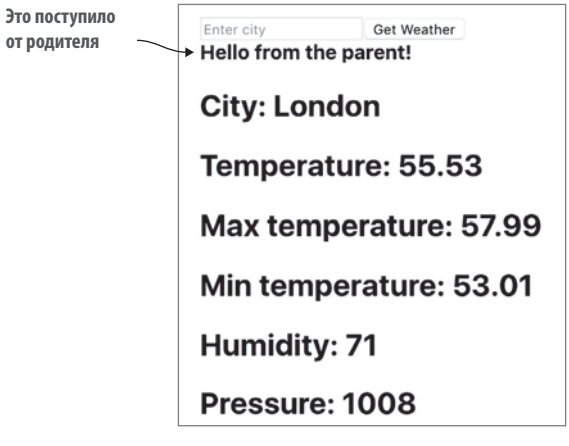
Компонент `WeatherInfo` также должен объявить свой интерес в получении не только объекта `Weather` (показанного ранее в листинге 13.13), но и содержимого `props.children` из `React.FC`:

```
const WeatherInfo: React.FC<{weather: Weather} >=
➥ ({ weather, children }) =>...
```

Эта строка сообщает React: «Мы дадим твоему компоненту `React.FC` объект со свойством `weather`, но мы бы также хотели использовать свойство `children`». Теперь локальная переменная `children` содержит переданную родителем разметку, которая может быть отображена наряду с погодными данными:

```
return (
    <div>
        {children}  ◄——  Отображает разметку, полученную от родителя
        <h2>City: {city}</h2>
        <h2>Temperature: {temp}</h2>
        { /* The rest of the JSX is omitted */ }
    </div>
);
```

После вложения выражения `{children}` компонент `WeatherInfo` будет отображен следующим образом:



Отображение содержимого, полученного через props.children

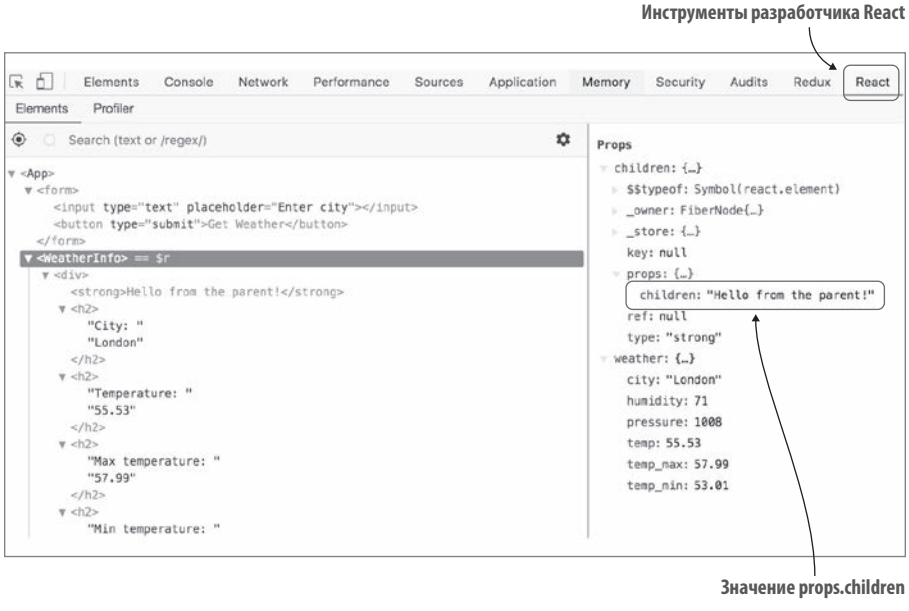


Рис. 13.11. Инструменты разработчика React

Очень хорошо, что родительский компонент передает данные своему потомку посредством `props`, но как насчет отправки данных в обратном направлении?

13.4.4. Как дочерний компонент может передавать данные родителю?

Существуют сценарии, в которых дочерний компонент должен отправлять данные своему родителю, что также выполняется при помощи `props`. Представьте дочерний компонент, подключенный к фондовой бирже, откуда он каждую секунду получает новые цены на акции. Если его родитель отвечает за обработку этих данных, то этому потомку нужна возможность их ему передавать.

Для простоты мы просто покажем, как компонент `WeatherInfo` может отправлять своему родителю текст. Это потребует написания некоторого кода как для родителя, так и для потомка. Начнем с первого.

Если родитель ожидает получения данных от потомка, то он должен объявить функцию, отвечающую за обработку этих данных. Затем в JSX родителя, куда он вкладывает дочерний компонент, мы добавляем к тегу этого компонента атрибут, содержащий ссылку на эту функцию. В нашем метеоприложении компонент

App является родителем, и мы хотим иметь возможность получать текстовые сообщения от его потомка, WeatherInfo. В следующем листинге показано, что нам нужно добавить в код компонента App, разработанного в предыдущем разделе.

Листинг 13.18. Добавление в компонент App кода для получения данных

```
const [msgFromChild, setMsgFromChild] = useState('');

const getMsgFromChild = (msg: string) => setMsgFromChild(msg);

return (
<>
    /* The rest of the JSX is omitted */

    {msgFromChild}
    {has(weather) ? (
        <WeatherInfo weather = {weather} parentChannel = {getMsgFromChild}>
</>
```

Объявляет переменную состояния msgFromChild для хранения сообщения потомка

Объявляет функцию для хранения сообщения потомка в состоянии msgFromChild

Отображает сообщение потомка

Добавляет потомку свойство parentChannel

Здесь мы добавили переменную состояния msgFromChild для хранения сообщения, полученного от потомка. Функция getMsgFromChild() получает сообщение и обновляет состояние, используя функцию setMsgFromChild(), что приводит к повторному отображению {msgFromChild} в UI компонента App.

В завершение нужно передать потомку ссылку для вызова обработчика сообщений. Мы решили назвать эту ссылку parentChannel и передали ее WeatherInfo следующим образом:

```
parentChannel = {getMsgFromChild}
```

parentChannel — это произвольное имя, которое потомок будет использовать для вызова обработчика сообщений getMsgFrom-Child().

На этом изменение кода родителя закончено, значит, пора переходить к дочернему компоненту WeatherInfo. В следующем листинге показаны дополнения в код потомка.

Листинг 13.19. Добавление кода в WeatherInfo для отправки данных родителю

```
const WeatherInfo: React.FC<{weather: Weather, parentChannel:
➥ (msg: string) => void}> =
    ({weather, children, parentChannel}) => {
/* The rest of the WebInfo code is omitted */

return (
```

Добавляет parentChannel в тип обобщенной функции

Добавляет parentChannel в аргумент функции

```
<div>
    <button
        onClick ={() => parentChannel ("Hello from child!")}>
        Say hello to parent
    </button>

/* The rest of the JSX code is omitted */
</div>
);
```

При клике по кнопке вызывает функцию родителя, используя в качестве ссылки parentChannel

Поскольку родитель сообщает потомку ссылку `parentChannel` для вызова функции обработчика событий, нам нужно изменить параметр типа компонента для добавления этой ссылки:

```
<{weather: Weather, parentChannel: (msg: string) => void}>
```

Тип `parentChannel` — это функция, получающая строчный параметр и не возвращающая значение. Наш компонент `WeatherInfo` будет обрабатывать `props`, который теперь является объектом с тремя свойствами: `weather`, `children` и `parentChannel`.

Обработчик кнопки `onClick` возвращает функцию, которая вызовет `parentChannel()`, передавая ей текстовое сообщение:

```
() => parentChannel ("Hello from child!")
```

Запустите эту версию метеоприложения, и если вы щелкнете по кнопке внизу компонента `WeatherInfo`, то родитель получит сообщение и отобразит его, как показано на рис. 13.12.

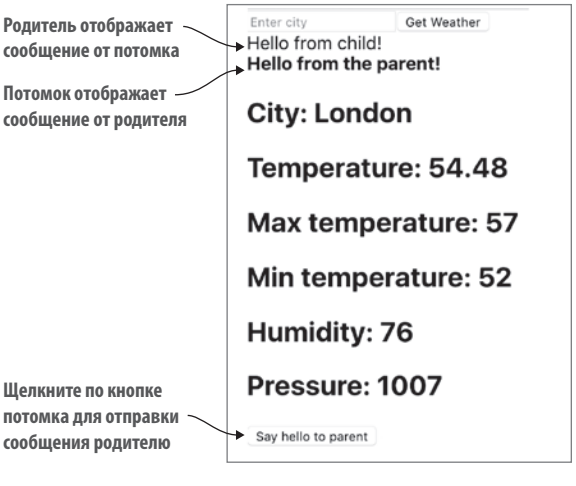


Рис. 13.12. Сообщение от потомка

ПРИМЕЧАНИЕ Код в листинге 13.19 можно оптимизировать, и это будет вашим домашним заданием. Этот код работает, но пересоздает функцию `() ? parentChannel("Hello from child!")` при каждом повторном отображении UI. Чтобы узнать, как этого избежать, прочитайте о `useCallback()` в документации React по ссылке http://mng.bz/MO8B и оберните функцию `parentChannel()` в этот хук.

Теперь, когда вы уже знакомы со `state` и `props`, мы хотели бы сделать акцент на том, что они служат разным целям:

- `state` хранит приватные данные компонента, а `props` используется для передачи данных дочернему компоненту или обратно от него родителю.
- Компонент может использовать полученный `props` для инициализации своего `state`, если таковое имеется.
- Прямое изменение значений свойств `state` запрещено. Чтобы обеспечить отражение изменений в UI, вы должны использовать функцию, возвращаемую хуком `useState()` в функциональных компонентах, или метод `setState()` в компонентах, основанных на классах.
- `props` являются неизменяемыми, и компонент не может изменить оригинальные данные, полученные через них.

ПРИМЕЧАНИЕ `State` и `props` служат разным целям, но UI в React — это функция, состоящая из того и другого.

В нашем кратком обзоре фреймворка React мы рассмотрели компоненты, состояние (`state`) и свойства (`props`), но даже самый краткий обзор React должен включать еще одну тему: виртуальная DOM.

13.5. ЧТО ТАКОЕ ВИРТУАЛЬНАЯ DOM?

Уникальная особенность React заключается в виртуальной DOM — слое между компонентом и DOM браузера. Каждый компонент состоит из элементов UI, а виртуальная DOM оптимизирует процесс отображения этих элементов в DOM браузера, как показано на рис. 13.13.

Когда вы запускаете приложение, React создает дерево UI компонентов в его собственной виртуальной DOM, которая отображает это дерево в DOM браузера. По мере работы пользователя с приложением DOM браузера активирует события. Если события обрабатываются JavaScript и если код обработки обновляет состояние компонента, то React пересоздает виртуальную DOM, отделяет новую версию от старой и синхронизирует их различия с DOM браузера. Применение такого алгоритма различения называется сверкой (подробнее см. в документации React по ссылке https://reactjs.org/docs/ reconciliation.html).

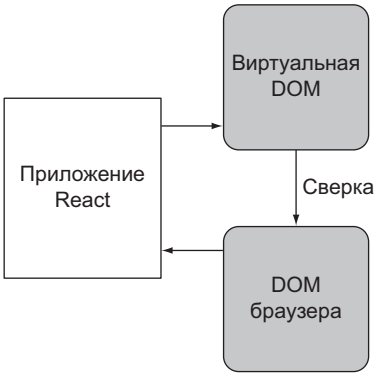


Рис. 13.13. Виртуальная DOM React

ПРИМЕЧАНИЕ Выражение «виртуальная DOM» несколько некорректно, поскольку библиотека React Native использует те же принципы для отображения iOS и UI Android, которые DOM не имеют.

В целом отображение UI из DOM браузера — это медленный процесс, и для его ускорения React не отображает повторно все элементы DOM браузера при каждом изменении элемента. Вы можете не заметить существенной разницы в скорости отображения веб-страницы, содержащей небольшое количество HTML-элементов. Но в случае со страницей, несущей тысячи таких элементов, разница в отображении, выполняемом React, в сравнении со стандартным JS весьма заметна.

Если вам сложно вообразить страницу с тысячами HTML-элементов, представьте финансовый портал, показывающий последние торговые операции в табличной форме. Если такая таблица будет состоять из 300 строк и 40 колонок, то в ней получится 12 000 ячеек, каждая из которых состоит из нескольких HTML-элементов. При этом порталу может потребоваться отображать по несколько таких таблиц.

Виртуальная DOM избавляет разработчиков от работы с API DOM браузера аналогично jQuery — просто обновите состояние компонента, и React обновит соответствующие элементы DOM наиболее эффективным способом. Это, по сути, все, что делает библиотека React, поэтому для реализации такой функциональности, как HTTP-запросы, маршрутизация или работа с формами, потребуется использовать другие библиотеки.

На этом завершается наше краткое знакомство с разработкой веб-приложений при помощи библиотеки React.js и TypeScript. В главе 14 мы рассмотрим код новой версии клиентского блокчейн-приложения, написанного на React.js.

ИТОГИ

- React.js — это отличная библиотека для отображения компонентов UI. Вы можете задействовать ее в любой части существующего приложения, использующего другие фреймворки или JavaScript. Другими словами, при желании с помощью React вы можете разрабатывать не только одностраничные приложения.
- Вы можете генерировать TS-React-приложение за одну-две минуты, используя инструмент командной строки `Create React App`. Полученное приложение будет полностью настроено и работоспособно.
- Как правило, компонент React реализуется либо в виде класса, либо как функция.
- UI-часть компонента обычно объявляется при помощи синтаксиса JSX. Компоненты React, написанные в TS, хранятся в файлах с расширением .tsx, что сообщает компилятору tsc о содержащемся в них JSX.
- Компонент React обычно имеет состояние (`state`), которое может быть представлено как одно (или более) свойство этого компонента. При каждом изменении свойства состояния React отображает компонент повторно.
- Родительский и дочерний компоненты могут обмениваться данными, используя объект свойств (`props`).
- React использует в оперативной памяти виртуальную DOM, которая служит посредником между компонентом и DOM браузера. Каждый компонент состоит из элементов UI, а виртуальная DOM оптимизирует процесс отображения этих элементов в DOM браузера.

14 Разработка блокчейн-клиента в React.js

В этой главе:

- ✓ Блокчейн-клиент, написанный на React.js.
- ✓ Обмен данными веб-клиента React.js с WebSocket-сервером.
- ✓ Запуск приложения React, работающего с двумя серверами в dev-режиме.
- ✓ Разделение UI блокчейн-клиента на компоненты и организация их связи.

В предыдущей главе вы изучили основы React, теперь же мы рассмотрим новую версию блокчейн-приложения, клиентская часть которого будет написана в React. Исходный код веб-клиента расположен в директории blockchain/client, а сервер обмена сообщениями — в соседней директории blockchain/server.

Код сервера остается таким же, как в главах 10 и 12, и функциональность этой версии блокчейн-приложения также повторяется, но вот его UI уже полностью переписан в React.

В этой главе мы не будем рассматривать функциональность блокчейна, так как это мы уже делали в предыдущих. Но тем не менее мы рассмотрим код, специфичный для библиотеки React.js. Возможно, понадобится вернуться к главе 10, чтобы получше вспомнить детали функциональности блокчейн-клиента и сервера обмена сообщениями.

Начнем мы как раз с запуска этого сервера и клиента React. Затем введем код UI компонентов, выделив различия между умными компонентами и компонентами представления. Вы также увидите много примеров кода, демонстрирующих межкомпонентное взаимодействия посредством `props`.

14.1. ЗАПУСК КЛИЕНТА И СЕРВЕРА ОБМЕНА СООБЩЕНИЯМИ

Для запуска сервера откройте терминал в директории blockchain/server, выполните команду `npm install`, чтобы установить зависимости сервера, а затем `npm start`. Вы увидите сообщение «Listening on http://localhost:3000». Оставьте сервер запущенным.

Для запуска клиента React откройте другое окно терминала в директории blockchain/client, затем также выполните `npm install`, чтобы установить React и его зависимости, а после `npm start`.

Блокчейн-клиент запустится на порте 3001, и спустя некоторое время будет сгенерирован первичный блок. На рис. 14.1 показан скриншот клиента React с ярлыками, указывающими на файлы UI приложения, среди которых вы видите App.tsx, который содержит код корневого компонента App.

Команда start определена в package.json как псевдоним команды `react-scripts start`. Она дает Webpack-команду собрать связки и использовать webpack-dev-server для запуска приложения. Мы проходили через аналогичный процесс при запуске блокчейн-клиента Angular в главе 12.

Как бы то ни было, приложения, сгенерированные CLI Create React App, предварительно настроены для запуска webpack-dev-server на порту 3000, поэтому наши порты вступили в конфликт, так как порт 3000 уже был занят сервером обмена сообщениями. Нам пришлось найти способ настроить для клиента другой порт, и вскоре мы объясним, как это сделали.

Проекты, сгенерированные инструментом Create React App, также предварительно настроены для считывания пользовательских переменных среды из файла .env.development (показан в следующем листинге) в dev-режиме или из .env.production в продакшене.

> **ПРИМЕЧАНИЕ** Подготовить оптимизированную продакшен-сборку вы можете, выполнив команду `npm run build`, которая сформирует index.html и связки приложения в директории build. Эта команда использует переменные среды из файла .env.production.

Листинг 14.1. env.development: переменные среды определены здесь

```
PORT=3001
REACT_APP_WS_PROXY_HOSTNAME=localhost:3000
```

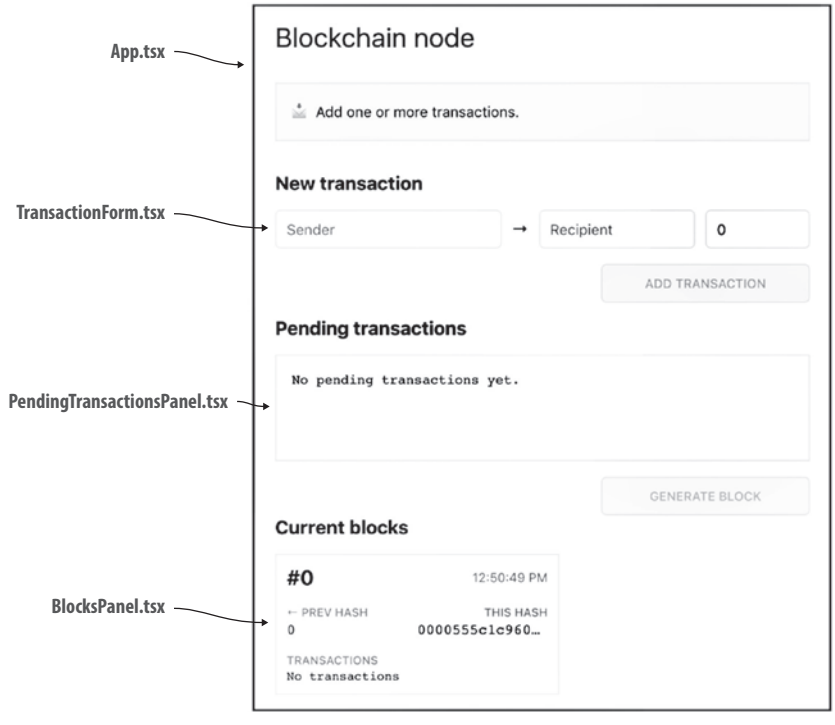


Рис. 14.1. Блокчейн-клиент запущен

В предыдущем листинге мы сначала объявили порт 3001, чтобы webpack-dev-server использовал его для запуска приложения, что разрешит конфликт, возникший с сервером, выполняемым на порте 3000. Затем мы объявили пользовательскую переменную среды `REACT_APP_WS_PROXY`, которая может использоваться для проксирования запросов клиента к серверу, выполняемому на localhost:3000. Ее мы задействуем в сценарии websocket-controller.ts. Вы можете подробнее узнать о добавлении пользовательских переменных среды в проекты `Create React App` из документации по ссылке http://mng.bz/adOm.

На рис. 14.2 показано выполнение нашего блокчейн-клиента в dev-режиме. CLI dev-сервер (на данный момент это webpack-dev-server) предоставляет приложение React на порт 3001, который затем соединяется с другим сервером, выполняемым на порте 3000.

ПРИМЕЧАНИЕ CLI Create React App позволяет вам добавлять в package.json свойство `proxy` и указывать URL для использования на случай, если dev-сервер не найдет запрашиваемый ресурс (например, proxy: http://localhost:3000). Но это свойство не работает с протоколом WebSocket, поэтому для указания URL сервера обмена сообщениями нам пришлось использовать пользовательскую переменную среды.

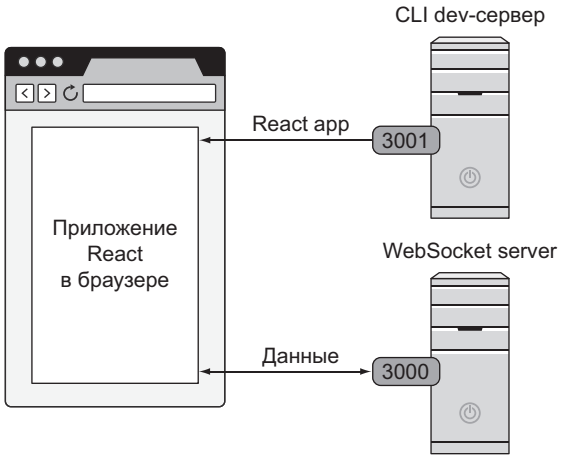


Рис. 14.2. Одно приложение, два сервера

И Angular- и React-проекты имеют схожую файловую структуру, а также процедуры запуска. Но так как наше приложение React использует в своем рабочем процессе Babel, настройка поддерживаемых браузеров не такая, как для tsc. В листинге 14.2 показано, что мы перечисляем конкретные версии браузеров, которые должно поддерживать наше приложение.

В разделе 6.4 мы показали вам, как Babel может использовать пресеты для определения, какие версии каких браузеров должно поддерживать приложение. Инструмент Create React App добавляет предустановленный раздел browserslists в package.json. Откройте этот файл, и по умолчанию вы найдете в нем следующую конфигурацию:

Листинг 14.2. Фрагмент из package.json

```
"browserslist": {
    "production": [  <-- Браузеры, которые должны поддерживаться в продакшен-сборках
        ">0.2%",
        "not dead",
        "not op_mini all"
    ],
    "development": [  <-- Браузеры, которые должны поддерживаться в dev-сборках
        "last 1 chrome version",
        "last 1 firefox version",
        "last 1 safari version"
    ]
}
```

В разделе 6.4 мы объяснили использование browserslist (см. https://browserl.ist), который позволяет вам настраивать tsc для генерации JS, выполняемого

в определенных браузерах. Create React App также его поддерживает, и используются настройки из package.json, когда вы создаете продакшен-сборку посредством запуска npm сценария build или когда создаете dev-связки выполнением сценария start. Скопируйте запись из browserslist вашего приложения на страницу https://browserl.ist, и вы увидите, какие браузеры в этой записи учтены.

ПРИМЕЧАНИЕ Несмотря на то что мы не используем tsc для компиляции кода в JavaScript, мы все равно определили в tsconfig.json опцию target: es5, чтобы TypeScript не жаловался на TS-синтаксис из нашего блокчейн-приложения. Удалите опцию target или измените ее значение на es6, и геттер get url() из листинга 14.5 будет подчеркнут красной волнистой линией в сопровождении ошибки: «Accessors are only available when targeting ECMAScript 5 and higher» (Аксессоры доступны, только когда целевая версия определена как ECMAScript 5 или выше).

Как и в предыдущих версиях блокчейн-приложения, код разделен на UI-часть и сопровождающие сценарии, реализующие алгоритмы блокчейна. В версии React UI компоненты расположены в директории components, а сопровождающие сценарии в lib.

14.2. ЧТО ИЗМЕНИЛОСЬ В ДИРЕКТОРИИ LIB

Напомним, что директория lib содержит код, который генерирует новые блоки, запрашивает длиннейшую цепочку, уведомляет остальные узлы о сгенерированных блоках и приглашает других членов блокчейна добывать блоки для заданных транзакций. Эти процессы были описаны в разделах 10.1 и 10.2.

Мы слегка изменили файл websocket-controller.ts, который содержит сценарий для связи с WebSocket-сервером. В главе 10 мы не использовали JS-фреймворки и просто инстанцировали класс WebsocketController при помощи оператора new. Мы передали messageCallback в конструктор для обработки поступающих с сервера сообщений.

В листинге 12.2 мы использовали внедрение зависимостей Angular, и этот фреймворк инстанцировал и внедрял объект WebsocketService в AppComponent. Мы могли положиться на то, что сначала Angular инстанцирует сервис и лишь затем компонент App.

ПРИМЕЧАНИЕ Если у вас есть опыт работы с Angular и вы привыкли к созданию сервисов-одиночек, которые можно внедрять в компоненты, почитайте о Context в React (https://reactjs.org/docs/context.html), который можно использовать в качестве общего хранилища данных, передаваемых от одного компонента другому. Другими словами, props — это не единственный способ передачи данных между компонентами.

В React-версии блокчейн-приложения мы вручную инстанцировали WebsocketController. Сценарий App.tsx запускается, как показано в следующем листинге.

Листинг 14.3. Инстанцирование классов до создания комопнентов

```
const server = new WebsocketController();  <-- Во-первых, инстанцирует WebsocketController
const node = new BlockchainNode();         <-- Во-вторых, инстанцирует BlockchainNode

const App: React.FC = () => {              <-- В-третьих, объявляет корневой компонент UI
    // Код компонента App опущен.
    // Мы рассмотрим его в этой же главе позже.
}
```

Чтобы гарантировать, что `WebsocketController` и `BlockchainNode` являются глобальными объектами, мы начинаем сценарий с их инстанцирования. Но `WebsocketController` нуждается в том, чтобы метод обратного вызова из компонента `App` обрабатывал сообщения сервера, изменяя состояние компонента. Проблема в том, что компонент `App` еще не инстанцирован, поэтому мы не можем передать такой обратный вызов в конструктор компонента.

Именно поэтому в `WebsocketController` мы создали метод `connect()`. Этот метод получает обратный вызов в качестве своего параметра. Полный код метода `connect()` показан в следующем листинге:

Листинг 14.4. Метод connect() из WebsocketController

```
                                  Передает обратный вызов в контроллер
connect(messagesCallback: (messages: Message) => void): <--+
                                                           |
                                                Promise<WebSocket> {
    this.messagesCallback = messagesCallback;              |
    return this.websocket = new Promise((resolve, reject) => {
        const ws = new WebSocket(this.url);  <-- Оборачивает создание сокета в промис
        ws.addEventListener('open', () => resolve(ws));
        ws.addEventListener('error', err => reject(err));
        ws.addEventListener('message', this.onMessageReceived);
    });
}
```

Компонент `App` будет вызывать `connect()`, передавая подходящий обратный вызов (который будет рассмотрен в следующем разделе и представлен в листинге 14.10). Каково же значение this.url, который должен указывать на WebSocket-сервер? В разделе 14.1 мы говорили о том, что доменное имя сервера и порт будут получены из переменных среды. В листинге 14.5 показан код получателя url в `WebsocketController`.

Переменная `REACT_APP_WS_PROXY` была определена в файле env.development, показанном в листинге 14.1. Свойство env глобальной переменной Node.js process

является тем местом, где ваш код может обратиться ко всем доступным переменным среды. Вы можете возразить, сказав, что наше приложение выполняется в браузере, а не в среде Node.js. Это верно, но в процессе сборки связок WebPack считывает значения переменных, доступных в process.env, и встраивает их в связки веб-приложения.

Листинг 14.5. Получатель url

```
private get url(): string {
    const protocol = window.location.protocol === 'https:' ? 'wss' : 'ws';
    const hostname = process.env.REACT_APP_WS_PROXY_HOSTNAME
                                || window.location.host;
    return `${protocol}://${hostname}`;
}
```

Получает имя хоста и порт из переменной среды

Если имя хоста не найдено в process.env, использует текущий хост приложения

ПРИМЕЧАНИЕ Значения пользовательских переменных среды встраиваются в связки. Выполните команду `npm run build` и откройте главную связку из директории build/static. Затем найдите значение одной из переменных из файла `env.production`, например `localhost:3002`.

После добавления метода `connect()` мы решили добавить метод `disconnect()`, который будет закрывать соединение сокета:

```
disconnect() {
    this.websocket.then(ws => ws.close());
}
```

В нашей версии блокчейн-клиента WebSocket-соединение устанавливается компонентом App, поэтому когда этот компонент уничтожается, уничтожается и все приложение, включая сокет-соединение. Но это может не всегда быть именно так, поэтому лучше иметь отдельный закрывающий сокет метод. Это позволит другим компонентам при необходимости устанавливать и уничтожать соединения.

14.3. УМНЫЙ КОМПОНЕНТ APP

UI этой версии блокчейна состоит из пяти компонентов React, расположенных в файлах .tsx директории components:

- `App`;
- `BlocksPanel`;
- `BlockComponent`;

- `PendingTransactionsPanel`;
- `TransactionForm`.

Файл App.tsx содержит корневой компонент `App`. Другие .tsx-файлы содержат код своих потомков, `TransactionForm`, `PendingTransactionPanel` и `BlocksPanel`, который также является родителем одного или более экземпляров `BlockComponent`, как показано на рис. 14.3.

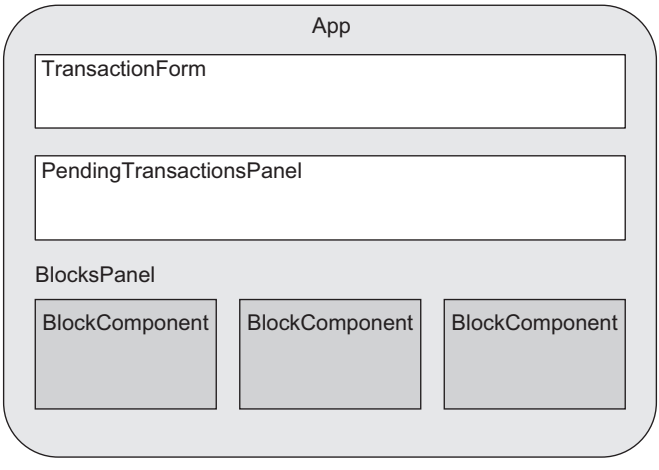


Рис. 14.3. Родительские и дочерние компоненты

В разделе 13.4.3 мы представили концепцию компонентов-контейнеров (умных) и компонентов представления. `App` является умным компонентом, который содержит ссылку на экземпляр узла блокчейн и все связанные с ним алгоритмы. Он также выполняет весь обмен данными с сервером сообщений.

Типичный компонент представления либо представляет данные, либо, исходя из действий пользователя, отправляет свои данные другим компонентам. Такие компоненты не реализуют сложную логику приложения. Например, если компонент `PendingTransactionPanel` должен инициировать создание нового блока, он просто активирует нужный обратный вызов в компоненте `App`, который начнет процесс генерации блока. В нашем блокчейн-клиенте компонентами представления являются `TransactionForm`, `PendingTransactionsPanel` и `BlocksPanel`.

Умный компонент `App` реализован в файле App.tsx, который также создает экземпляры `BlockchainNode` и `WebsocketController`. Чтобы подробно продемонстрировать вам, как компонент `App` общается со своими потомками, в следующем листинге показана его JSX-часть.

Листинг 14.6. JSX компонента App

```
const App: React.FC = () => {

    // Остальной код опущен в целях сокращения.

    return (
        <main>
            <h1>Blockchain node</h1>
            <aside><p>{status}</p></aside>
            <section>
                <TransactionForm
                    onAddTransaction={addTransaction}
                    disabled={node.isMining || node.chainIsEmpty}
                />
            </section>
            <section>
                <PendingTransactionsPanel
                    formattedTransactions={formatTransactions(
                                    node.pendingTransactions)}
                    onGenerateBlock={generateBlock}
                    disabled={node.isMining || node.noPendingTransactions}
                />
            </section>
            <section>
                <BlocksPanel blocks={node.chain} />
            </section>
        </main>
    );
}
```

Первый дочерний компонент

Этот потомок может вызвать addTransaction() в компоненте App

Второй потомок

Этот потомок может вызвать generateBlock() в компоненте App

Третий потомок

14.3.1. Добавление транзакции

JSX-компонент `App` включает дочерние компоненты, которые могут вызывать в `App` методы обратного вызова. В разделе 13.4.4 мы уже обсуждали, как дочерний компонент может взаимодействовать со своим родителем. В листинге 14.6 вы видели, что компонент `App` передает обратный вызов `onAddTransaction` в `TransactionForms` посредством `props`.

Соответствующим образом, когда пользователь щелкает по кнопке `ADD TRANSACTION` в компоненте `TransactionForm`, тот вызывает свой метод `onAddTransaction()`, что приводит к вызову метода `addTransaction()` в компоненте `App`. Вот что означает слово «неявно» на рис. 14.4, на котором приведен скриншот из VS Code со структурой компонента `App`.

Для завершения обсуждения процесса добавления транзакций давайте посмотрим, что активирует изменение UI при добавлении новой транзакции. В React для обновления UI мы изменяем состояние компонента и делаем это посредством вызова функции, возвращаемой хуком `useState()`. В компоненте `App` имя этой функции `setStatus()`.

Листинг 14.7. Код App, добавляющий транзакцию

```
const node = new BlockchainNode();  <-- Экземпляр узла

const App: React.FC = () => {
    const [status, setStatus] = useState<string>('');  <-- Объявляет состояние status компонента App

  function addTransaction(transaction: Transaction): void {
        node.addTransaction(transaction);
        setStatus(getStatus(node));  <-- Обновляет состояние
    }                                <-- Добавляет транзакцию, полученную от дочернего компонента

    // Остальная часть кода опущена в целях сокращения.
}

function getStatus(node: BlockchainNode): string {  <-- Эта функция реализуется вне компонента App
    return node.chainIsEmpty          ? '⌛ Initializing the blockchain...' :
           node.isMining              ? '⌛ Mining a new block...' :
           node.noPendingTransactions ? '✉ Add one or more transactions.' :
                                        '✓ Ready to mine a new block.';
}
```

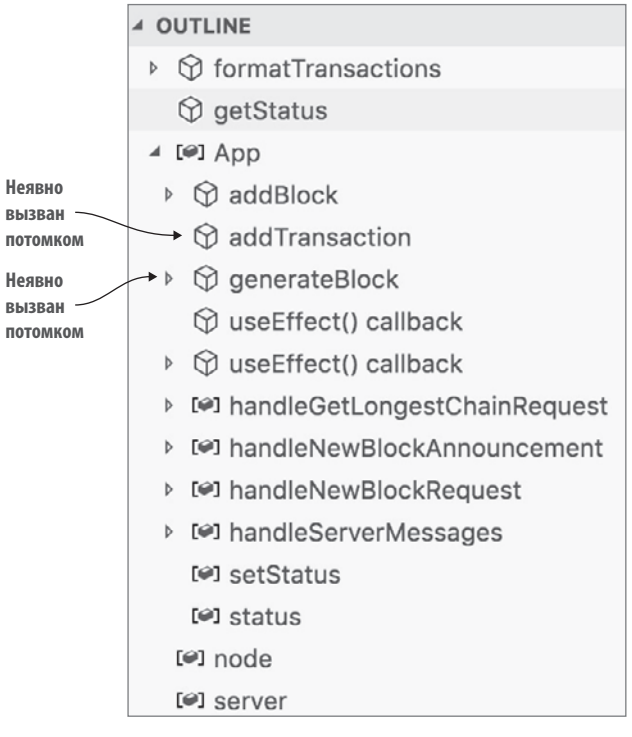


Рис. 14.4. Компонент App содержит методы, которые будут вызваны его потомками

Мы рассмотрели реализацию узла блокчейна и выявили операции, которые изменяют его внутреннее состояние. Мы добавили вспомогательные свойства `chainIsEmpty`, `isMining` и `noPendingTransactions` для определения внутреннего состояния узла. После каждой операции, которая может изменить это внутреннее состояние узла, мы сверяем с ним состояние UI, и React применяет все необходимые изменения. Если любое из этих значений изменится, нам нужно обновить UI компонента `App`. Но что мы можем использовать в компоненте `App` для активации вызова `setStatus()`?

Функция `getStatus()` возвращает текст, описывающий текущий статус узла блокчейн, и компонент `App` отображает соответствующее сообщение (см. листинг 14.7). Изначально значение статуса — это «Initializing the blockchain...» (Инициализация блокчейн...). Если пользователь добавляет новую транзакцию, `getStatus()` вернет «Add one or more transactions» и строка `setStatus(getStatus(node))`; изменит состояние компонента, приводя к повторному отображению UI. На рис. 14.5 показан статус приложения (или состояние): «Add one or more transactions».

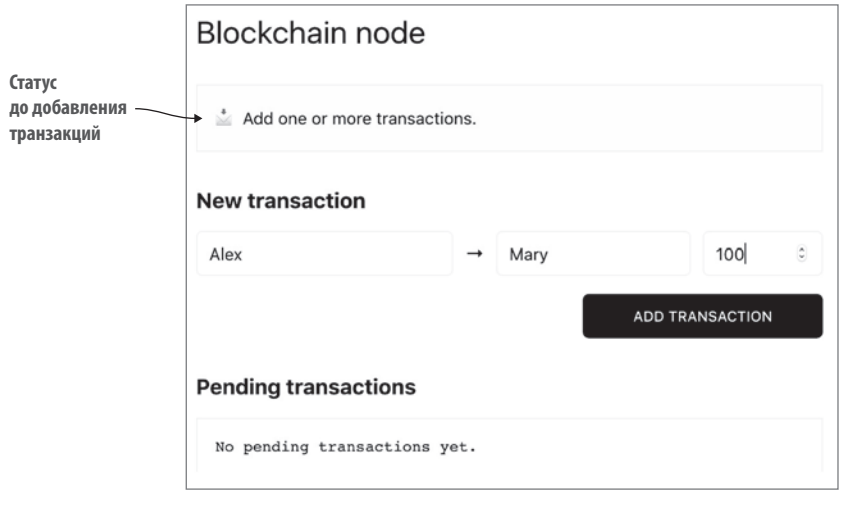


Рис. 14.5. Пользователь готов добавить транзакцию

На рис. 14.6 показан скриншот, сделанный после того, как пользователь щелкнув по кнопке `ADD TRANSACTION`. Состояние компонента `App` «Ready to mine a new block» (Готов к добыче нового блока), и кнопка `GENERATE BLOCK` активна.

В процессе тестирования этого приложения мы обнаружили один баг: если пользователь добавляет более одной транзакции, в поле `Pending Transactions` будет по-прежнему отображаться только первая добавленная транзакция. Однако после щелчка по кнопке `GENERATE BLOCK` новый блок будет содержать все

ожидающие транзакции. По изначально неясной причине React повторно отображал UI только для первой транзакции.

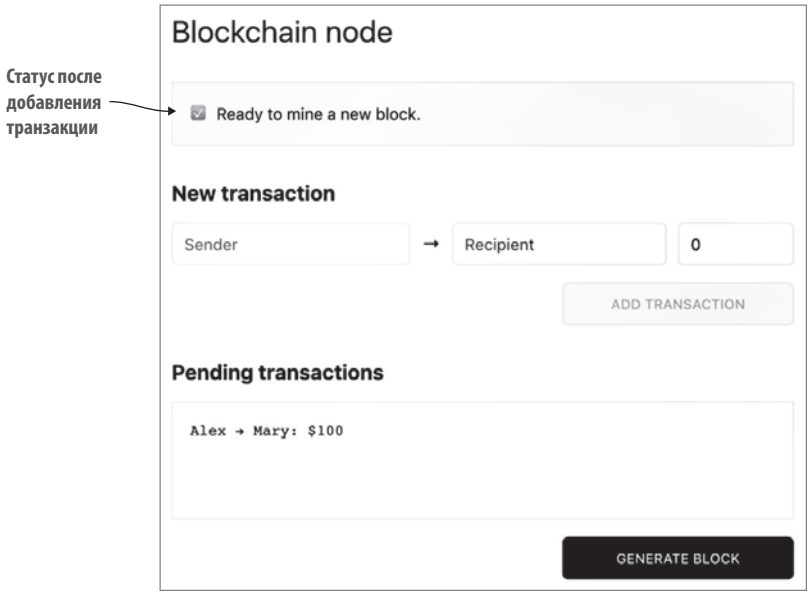


Рис. 14.6. Пользователь готов к добыче нового блока

Как мы в итоге выяснили, проблема была в том, что после добавления первой транзакции состояние компонента `App` содержало «Ready to mine a new block», и это значение не изменялось после добавления других транзакций. Соответственно, строка `setStatus(getStatus(node))`; не изменяла состояние компонента, и React не видел причин для повторного отображения UI.

Этот баг было легко исправить. Мы немного изменили функцию `getStatus()`, добавив в строку статуса счетчик транзакций. Теперь вместо сохранения статуса «Ready to mine a new block» статус включает изменяющуюся часть:

```
`Ready to mine a new block (transactions:
➥ ${node.pendingTransactions.length}).`;
```

Теперь каждый вызов `addTransaction()` будет изменять состояние компонента.

> **ПРИМЕЧАНИЕ** Этот баг было бы лучше исправить, сделав узел неизменяемым. Для неизменяемого объекта при любом изменении состояния создавался бы новый экземпляр, и обход в виде счетчика транзакций оказался бы не нужен.

14.3.2. Генерация нового блока

Давайте еще раз взглянем на листинг 14.6. Компонент App передает обратный вызов onGenerateBlock() в PendingTransactionsPanel, используя props. Когда пользователь щелкает по кнопке GENERATE BLOCK, компонент PendingTransactionsPanel вызывает onGenerateBlock(), который, в свою очередь, вызывает метод generateBlock() в компоненте App. Этот метод показан в следующем листинге.

Листинг 14.8. Метод generateBlock()

```
async function generateBlock() {
    server.requestNewBlock(node.pendingTransactions);
    const miningProcessIsDone = node.mineBlockWith(node. pendingTransactions);

    setStatus(getStatus(node));

    const newBlock = await miningProcessIsDone;
    addBlock(newBlock);
}
```

Приглашает другие узлы начать генерацию нового блока для ожидающих транзакций

Объявляет выражение для добычи блока

Изменяет состояние компонента

Начинает добычу блока и ожидает завершения

Добавляет новый блок в блокчейн

ПРИМЕЧАНИЕ Предоставляя дочерним компонентам ссылки методов родителя, мы контролируем, какие элементы родителя будут доступны потомку. В этом приложении ни TransactionForm, ни PendingTransactionsPanel не имеют доступа к объектам BlockchainNode и WebsocketController. Эти потомки являются строго компонентами представления и могут лишь отображать данные или уведомлять родителя о событиях.

14.3.3. Объяснение хуков useEffect()

В коде компонента App вы можете найти два хука useEffect(). Как вы можете вспомнить, эти хуки могут быть вызваны автоматически при изменении заданной переменной. В следующем листинге показан первый хук useEffect(), который срабатывает только при запуске приложения.

Листинг 14.9. Первый хук useEffect()

```
useEffect(() => {
    setStatus(getStatus(node));
}, []);
```

Цель этого эффекта инициализировать состояние компонента с сообщением «Initializing the blockchain...». Если вы закомментируете этот хук, приложение по-прежнему будет работать, но при его запуске не будет отображаться сообщение статуса.

В листинге 14.10 показан второй хук `useEffect()`, который подключается к WebSocket-серверу, передавая ему обратный вызов `handleServerMessages`, обрабатывающий переданные сервером сообщения.

Листинг 14.10. Хук useEffect(), прикрепленный к handleServerMessages

```
useEffect(() => {
    async function initializeBlockchainNode() {
        await server.connect(handleServerMessages);
        const blocks = await server.requestLongestChain();
        if (blocks.length > 0) {
            node.initializeWith(blocks);
        } else {
          await node.initializeWithGenesisBlock();
        }
        setStatus(getStatus(node));
    }

    initializeBlockchainNode();

    return () => server.disconnect();

}, [handleServerMessages]);
```

Объявляет функцию initializeBlockchainNode()

Подключается к Websocket-серверу, передавая обратный вызов

Запрашивает длиннейшую цепочку

Блокчейн уже содержит блоки

Блоков еще не существует; создает первичный блок

Обновляет состояние App

Вызывает функцию initializeBlockchainNode()

Отключается от WebSocket при уничтожении компонента App

Этот хук прикреплен к handleServerMessages

Если функция используется только внутри эффекта, рекомендуется объявить ее внутри него. Поэтому, так как `initializeBlockchainNode()` используется только предыдущим `useEffect()`, мы объявили ее внутри этого хука.

Он устанавливает изначальное соединение с сервером и инициализирует блокчейн, поэтому нам нужно вызвать его только один раз. Чтобы это обеспечить, мы попытались согласно документации использовать в качестве второго аргумента пустой массив. Пустой массив означает, что этот эффект не использует никаких значений, которые могут участвовать в потоке данных React, следовательно, вызвать его только один раз будет безопасно.

Но React заметил, что этот хук использует работающую в области компонента функцию `handleServerMessages()`, которая, являясь замыканием, может потенциально захватить переменную состояния компонента. Это может утратить актуальность после повторного отображения, но наш эффект сохранит ссылку на `handleServerMessages()`, захватившую старое состояние. Из-за этого React принудил нас заменить пустой массив на `[handleServerMessages]`. Тем не менее, так как мы не будем изменять состояние внутри этого обратного вызова, описываемый `useEffect()` будет вызван только один раз.

Обратите внимание на инструкцию `return` в конце `useEffect()`, показанного в листинге 14.10. Возвращение функции из `useEffect()` необязательно, но если она там присутствует, React гарантирует, что вызовет эту функцию, когда компонент

будет близок к уничтожению. Если бы мы установили Websocket-соединение в других компонентах (не в корневом), то было бы хорошим решением использовать в `useEffect()` инструкцию `return` во избежание утечек памяти.

В процессе обертывания асинхронной функции внутри эффекта в листинге 14.10 изначально мы старались просто добавить ключевое слово `async` следующим образом:

```
useEffect(async () => { await ...})
```

Но так как любая `async`-функция возвращает `Promise`, TS начал жаловаться, что «Type 'Promise<void>' is not assignable to type ' () ? void | undefined'». (Тип 'Promise<void> не может быть присвоен типу ' () ? void | undefined'). Хуку `useEffect()` не понравилась функция, которая возвращала бы `Promise`. Поэтому мы несколько изменили ее сигнатуру:

```
useEffect(() => {
        async function initializeBlockchainNode() {...}
        initializeBlockchainNode();
    }
)
```

Сначала мы объявили асинхронную функцию, а затем ее вызвали. Это удовлетворило TS, и мы смогли повторно использовать тот же код внутри `useEffect()` в главах 10 и 12.

14.3.4. Мемоизация с помощью хука useCallback()

Теперь давайте поговорим о еще одном хуке React — `useCallback()`, который возвращает *мемоизованный* обратный вызов. Мемоизация — это техника оптимизации, которая позволяет хранить результаты вызовов функции и в случае повторения одинакового ввода возвращать кэшированный результат.

Предположим, у вас есть функция `doSomething(a, b)`, которая выполняет длительные вычисления переданных аргументов. К примеру, эти вычисления занимают 30 секунд, и функция, будучи чистой, всегда при одинаковом вводе возвращает одинаковый результат. Это значит, что следующий фрагмент кода должен выполняться 90 секунд, верно?

```
let result: number;
result = doSomething(2, 3); // 30 sec
result = doSomething(10, 15); // 30 sec
result = doSomething(2, 3); // 30 sec
```

Но если бы мы сохранили в таблице результаты для каждой пары аргументов, то нам не понадобилось бы повторно вызывать `doSomethin(2, 3)`, так как у нас

уже был результат для этой пары. Нам бы просто понадобился быстрый поиск по таблице результатов. Это пример, в котором мемоизация способна оптимизировать код, чтобы он выполнялся не 90 секунд, а немногим более шестидесяти.

При этом в компонентах React вам не нужно вручную реализовывать мемоизацию для каждой функции, вместо этого можно использовать хук `useCallback()`. В листинге 14.11 показан этот хук, возвращающий мемоизированную версию функции `doSomething()`.

Листинг 14.11. Обертывание функции в хук useCallback()

```
const memoizedCallback = useCallback(   <-- Хук useCallback()
    () => {
        doSomething(a, b);   <-- Мемоизированная функция doSomething()
    },
    [a, b],   <-- Зависимости doSomething()
);
```

Если функция `doSomething()` является частью компонента React, мемоизация избавит эту функцию от необязательного пересоздания в процессе каждого отображения UI до тех пор, пока не изменятся ее зависимости `a` или `b`.

В компоненте `App` мы обернули все функции, обрабатывающие сообщения от WebSocket-сервера (вроде `handleServerMessages()`) в хук `useCallback()`. На рис. 14.7 показан скриншот кода компонента `App` со свернутыми телами функций, обернутых в `useCallback()`.

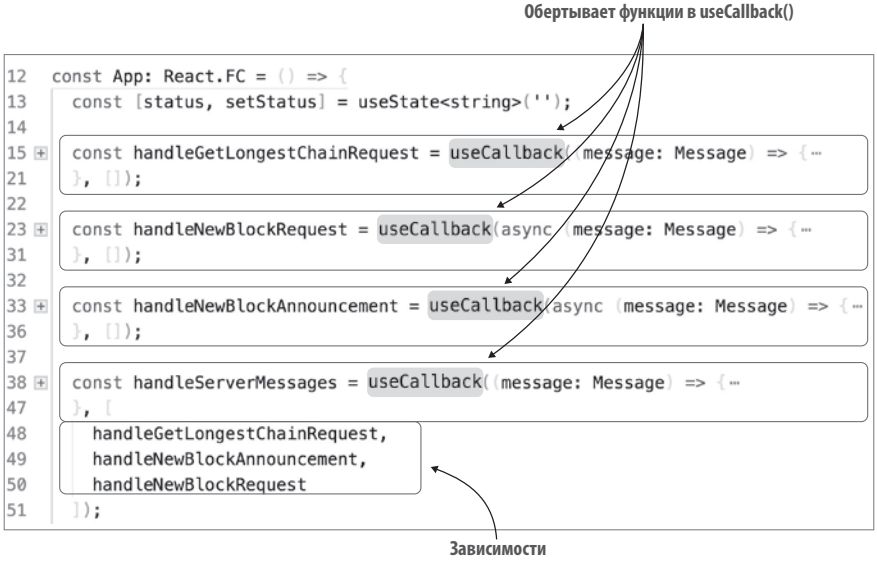


Рис. 14.7. Мемоизированные функции в компоненте App

На рис. 14.7 каждая переменная, объявленная в строках 15, 23, 33 и 38, локальна для компонента App, поэтому React предполагает, что их значения (выражения функций) могут измениться. Обертывая тела этих функций в хуки useCallback(), мы даем React команду повторно использовать один и тот же экземпляр функции при каждом отображении, повышая тем самым его эффективность.

Взгляните на последнюю строку листинга 14.10 — handleServerMessages является зависимостью useEffect(). Технически, если бы вместо выражения функции const handleNewBlockRequest = useCallback() мы использовали function handleNewBlockRequest(), приложение бы по-прежнему работало, но каждая функция пересоздавалась бы при каждом отображении.

В строках 21, 31 и 36 на рис. 14.7 массив зависимостей пуст. Это говорит нам, что указанные обратные вызовы не могут иметь просроченные значения и в зависимостях нет необходимости. В строках 48–49 мы перечислили переменные handleGetLongestChainRequest, handleNewBlockAnnouncement и handleNewBlockRequest как зависимости, используемые внутри обратного вызова handleServerMessages(), как можно видеть в следующем листинге. Мы, конечно, недавно утверждали, что эти обратные вызовы не будут создавать ситуации с просроченным состоянием, но React не может заглянуть внутрь обратных вызовов, чтобы это увидеть.

Листинг 14.12. Обратный вызов handleServerMessages()

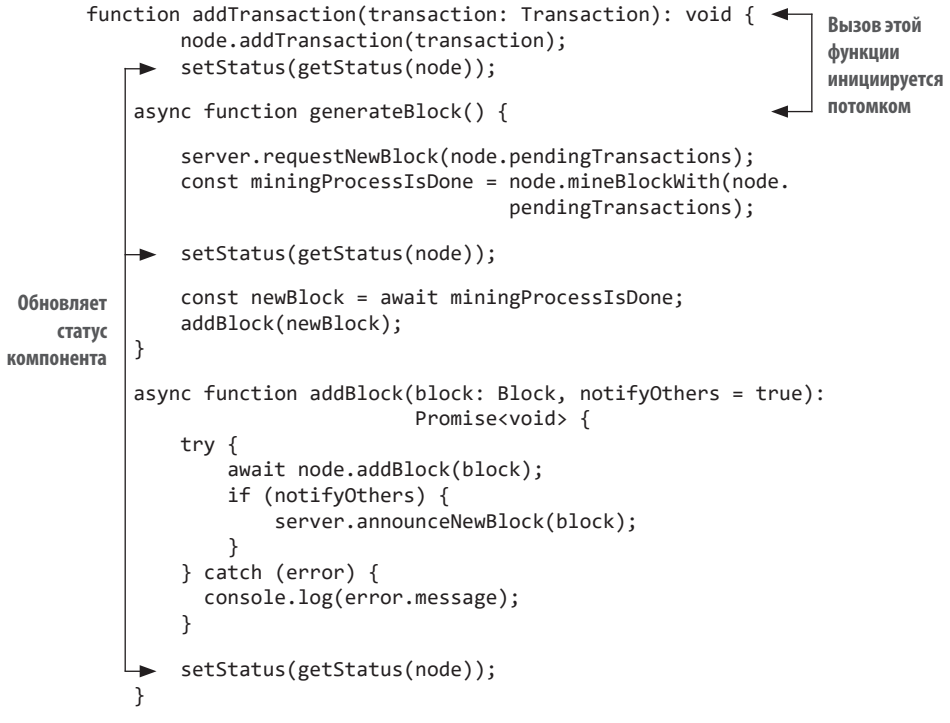
```
const handleServerMessages = useCallback((message: Message) => {
    switch (message.type) {
        case MessageTypes.GetLongestChainRequest:
            return handleGetLongestChainRequest(message);
        case MessageTypes.NewBlockRequest :
            return handleNewBlockRequest(message);
        case MessageTypes.NewBlockAnnouncement :
            return handleNewBlockAnnouncement(message);
        default: {
          console.log(`Received message of unknown type: "${message.type}"`);
        }
    }
}, [
    handleGetLongestChainRequest,
    handleNewBlockAnnouncement,
    handleNewBlockRequest
]);
```

Использует зависимость внутри useCallback()

Объявляет зависимость useCallback()

Помимо функций, сообщающихся с WebSocket-сервером, компонент App имеет еще три функции, которые обмениваются данными с экземпляром BlockChainNode. В следующем листинге показаны функции addTransaction(), generateBlock() и addBlock(). Мы не изменяли логику этих операций, но каждая из этих функций теперь заканчивается вызовом setState() React, который запрашивает повторное отображение.

Листинг 14.13. Три дополнительные функции компонента App




```
function addTransaction(transaction: Transaction): void {
    node.addTransaction(transaction);
    setStatus(getStatus(node));
}

async function generateBlock() {

    server.requestNewBlock(node.pendingTransactions);
    const miningProcessIsDone = node.mineBlockWith(node.
                                pendingTransactions);

    setStatus(getStatus(node));

    const newBlock = await miningProcessIsDone;
    addBlock(newBlock);
}

async function addBlock(block: Block, notifyOthers = true):
                        Promise<void> {
    try {
        await node.addBlock(block);
        if (notifyOthers) {
            server.announceNewBlock(block);
        }
    } catch (error) {
      console.log(error.message);
    }

    setStatus(getStatus(node));
}
```

ПРИМЕЧАНИЕ Вызов функций `addTransaction()` и `generateBlock()` выполняется дочерними компонентами `TransactionForm` и `PendingTransactionsPanel` соответственно. Мы рассмотрим относящийся к этому код в следующем разделе.

Функция `addTransaction()` накапливает ожидающие транзакции, которые обрабатываются функцией `generateBlock()`, и когда один из узлов первым завершает добычу блока, функция `addBlock()` пытается добавить его в блокчейн. Если ваш узел оказался первым в добыче, эта функция добавила бы новый блок и уведомила об этом остальных. В противном случае новый блок поступает с сервера через обратный вызов `handleNewBlockAnnouncement()`.

Функция `getStatus()` расположена в файле App.tsx, но реализована она вне компонента App.

Листинг 14.14. Функция getStatus() из App.tsx

```
function getStatus(node: BlockchainNode): string {
    return node.chainIsEmpty          ? '⏳ Initializing the blockchain...' :
           node.isMining              ? '⏳ Mining a new block...' :
           node.noPendingTransactions ? '✉ Add one or more transactions.' :
                                        '✓ Ready to mine a new block.';
}
```

Когда компонент App вызывает setStatus(getStatus(node)), возможно два исхода: getStatus() вернет либо тот же статус, что и ранее, либо новый. Если статус не изменился, вызов setStaus() не приведет к повторному отображению UI, и наоборот.

На этом мы закончим рассмотрение особенностей React в отношении умного компонента App. Давайте теперь познакомимся с кодом компонентов представления.

14.4. КОМПОНЕНТ ПРЕДСТАВЛЕНИЯ TRANSACTIONFORM

На рис. 14.8 показан UI компонента TransactionForm, который позволяет пользователю наряду с суммой транзакции вводить имена отправителя и получателя. Когда пользователь кликает по кнопке ADD TRANSACTION, эта информация должна отправляться родительскому компоненту App, так как только он знает, как обрабатывать данные.

Рис. 14.8. UI компонента TransactionForm

JSX компонента App, сообщающийся с TransactionForm, показан в следующем листинге:

Листинг 14.15. JSX комопнента App для отображения TransactionForm

```
<TransactionForm
    onAddTransaction={addTransaction}
    disabled={node.isMining || node.chainIsEmpty}
/>
```

Вызов onAddTransaction() потомка приводит к вызову addTransaction() родителя

Условно активирует или отключает потомка

Из этого JSX мы можем догадаться, что когда компонент TransactionForm вызывает свою функцию onAddTransaction(), компонент App вызывает свою addTransaction() (показанную ранее в листинге 14.13). Мы также видим, что дочерний компонент содержит свойство disabled, управляемое статусом переменной node, которая содержит ссылку на экземпляр BlockchainNode.

В следующем листинге показана первая половина кода файла TransactionForm.tsx.

Листинг 14.16. Первая часть TransactionForm.tsx

```
import React, { ChangeEvent, FormEvent, useState } from 'react';
import { Transaction } from '../lib/blockchain-node';

type TransactionFormProps = {
    onAddTransaction: (transaction: Transaction) => void,
    disabled: boolean
};

const defaultFormValue = {recipient: '', sender: '', amount: 0};

const TransactionForm: React.FC<TransactionFormProps> =
➥ ({onAddTransaction, disabled}) => {
    const [formValue, setFormValue] = useState<Transaction>(defaultFormValue);
    const isValid = formValue.sender && formValue.recipient &&
➥ formValue.amount > 0;

    function handleInputChange({ target }: ChangeEvent<HTMLInputElement>) {
        setFormValue({
            ...formValue,
            [target.name]: target.value
        });
    }

    function handleFormSubmit(event: FormEvent<HTMLFormElement>) {
        event.preventDefault();
        onAddTransaction(formValue);
        setFormValue(defaultFormValue);
    }

    return (
        // JSX показан в листинге 14.17.
    );
}
```

Props для отправки данных родителю

Props для получения данных от родителя

Объект со значениями по умолчанию для формы

Этот компонент принимает два объекта props

Состояние компонента

Флаг isValid определяет, когда кнопка может быть активна

Один обработчик событий для всех полей ввода

Передает родителю объект formValue

Сбрасывает форму

Когда пользователь щелкает по кнопке `ADD TRANSACTION`, компонент `TransactionForm` должен вызвать в родителе некую функцию. Поскольку React не хочет, чтобы потомок знал содержимое родителя, этот потомок получает только `props onAddTransaction`, но он должен знать правильную сигнатуру родительской функции, соответствующей `onAddTransaction`. Следующая строка отображает имя `props onAddTransaction` в сигнатуру функции, которая должна быть вызвана в родителе:

```
onAddTransaction: (transaction: Transaction) => void,
```

В листинге 14.13 вы видели, что родительская функция `addTransaction()` имеет сигнатуру `(transaction: Transaction) => void`. В листинге 14.6 вы можете легко найти строку, отображающую родительскую `addTransaction` в `onAddTransaction` потомка.

Компонент `TransactionForm` отрисовывает простую форму и определяет только одну переменную состояния, `formValue`, которая является объектом, содержащим значения текущей формы. Когда пользователь набирает символы в поле ввода, вызывается обработчик событий `handleInputChange()` и сохраняет введенное значение в `formValue`. В листинге 14.17 вы увидите, что этот обработчик событий присвоен каждому полю ввода в форме.

В обработчике `handleInputChange()` мы используем деструктуризацию, чтобы извлечь объект `target`, который указывает на вызвавшее это событие поле ввода. Мы динамически получаем из объекта `target` имя и значение элемента DOM. Свойство `target.name` будет содержать имя поля, а `target.value` — его значение. Чтобы увидеть, как это работает, в инструментах разработчика установите точку останова на методе `handleInputChange()`. Вызывая `setFormValue()`, мы изменяем состояние компонента для отображения текущих значений полей ввода.

> **ПРИМЕЧАНИЕ** В процессе вызова `setState()` мы используем клонирование объекта при помощи оператора распространения. Эта техника описана в разделе A.7 приложения.

Значения по умолчанию для формы транзакции хранятся в переменной `defaultFormValue`. Они используются для первичного отображения формы, а также для сброса формы после нажатия кнопки `ADD TRANSACTION`. Когда пользователь щелкает по этой кнопке, функция `handleFormSubmit()` вызывает `onAddTransaction()`, передавая объект `formValue` родителю (компоненту `App`).

В листинге14/17 показан JSX компонента `TransactionForm`. Это форма с тремя полями ввода и кнопкой `submit`.

React обрабатывает HTML-формы не так, как другие элементы, так как у форм есть внутреннее состояние — объект со значениями всех полей формы. В React вы можете преобразовать стандартное поле формы в *контролируемый компонент* посредством привязки свойства объекта состояния (вроде `formValue.sender`) его атрибуту `value` и добавления обработчика событий `onChange`.

В нашей форме есть три контролируемых компонента (поля ввода), и каждое изменение состояния будет иметь соответствующую функцию обработчика.

В компоненте `transactionForm` такой функцией является `handleInputChange()`. Как вы видите в листинге 14.17, мы просто клонируем объект состояния в `handleInputChange()`, но вы можете разместить в подобном обработчике любую логику приложения.

Листинг 14.17. Вторая часть TransactionForm.tsx

```
return (
    <>
        <h2>New transaction</h2>
        <form className="add-transaction-form" onSubmit={handleFormSubmit}>
            <input
                type="text"
                name="sender"
                placeholder="Sender"
                autoComplete="off"
                disabled={disabled}
                value={formValue.sender}
                onChange={handleInputChange}
            />
            <span className="hidden-xs">•</span>
            <input
                type="text"
                name="recipient"
                placeholder="Recipient"
                autoComplete="off"
                disabled={disabled}
                value={formValue.recipient}
                onChange={handleInputChange}
            />
            <input
                type="number"
                name="amount"
                placeholder="Amount"
                disabled={disabled}
                value={formValue.amount}
                onChange={handleInputChange}
            />
            <button type="submit"
                disabled={!isValid || disabled}
                className="ripple">ADD TRANSACTION</button>
        </form>
    </>
);
```

Активирует кнопку, только когда форма действительна

Привязывает значение из соответствующего свойства состояния

Вызывает handleInputChange при каждом изменении состояния

Привязывает атрибут disable к отключенному props

Привязывает значение из соответствующего свойства состояния

Вызывает handleInputChange при каждом изменении состояния

Привязывает значение из соответствующего свойства состояния

Привязывает атрибут disable к отключенному props

Вызывает handleInputChange при каждом изменении состояния

Активирует кнопку, только когда форма действительна

Мы бы хотели еще раз отметить, что `TransactionForm` — это компонент представления, который умеет только представлять свои значения и определять, какую функцию вызывать при отправке формы. Он ничего не знает о своем родителе и не сообщается ни с какими внешними сервисами, что дает нам полную свободу в его повторном использовании.

14.5. КОМПОНЕНТ ПРЕДСТАВЛЕНИЯ PENDINGTRANSACTIONSPANEL

Каждый раз, когда пользователь кликает по кнопке `ADD TRANSACTION` в компоненте `TransactionForm`, введенная транзакция должна передаваться в `PendingTransactionsPanel`. На рис. 14.9 показан этот компонент, отображенный с двумя ожидающими транзакциями. Эти два компонента не знают друг о друге, следовательно, `App` может выступить в роли посредника при передаче данных между ними.

Рис. 14.9. UI компонента PendingTransactionsPanel

В листинге 14.18 показан фрагмент JSX компонента `App`, который отображает компонент `PendingTransactionsPanel`. `App` сообщается с `PendingTransactionsPanel` практически так же, как и с `TransactionForm`. Этот компонент получает от `App` три `props`.

Листинг 14.18. JSX компонента App для отображения PendingTransactionsPanel

```
<PendingTransactionsPanel
    formattedTransactions={formatTransactions(node.pendingTransactions)}
    onGenerateBlock={generateBlock}
    disabled={node.isMining || node.noPendingTransactions}
/>
```

Форматирует транзакцию и передает ее этому компоненту

Изначально этот потомок должен быть отключен

Вызов onGenerateBlock() потомка приводит к вызову generateBlock() родителя

Первый `props` — это `formattedTransactions`, и компонент `App` передает его в `PendingTransactionsPanel` для отображения. Во время рассмотрения кода файла `App.tsx` мы не затронули его служебную функцию `formatTransactions()`, которая просто создает приятно отформатированное сообщение о транзакции. В следующем листинге показан код не требующей объяснения функции `formatTransactions()`, расположенной в файле `App.tsx` вне компонента `App`. На рис. 14.9 показано, как выглядят транзакции после форматирования.

Листинг 14.19. Функция formatTransactions()

```
function formatTransactions(transactions: Transaction[]): string {
    return transactions.map(t =>`${t.sender} • ${t.recipient}: $${t.amount}`)
    .join('\n');
}
```

Второй объект `props`, `onGenerateBlock` является ссылкой на функцию, которая должна вызываться в родителе компонента `PendingTransactionsPanel`, когда пользователь щелкает по кнопке `GENERATE BLOCK`.

Листинг 14.20. Файл PendingTransactionsPanel.tsx

```
import React from 'react';

type PendingTransactionsPanelProps = {
    formattedTransactions: string;
    onGenerateBlock: () => void;
    disabled: boolean;
}

const PendingTransactionsPanel: React.FC<PendingTransactionsPanelProps> =
            ({formattedTransactions, onGenerateBlock, disabled}) => {
    return (
        <>
            <h2>Pending transactions</h2>
            <pre className="pending-transactions list">
                {formattedTransactions || 'No pending transactions yet.'}
            </pre>
            <div className="pending-transactions form">
                <button disabled={disabled}
                        onClick={() => onGenerateBlock()}
                        className="ripple"
                        type="button">GENERATE BLOCK</button>
            </div>
            <div className="clear"></div>
        </>
    );
}

export default PendingTransactionsPanel;
```

props для отформатированных транзакций

props onGenerateBlock должен использовать эту сигнатуру метода

Выравнивает все (включая последующие родственные элементы) по правой стороне родительского контейнера

Отображает либо предоставленную транзакцию, либо текст по умолчанию

Вызывает props onGenerateBlock()

Очищает правое выравнивание

Когда пользователь кликает по кнопке `GENERATE BLOCK`, мы вызываем `props onGenerateBlock()`, который, в свою очередь, вызывает в компоненте App функцию `generateBlock()`.

В селекторе стилей `.pending-transactions_form` (в index.css) мы используем команду `float:right`, которая выравнивает все, включая последующие элементы потомков, по правой стороне родительского контейнера. Стиль `clear` определен как `clear: both`, что прекращает порядок правого выравнивания, чтобы не нарушить следующий раздел Current Blocks.

14.6. КОМПОНЕНТЫ ПРЕДСТАВЛЕНИЯ BLOCKSPANEL И BLOCKCOMPONENT

Когда пользователь щелкает по кнопке `GENERATE BLOCK` в компоненте `PendingTransactionsPanel`, все активные блоки блокчейна начинают процесс добычи, и после достижения консенсуса новый блок будет добавлен в блокчейн и отображен в компоненте `BlocksPanel`, который может служить родителем одному или более компоненту `BlockComponent`. На рис. 14.10 показано отображение `BlocksPanel` с блокчейном, состоящим из двух блоков.

Рис. 14.10. UI компонента BlocksPanel

В процессе добычи блока и достижения консенсуса привлекаются экземпляры `BlockchainNode` и `WebsocketController`, но так как это компонент представления, `BlocksPanel` не общается напрямую ни с одним из этих объектов. Эта работа делегируется умному компоненту `App`. Компонент `BlocksPanel` не отправляет никаких данных своему родителю. Его цель — отобразить блокчейн, переданный через `props block`:

```
<BlocksPanel blocks={node.chain} />
```

Файл BlocksPanel.tsx содержит код двух компонентов: `BlocksPanel` и `BlockComponent`. В следующем листинге показан код `BlockComponent`, который отвечает за отображение одного блока блокчейна. На рис. 14.10 показаны два экземпляра `BlockComponent`.

Листинг 14.21. BlockComponent

```
const BlockComponent: React.FC<{ index: number, block: Block }> =
➥ ({ index, block }) => {
    const formattedTransactions = formatTransactions(block.transactions);
    const timestamp = new Date(block.timestamp).toLocaleTimeString();

    return (
        <div className="block">
```

Функция formattedTransaction() такая же, как в компоненте App

```
            <div className="block header">
                <span className="block index">#{index}</span>
                <span className="block timestamp">{timestamp}</span>
            </div>
            <div className="block hashes">
                <div className="block hash">
                    <div className="block label">• PREV HASH</div>
                    <div className="block hash-value">{block.previousHash}</div>
                </div>
                <div className="block hash">
                    <div className="block label">THIS HASH</div>
                    <div className="block hash-value">{block.hash}</div>
                </div>
            </div>
            <div>
                <div className="block label">TRANSACTIONS</div>
                <pre className="block transactions">{formattedTransactions
                                    || 'No transactions'}</pre>
            </div>
        </div>
    );
}
```

Номер блока

Хеш предыдущего блока

Хеш блока

Транзакции блока

ПРИМЕЧАНИЕ Согласно рекомендации методологии Блок-Элемент-Модификатор (БЭМ), описанной на сайте Get BEM (http://getbem.com), при именовании некоторых стилей мы используем символы и знаки `--`.

В следующем листинге показан компонент `BlocksPanel`, который служит в качестве контейнера для всех компонентов `BlockComponent`.

Листинг 14.22. Компонент BlocksPanel

```
import React from 'react';
import { Block, Transaction } from '../lib/blockchain-node';

type BlocksPanelProps = {
    blocks: Block[]
};

const BlocksPanel: React.FC<BlocksPanelProps> = ({blocks}) => {
    return (
        <>
            <h2>Current blocks</h2>
            <div className="blocks">
                <div className="blocks ribbon">
                    {blocks.map((b, i) =>
                            <BlockComponent key={b.hash} index={i} block={b}>
➥ </BlockComponent>)}
                </div>
```

Массив экземпляров Block является здесь единственным объектом props

Использует Array.map() для преобразования данных в компоненты

Передает ключ, индекс и props block BlockComponent

```
                <div className="blocks overlay"></div>
            </div>
        </>
    );
}
```

`BlocksPanel` получает массив экземпляров `Block` от компонента `App` и применяет метод `Array.map()` для преобразования каждого объекта `Block` в `BlockComponent`. Метод `map()` передает ключ (код хеша), уникальный индекс блока и объект `Block` в каждый экземпляр `BlockComponent`.

Объекты props для `BlockComponent` — это `index` и `block`. Обратите внимание, что мы присваиваем хеш блока в качестве `props key` каждому экземпляру `BlockComponent` несмотря на то, что `props key` даже не упоминался в листинге 14.21. Именно поэтому, когда у вас есть коллекция отображенных объектов (вроде единиц списка или нескольких экземпляров одного и того же компонента), React нужен способ уникальной идентификации каждого компонента в процессе его сверки с виртуальной DOM, чтобы отслеживать данные, связанные с каждым элементом DOM.

Если вы не используете уникальное значение для `props key` в каждом `BlockComponent`, React выведет в консоли предупреждение «Each child in array or iterator should have a unique key props» (Каждый потомок в массиве или итераторе должен иметь уникальный `props key`). В нашем приложении это не спутает данные, потому что мы добавляем новые блоки только в конец массива, но если бы пользователь мог добавлять или удалять из коллекции компонентов UI произвольные элементы, не используя уникальный `props key`, то могла бы возникнуть ситуация, в которой элемент UI не совпадал бы с лежащими в его основе данными.

На этом завершается наше рассмотрение кода React версии блокчейн-приложения.

ИТОГИ

- В процессе разработки наше веб-приложение React было развернуто под dev-сервером Webpack, но оно также обменивалось данными и с сервером сообщений. Чтобы этого добиться, мы объявили пользовательские переменные среды с URL сервера сообщений. Для WebSocket-сервера этого было достаточно, но если использовать HTTP-серверы, то придется проксировать HTTP-запросы, как описано в документации к Create React App: http://mng.bz/gV9v.

- Как правило, UI приложения React состоит из умных компонентов и компонентов представления. Не размещайте логику приложения в последних, которые предназначены для представления данных, полученных от других компонентов. Компоненты представления также могут реализовывать взаимодействие с пользователем и отправлять вводимые им данные другим компонентам.
- Дочерний компонент никогда не должен вызывать API напрямую из своего родителя. Используя `props`, компонент-родитель должен дать потомку имя, отображенное в функцию, которая будет вызываться в результате его действий. Потомок будет вызывать ссылку предоставленной функции, не зная реального имени родительской функции.
- Для предотвращения необязательного пересоздания выражений функций, расположенных в компонентах React, рассмотрите применение мемоизации с хуком `useCallback()`.

15 Разработка приложений Vue.js с помощью TypeScript

В этой главе:

- ✓ Обзор фреймворка Vue.js.
- ✓ Запуск нового проекта с помощью Vue CLI.
- ✓ Работа с основанными на классах компонентами.
- ✓ Настройка навигации на клиентской стороне при помощи Vue Router.

Angular — это фреймворк, React — библиотека, а Vue.js (она же Vue) можно назвать «библиотекой плюс плюс». Vue (https://vuejs.org) была разработана Эваном Ю (Evan You) в 2014 году в рамках попытки создать облегченную версию Angular. На момент написания книги Vue.js насчитывала 155 000 звезд и 285 участников на GitHub. Эти показатели достаточно высоки, но при этом за Vue не стоит ни одна крупная корпорация в отличие от Angular (Google) или React.js (Facebook).

Vue является прогрессивным, поэтапно внедряемым JS-фреймворком для создания UI в веб, поэтому если у вас уже есть веб-приложение, написанное с помощью или без помощи любой JS-библиотеки, то вы можете ввести Vue сперва в небольшую его часть, а затем постепенно добавлять ее и для других частей по необходимости. Как и в React, вы можете прикреплять экземпляр

Vue к любому HTML-элементу (вроде `<div>`), и в итоге она будет контролировать только его.

Эта библиотека основана на компонентах, которые сосредоточены именно на представлении приложения (V в шаблоне MVC). В своей основе Vue фокусируется на декларативном представлении компонентов UI и, наподобие React, использует виртуальную DOM. Помимо библиотеки кода, она содержит и другие модули, осуществляющие клиентскую маршрутизацию, управление состоянием и др.

> **ПРИМЕЧАНИЕ** Две первые версии Vue были написаны на TypeScript с нуля. Мы писали эту главу в конце 2019-го, и создатели Vue объявили, что готовящаяся к выпуску Vue 3 будет включать ряд существенных изменений: встроенный API реактивности, API-аналог хуков, а также улучшенную интеграцию с TS. Новый функциональный Composition API находится в разработке (https://github.com/vuejs/rfcs/pull/78), так же как и уплощение внутренней структуры узла в виртуальной DOM. Создатели новой версии Vue утверждают, что новый API будет на 100% совместим с текущим синтаксисом и будет работать вместе с другими API, но для внедрения основных изменений они предложат инструмент обновления версий, который должен автоматически перевести вашу базу кода в Vue 3.

Уже имея некоторое представление об Angular и React, вам будет несложно разобраться с принципами работы веб-компонентов, которые могут быть представлены пользовательскими тегами вроде `<transaction-form-component>` или `<BlocksPanel>`. Подобные компоненты могут иметь собственное состояние и получать/отправлять данные, что позволяет им взаимодействовать друг с другом. Компонент может иметь потомков, и в этом отношении Vue работает аналогично Angular или React, хотя и использует для объявления компонентов другой синтаксис.

Как и в случае с Angular/React, вы можете сделать наброски проекта Vue с помощью CLI. Но мы бы хотели, чтобы вы начали с максимально простого способа. В следующем разделе вы увидите, как создавать веб-приложение Hello World без дополнительных инструментов.

15.1. РАЗРАБОТКА ПРОСТЕЙШЕЙ ВЕБ-СТРАНИЦЫ С ПОМОЩЬЮ VUE

В этом разделе мы покажем вам очень простую страницу, написанную с помощью Vue и TS. Эта страница отображает внутри HTML-элемента `<div>` сообщение Hello World. Мы добавим в эту HTML-страницу библиотеку Vue,

используя тег `<script>`, указывающий на URL-адрес сети доставки содержимого (CDN) Vue.

Листинг 15.1. Добавление Vue в index.html

```
<!DOCTYPE html>
    <body>
        <div id="one"></div>
        <div id="two"></div>
                                                    Добавляет Vue из CDN
        <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js></script>
    </body>
</html>
```

Эта веб-страница содержит два пустых тега `<div>` и тег `<script>` для загрузки кода библиотеки Vue. Мы намеренно добавили два тега `<div>`, чтобы продемонстрировать, как вы можете прикрепить экземпляр Vue к одному конкретному HTML-элементу.

Загружая Vue в веб-странице, мы делаем все ее API доступными для сценариев этой страницы. Обратите внимание, что наши элементы `<div>` имеют разные ID, поэтому мы можем дать Vue команду контролировать `<div>`, имеющий ID `one`. Второй же `<div>` может включать содержимое существующего приложения, написанного при помощи другой технологии, следовательно, нам не нужно, чтобы Vue его контролировал.

Конструктор объекта Vue требует аргумент, имеющий тип `Component-Options`. Вы также можете найти имена всех его опциональных свойств в файле определений типов options.d.ts. Мы только укажем свойство `el` (сокращение от element), содержащее ID HTML-элемента, передаваемого под контроль Vue, а также свойство data, которое будет хранить данные для отображения. В следующем листинге показан сценарий, создающий и прикрепляющий экземпляр Vue к первому div, передавая в качестве данных приветствие `"Hello World"`.

Листинг 15.2. Прикрепление экземпляра Vue к первому div

```
<!DOCTYPE html>
    <body>
        <div id="one">          Этот div будет контролироваться Vue
            <h1>{{greeting}}</h1>   Привязка данных будет показывать значение переменной greeting
        </div>
        <div id="two">          Этот div не контролируется Vue
            <h1>{{greeting}}</h1>   Здесь нет привязки данных; браузер будет отображать текст {{greeting}}
        </div>

        <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
```

```
        <script type="text/javascript">
            const myApp = new Vue({   ◄── Создает экземпляр Vue
                el: "#one",   ◄── Прикрепляет экземпляр Vue к элементу с ID «one»
                data: {   ◄── Передает элементу данные
                    greeting: "Hello World"
                }
            })
        </script>
    </body>
</html>
```

Откройте файл index.html в браузере Chrome во вкладке Elements инструментов разработчика, и вы увидите веб-страницу, показанную на рис. 15.1. На этой странице нижний `<div>` является обычным HTML-элементом, и браузер отображает `"{{expression}}"` как текст. В процессе инстанцирования Vue мы передали JS-объект, сообщающий ID верхнего элемента `<div> ("one")`, следовательно, экземпляр Vue запустил его и применил к переменной `greeting` привязку, отобразив ее значение `"Hello World"`.

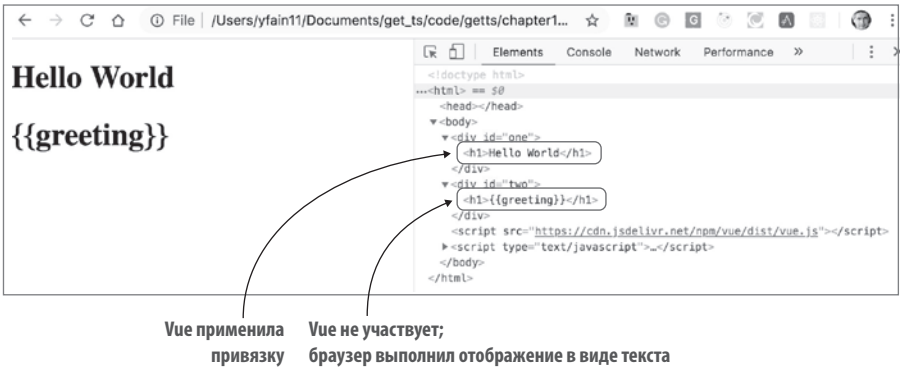


Рис. 15.1. Отображение div с Vue и без

Когда HTML-элемент использует нотацию с двойными фигурными скобками — `{{expression}}`, Vue понимает, что нужно отобразить вычисление этого выражения. UI приложения становится *реактивным*, и как только значение переменной `greeting` изменится, внутри верхнего `<div>` (с ID `"obe"`) будет отображено новое значение. Так как экземпляр Vue ограничен областью конкретного элемента DOM, ничто не мешает вам создать несколько таких экземпляров, привязанных к разным элементам DOM.

Вы можете обратиться ко всем свойствам, определенным в объекте `data`, через ссылочную переменную `myApp`. На рис. 15.2 показан скриншот, сделанный после того, как мы ввели `myApp.greeting = "Hello USA!"`. Новое значение отображено в верхнем `div`.

ПРИМЕЧАНИЕ Введите `URL CDN` в браузере, и вы увидите, какую версию библиотеки Vue она содержит. На момент написания книги была версия 2.6.10.

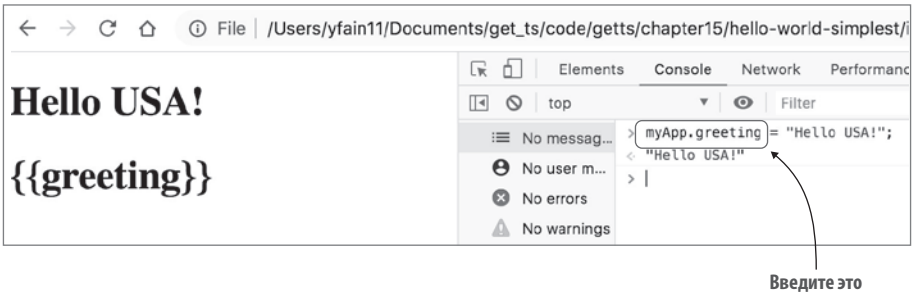


Рис. 15.2. Изменение значения приветствия в консоли браузера

Экземпляр Vue содержит один или более компонентов UI. В листинге 15.2 мы передали в этот экземпляр объектный литерал с двумя свойствами, но могли бы передать объект с функцией `render()` для отображения компонента верхнего уровня:

```
new Vue({
    render: h => h(App) // App является компонентом верхнего уровня.
})
```

Начиная со следующего раздела, вы будете встречать этот синтаксис в приложениях, сгенерированных при помощи Vue CLI. Здесь буква `h` означает сценарий, генерирующий HTML-структуры, подобные функции `createElement()`. Это способ сокращенного написания следующего кода:

```
render: function (createElement) {
    return createElement(App);
}
```

Итак, `h` — это функция `createElement()`, описанная в документации Vue по ссылке http:// mng.bz/5AJO.

ПРИМЕЧАНИЕ `h` означает `hyperscript` — сценарий, генерирующий HTML-структуры.

В нашем простом приложении, показанном в листинге 15.2, в роли компонента UI мы просто использовали элемент DOM `<div>`, но в следующем разделе вы увидите, как объявлять компонент Vue, имеющий три раздела:

- декларативный шаблон;
- сценарий;
- стили.

Свойство `data` в листинге 15.2 играло роль состояния компонента. Если бы мы добавили элемент ввода, чтобы пользователь мог вводить данные (например, имя), экземпляр Vue обновил бы состояние компонента, и функция `render()` отобразила бы компонент повторно с новым состоянием.

Давайте переключимся на настройку Node-проекта, чтобы посмотреть, как приложение Vue может быть разделено на компоненты UI в реальном сценарии.

15.2. ГЕНЕРАЦИЯ И ЗАПУСК ПРИЛОЖЕНИЯ С ПОМОЩЬЮ VUE CLI

Интерфейс командной строки под названием Vue CLI (https://cli.vuejs.org/) автоматизирует процесс создания проекта Vue, имеющего компилятор, бандлер, сценарии для воспроизводимых сборок и файлы конфигурации. Этот инструмент генерирует все необходимые для Webpack файлы конфигурации, что дает возможность сконцентрироваться на написании приложения, а не тратить время на настройку инструментов.

Чтобы установить пакет Vue CLI глобально на компьютер, выполните в терминале следующую команду:

```
npm install @vue/cli -g
```

> **ПРИМЕЧАНИЕ** Чтобы увидеть, какая версия Vue CLI была установлена, введите команду `vue --version`. При написании этой книги мы использовали версию 3.9.2.

Теперь, чтобы сгенерировать новый проект, вы можете также в терминале выполнить команду `vue`. Для создания приложения TS выполните команду `vue create`, сопроводив ее именем приложения:

```
vue create hello-world
```

Эта команда откроет диалоговое окно, предлагающее выбрать опции для создаваемого проекта. Конфигурация по умолчанию предполагает Babel и ESLint, но для работы с компилятором TS нужно выбрать опцию Manually Select Features (Самостоятельный выбор инструментов). После этого вы увидите список опций, показанный на рис. 15.3, и сможете выбрать или, наоборот, отменить выбранные опции, используя стрелки вверх/вниз и клавишу пробела.

Для нашего проекта hello-world мы выбрали только TypeScript и нажали Ввод. Следующим вопросом будет «Use class-style component syntax?» (Использовать синтаксис компонентов в стиле класса?). С этим соглашайтесь. Далее последует

вопрос о том, хотите ли вы параллельно с TypeScript использовать Babel (мы от этой опции отказались). Среди остальных вопросов мы выбрали оставить раздельными файлы конфигурации для Babel и других инструментов, не стали сохранять ответы для дальнейших проектов и в качестве пакетного менеджера по умолчанию выбрали npm.

Рис. 15.3. Ручной выбор инструментов проекта

Vue CLI сгенерировал новый проект Node в директории hello-world и установил все необходимые зависимости. Если вы готовы запустить это приложение, просто введите в окне терминала следующие команды:

```
cd hello-world
npm run serve
```

Код сгенерированного проекта будет скомпилирован, и Webpack DevServer передаст приложение на localhost:8080, как показано на рис. 15.4.

Процедура генерации и запуска проекта Vue аналогична использованию CLI для генерации проектов Angular и React. Изнутри Vue CLI также использует Webpack для связывания и его webpack-dev-server для обслуживания приложения в dev-режиме. Когда Webpack собирает связки для развертывания, он использует специальный плагин Vue для преобразования кода каждого компонента в JS, чтобы браузеры могли его считать и отобразить.

В следующем листинге показан файл package.json, включающий команды npm сценария `serve` и `build` для запуска dev-сервера и сборки связок с помощью Webpack.

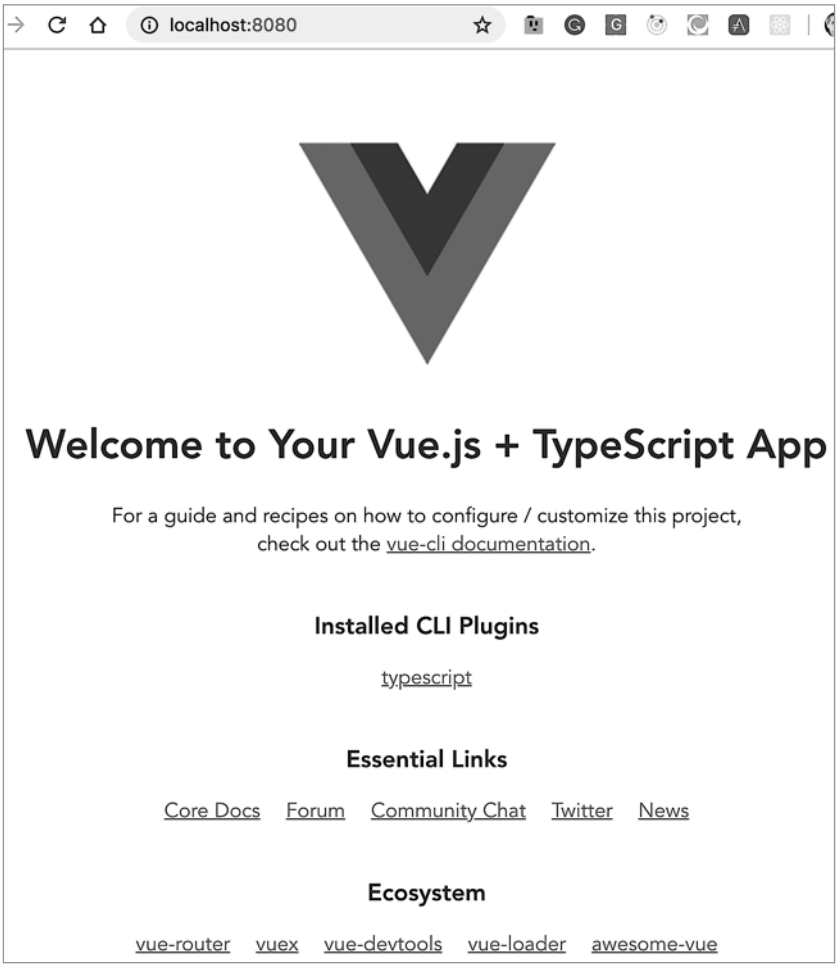


Рис. 15.4. Запуск первично сгенерированного проекта

ПРИМЕЧАНИЕ Установка Vue с использованием npm дает вам файлы объявлений типов TS, благодаря которым IDE предложит автоподстановку и помощь со статическими типами без необходимости использовать дополнительные инструменты.

Давайте откроем сгенерированный проект в VS Code и познакомимся с его структурой, показанной на рис. 15.5. Он структурирован как типичный проект Node.js, где все зависимости перечислены в `package.json` и установлены в директории `node_modules`. Поскольку мы пишем на TypeScript, опции компилятора содержатся в `tsconfig.json`. Файл `main.ts` загружает компонент верхнего уровня из файла `App.vue`. Все компоненты UI расположены в директории `components`.

Листинг 15.3. Сгенерированный файл package.json

```
{
    "name": "hello-world",
    "version": "0.1.0",

    "private": true,
    "scripts": {
        "serve": "vue-cli-service serve",
        "build": "vue-cli-service build"
    },
    "dependencies": {
        "vue": "^2.6.10",
        "vue-class-component": "^7.0.2",
        "vue-property-decorator": "^8.1.0"
    },
    "devDependencies": {
        "@vue/cli-plugin-typescript": "^3.9.0",
        "@vue/cli-service": "^3.9.0",
        "typescript": "^3.4.3",
        "vue-template-compiler": "^2.6.10"
    }
}
```

Запускает приложение, используя dev-сервер Webpack

Собирает связки приложения с помощью Webpack

TypeScript плагин CLI

Компилятор TypeScript

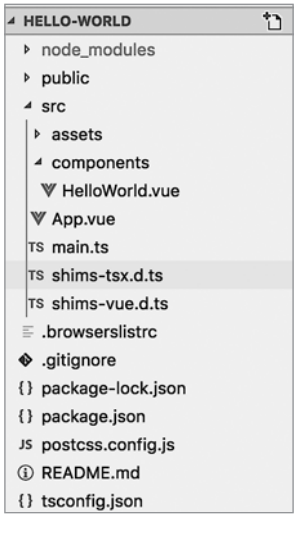


Рис. 15.5. Структура проекта Hello World, сгенерированного CLI

Каталог public, в свою очередь, содержит файл index.html, хранящий разметку, включая HTML-элемент, который будет контролироваться Vue. Процесс сборки изменит этот файл включением сценариев со связками приложения. Вы вольны добавлять директории и файлы, содержащие логику приложения, и мы сделаем это в главе 16 при работе с Vue-версией блокчейн-клиента.

В листинге 15.4 показано содержимое файла main.ts, который создает экземпляр Vue и первично запускает приложение. На этот раз сценарий использует объект `options` со свойством `render`, которое хранит функцию (`h => h(App)`), создающую экземпляр компонента `App` и отображающую его в элементе DOM с ID app.

Листинг 15.4. main.ts

```
import Vue from 'vue'
import App from './App.vue'

Vue.config.productionTip = false

new Vue({  ◄—— Инстанцирует Vue и передает объект options
    render: h => h(App),  ◄—— Начинает отображение дерева компонентов
}).$mount('#app')  ◄—— Прикрепляет экземпляр Vue к элементу DOM с ID «app»
```

> **ПРИМЕЧАНИЕ** Если вы разрабатываете в VS Code, установите расширение под названием Vetur (https://vuejs.github.io/vetur). Оно предлагает выделение синтаксиса Vue, линтинг, автоподстановку, форматирование и не только.

В листинге 15.2 мы *встроили* Vue в конкретный HTML-элемент посредством объекта конфигурации со свойством `el`:

```
const myApp = new Vue({
            el: "#one"
            ...
            }
        })
```

В листинге 15.4 экземпляр Vue не получил свойство `el`. Вместо этого вызов метода `$mount('#app')` начинает процесс встраивания и прикрепляет экземпляр Vue к элементу DOM с ID app. Если вы откроете сгенерированный файл public/index.html, то найдете в нем элемент `<div id="app"><?div>`.

Теперь давайте рассмотрим код файла App.vue. Он состоит из трех разделов: `<template>`, `<script>` и `<style>`. На рис. 15.6 показаны три этих раздела после того, как мы свернули их содержимое в VS Code. Обратите внимание, что раздел `<script>` имеет атрибут `lang = "ts"`, который означает TypeScript.

В листинге 15.5 показано содержимое сгенерированного CLI раздела `<template>`. Поскольку ранее вы видели, как Angular и React представляют пользовательские веб-компоненты, то должны с легкостью определить здесь тег дочернего компонента `<HelloWorld>`.

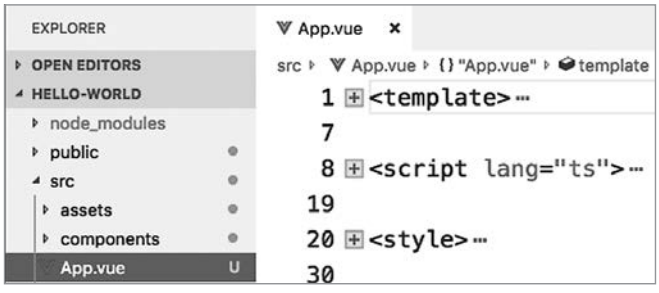


Рис. 15.6. Три раздела файла App.vue

Листинг 15.5. Раздел <template> компонента App

```
<template>
    <div id="app">
        <img alt="Vue logo" src="./assets/logo.png">
        <HelloWorld msg="Welcome to Your Vue.js + TypeScript App"/>
    </div>
</template>
```

Дочерний компонент HelloWorld

Из этого шаблона видно, что компонент `HelloWorld` получает свойство `msg`, а компонент `App` передает в него приветственное сообщение. Большая часть содержимого сгенерированного приложения отображается компонентом `HelloWorld`.

В листинге 15.6 показан раздел `<script>` компонента `App`. Как и в случае с Angular, при написании приложений Vue в TypeScript вы можете использовать декораторы. В данном случае пример — это `@Component()`, получающий опциональный аргумент типа `ComponentOprions`, который имеет такие свойства, как `el`, `data`, `template`, `props`, `components` и др.

Листинг 15.6. Раздел <script> компонента App

```
<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';
import HelloWorld from './components/HelloWorld.vue';

@Component({
    components: {
        HelloWorld,
    },
})
export default class App extends Vue {}
</script>
```

Применяет декоратор Component к классу App

Передает аргумент ComponentOptions со свойством components

Компонент — это класс, расширяющий Vue

> **ПРИМЕЧАНИЕ** Для поддержки декораторов опция компилятора `experimentalDecorators` в tsconfig.json должна быть определена как `true`.

В процессе генерации кода CLI добавил в package.json две зависимости: vue-class-component и vue-property-decorator. Пакет vue-class-component позволяет нам написать компонент Vue как класс, расширяющий Vue, но начиная с Vue 3.0, классовые компоненты будут поддерживаться нативно. Пакет vue-property-decorator позволяет нам использовать разнообразные декораторы вроде `@Component()`, `@props()` и др.

В отсутствие этих пакетов мы могли бы использовать нотацию объектного литерала и экспортировать вместо класса объект в раздел `<script>` файла App.vue:

```
import HelloWorld from './components/HelloWorld.vue';

export default {
    name: 'app',
    components: {
        HelloWorld
    }
}
```

Дочерний компонент `HelloWorld` имеет большой раздел `<template>` со множеством тегов `<a>`, но в самом верху вы увидите привязанное значение `{{msg}}`, как показано в листинге 15.7.

Листинг 15.7. Фрагмент шаблона компонента HelloWorld

```
<template>
    <div class="hello">
        <h1>{{ msg }}</h1>  ◄—— Привязывает значение свойства msg к представлению
        <p>
        <!—Оставшееся содержимое опущено в целях сокращения-->
</template>
```

Раздел `<script>` компонента `HelloWorld` показан в следующем листинге. В нем вы можете видеть два TS-декоратора: `Component()` и `Prop()`. В главе 13 мы представили объект `props` в React.js. В Vue эти объекты играют ту же роль — передают данные от родителя потомку.

Листинг 15.8. Раздел <script> в HelloWorld.vue

```
<script lang="ts">
import { Component, Prop, Vue } from 'vue-property-decorator';

@Component  ◄—— Использует декоратор класса @Component() без аргументов
export default class HelloWorld extends Vue {
    @Prop() private msg!: string;  ◄—— Использует декоратор свойства @Prop()
}
</script>
```

Вы заметили восклицательный знак после `msg`? Это оператор ненулевого утверждения. Добавляя точку восклицания к имени свойства, вы говорите модулю проверки типов TS: «Не ругайся на возможность msg быть null или undefined, такого не будет, я тебе обещаю!»

Вы также можете предоставить для `msg` значение по умолчанию следующим образом:

```
@Prop({default: "The message will go here"}) private msg: string;
```

Vue CLI сгенерировал код с декоратором уровня свойств `@Prop()`, чтобы объявить, что компонент `HelloWorld` получает одно свойство — `msg`. Иначе это можно сделать, использовав свойство `props` декоратора `@Component()`. В следующем фрагменте кода показан альтернативный способ передачи `props msg` через свойство `@Component()`:

```
@Component({
    props: {
        msg: {
            default: "The message will go here"
        }
    }
}
export default class HelloWorld extends Vue { }
```

Если родительский компонент не присвоит атрибуту `msg` значение, например `<HelloWorld />`, то будет отображено значение свойства по умолчанию.

Используя props, вы можете отправлять данные от родителя потомку, но для обратной передачи данных от потомка родителю нужно использовать метод `$emit()`. Например, дочерний компонент `<order-component>` может отправить родителю событие `place-order` с `orderData` в качестве полезной нагрузки следующим образом:

```
this.$emit("place-order", orderData);
```

Родитель же может получить это событие так:

```
<order-component @place-order = "processOrder">

...

processOrder(payload) {

// обработка полезной нагрузки, то есть orderData, полученной из order-component.

}
```

> **ПРИМЕЧАНИЕ** Вы увидите пример использования `$emit()` в листинге 16.6. В нем компонент `PendingTransactionsPanel` отправляет своему родителю событие `generate-block`.

Теперь, когда вы понимаете, как работает простое приложение Vue, мы покажем вам, как организуется клиентская навигация при помощи маршрутизатора, предлагаемого Vue.

15.3. РАЗРАБОТКА ОДНОСТРАНИЧНЫХ ПРИЛОЖЕНИЙ С МАРШРУТИЗАЦИЕЙ

В главе 11 мы разработали простое приложение Angular, которое считывало и отображало данные из файла products.json. В текущем разделе мы создадим одностраничное приложение Vue, которое также будет считывать файл и показывать список товаров. Через это приложение мы представим вам Vue Router и покажем, как использовать некоторые из директив этой библиотеки для отображения списка товаров. Затем мы рассмотрим еще одно приложение, в котором покажем, как вы можете передавать параметры при переходе по маршруту, ведущему к представлению описания товара.

Первое приложение расположено в директории router-product-list, а второе в router-product-details.

> **ПРИМЕЧАНИЕ** В главе 11 мы представили маршрутизатор Angular. В свою очередь, пакет vue-router реализует навигацию на клиентской стороне, используя аналогичные принципы.

Посадочная страница одностраничного приложения router-product-list будет показывать список товаров в компоненте `Home`, как изображено на рис. 15.7. Пользователь сможет щелкнуть по выбранному товару, чтобы приложение при необходимости его обрабатывало. Ссылка `About` будет вести к представлению `About`, не отправляя запросов к серверу.

Смысл использования Vue Router — в поддержании навигации пользователя на клиентской стороне и сохранении состояния в адресной строке. Он создает место для закладки, которую можно открыть напрямую или предоставить для общего использования без необходимости прохождения нескольких шагов для просмотра нужного представления. Кроме того, маршрутизатор позволяет вам избегать загрузки отдельных веб-страниц с сервера — страница остается той же самой, но пользователь может перемещаться по клиенту от одного представления к другому, не запрашивая у сервера загрузку разных страниц. Это возможно благодаря тому, что код всех компонентов UI уже загружен браузером.

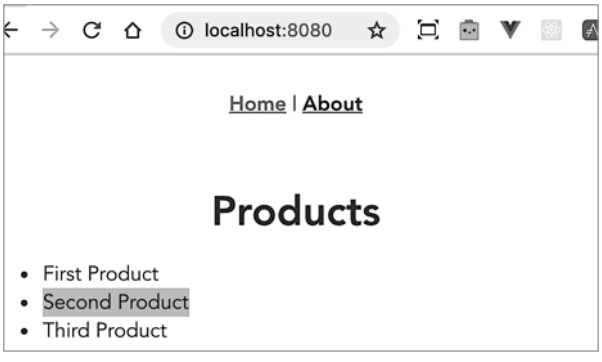


Рис. 15.7. В списке выделен второй товар

В одностраничных приложениях мы не используем для ссылок оригинальные теги `<a href="...">`, поскольку они бы вызывали запросы к серверу и перезагрузку страницы. Фреймворк, поддерживающий маршрутизацию на стороне клиента, производит теги привязки, которые включают обработчики кликов для вызова функций на клиенте и обновления адресной строки.

В Vue маршрутизатор предлагает тег `<router-link>`, который не отправляет запрос серверу. Для маршрута `about` Vue Router сформирует URL localhost:8080/about, а затем считает отображение для сегмента /about и отрисует компонент `About` в области `<router-view>`. Если пользователь кликает по ссылке `About` в первый раз, Vue лениво загружает компонент `About`, прежде чем его отрисовать. При всех последующих кликах по этой ссылке будет просто отображаться компонент `About`.

Vue Router реализован в пакете vue-router, и вы найдете его в списке зависимостей файла package.json.

15.3.1. Генерация нового приложения с Vue Router

И вновь мы прибегли к использованию CLI, чтобы сгенерировать проект router-product-list. Но на этот раз мы дополнительно выбрали Router в списке опций CLI. CLI также спросил, должно ли это приложение использовать для маршрутизатора режим истории, на что мы ответили согласием.

API History реализуется браузерами, поддерживающими HTML5 API, следовательно, если ваше приложение должно поддерживать очень старые браузеры, то режим истории выбирать не стоит. Без этого режима все URL для разделения серверной и клиентской частей приложения будут включать символ решетки.

Например, URL клиентского ресурса может выглядеть как http://localhost:8080/#about, где сегмент слева от решетки обрабатывается сервером, а сегмент справа от нее уже обрабатывается клиентским приложением. Если же вы выбираете режим истории, то URL того же ресурса будет выглядеть как http://localhost:8080/about. Подробнее о режиме истории в HTML5 вы можете прочитать в документации к Mozilla по ссылке http://mng.bz/6w5e.

Перейдите в директорию router-product-list и выполните команду `npm run serve`, после чего вы увидите посадочную страницу сгенерированного приложения. Она будет похожа на изображенную на рис. 15.4, но с небольшим дополнением: в верхней части окна будут отображены две ссылки — `Home` и `About`, как на рис. 15.8.

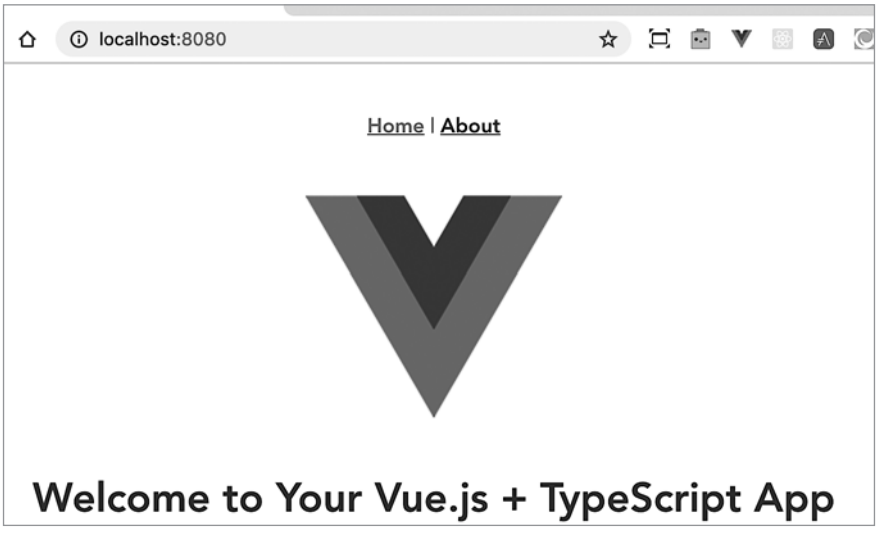


Рис. 15.8. Сгенерированный компонент App с двумя ссылками сверху

CLI сгенерировал директорию src/views с двумя файлами: Home.vue и About.vue. Эти компоненты контролируются Vue Router. Обратите внимание на URL на рис. 15.8 — это просто протокол (не показан), имя домена и порт. В этом URL отсутствует клиентский сегмент. Мы можем предположить, что маршрутизатор был настроен на отображение по умолчанию компонента `Home` в случае, если URL не содержит клиентский сегмент. Если пользователь кликает по ссылке `About`, браузер отображает компонент `About`, как показано на рис. 15.9.

На этот раз URL содержит клиентский сегмент /about, и снова мы можем предположить, что маршрутизатор был настроен на отображение для этого сегмента компонента `About`. Вскоре вы увидите, что эти наши догадки были верны.

Рис. 15.9. Сгенерированный компонент About

Сгенерированный файл main.ts импортирует объект `Router` и добавляет его в экземпляр Vue.

Листинг 15.9. Файл main.ts импортирует конфигурацию маршрутизатора

```
import Vue from 'vue';
import App from './App.vue';
import router from './router';  ◄—— Импортирует конфигурацию маршрутов из router.ts

Vue.config.productionTip = false;

new Vue({
    router,  ◄—— Добавляет в экземпляр Vue объект Router с настроенными маршрутами
    render: (h) => h(App),
}).$mount('#app');
```

> **ПРИМЕЧАНИЕ** В листинге 15.9 мы передаем в экземпляр Vue объектный литерал, и ES6 позволяет нам использовать сокращенный синтаксис: можно указать только имя свойства, если оно совпадает с именем переменной, содержащей значение. Именно поэтому вместо написания `router: router` мы написали `router`.

Экземпляр `Vue` включает ссылку на объект `Router`, содержащий настроенные маршруты, поэтому он знает, какой компонент отображать, когда пользователь кликает по `Home` или `About`. Изначальная конфигурация маршрутов была сгенерирована в файле router.ts, а его содержимое показано в следующем листинге. В целом, когда вы проектируете одностраничное приложение, то должны подумать о навигации пользователя по клиенту и создать массив, который отображает сегменты в компоненты UI. В следующем листинге такой массив назван `routes`.

Листинг 15.10. Файл router.ts, сгенерированный CLI

```
import Vue from 'vue';
import Router from 'vue-router';
import Home from './views/Home.vue';

Vue.use(Router);  ◄—— Активирует использование пакета Router
```

```
export default new Router({  ◄── Создает объект Router

  mode: 'history',  ◄── Поддерживает HTML5 History API (# отсутствует в URL)
  base: process.env.BASE_URL,  ◄── Использует в качестве base URL сервера
  routes: [  ◄── Конфигурирует массив маршрутов
    {
        path: '/',
        name: 'home',
        component: Home,  ◄── Отображает компонент Home для пути / по умолчанию
    },
    {
        path: '/about',
        name: 'about',
        component: () => import(/* webpackChunkName: "about" */
➥ './views/About.vue'),  ◄── Отображает компонент About для пути /about
    },
  ],
});
```

Когда вы создаете экземпляр объекта `Router`, то передаете в его конструктор объект типа `RouterOptions`. Мы не использовали здесь свойство `linkActiveClass`, но если вам не нравится зеленый цвет для активной ссылки, можете изменить его при помощи этого свойства.

Активируя пакет `Router`, мы получаем доступ к особой переменной `$route` и будем использовать ее в следующем разделе для получения параметров, передаваемых в процессе навигации. Обратите внимание на свойство `mode`: его значение `history`, потому что мы выбрали режим истории в процессе генерации приложения. Путь '/ отображается в компонент Home, и CLI не забыл импортировать этот компонент из файла Home.vue. Но вот путь /about, вместо того чтобы отображаться в компонент `About`, отображается в следующее стрелочное выражение:

```
() => import(/* webpackChunkName: "about" */ './views/About.vue')
```

Эта строка говорит, что маршрутизатор должен лениво загружать компонент `About` при посещении его маршрута. Чтобы это сработало, код дает команду Webpack, чтобы он при генерации связок продакшена разделил код и сгенерировал для этого маршрута отдельную часть (about.[хеш].js). Импорт выполняется динамически только после того, как пользователь решает перейти к представлению About.

> **ПРИМЕЧАНИЕ** Чтобы увидеть, что Webpack действительно разделяет код, запустите продакшен-сборку с командой `npm run build` и проверьте содержимое директории dist. В ней вы найдете отдельный файл с именем, аналогичным about.8027d92e.js. В продакшене этот файл не будет загружаться до тех пор, пока маршрутизатор не перейдет к представлению `About`.

Наш компонент верхнего уровня App.vue имеет ссылки, которые будут вести к отображению одного или другого представления, и листинг 15.11 показывает раздел template этого компонента, который содержит теги `<router-link>` и `<router-view>`. Тег `<router-view>` определяет область для отображения изменяющегося содержимого (компонента `Home` или `About`).

Каждый `<router-link>` имеет атрибут `to`, сообщающий Vue, какой компонент отображать, основываясь на настроенных маршрутах. В атрибуте `to` тега `<router-link>` вы указываете, куда следовать, и маршрутизатор использует значение этого атрибута, чтобы решить, какой компонент отображать. Например, `to "/"` указывает, куда проследовать, если URL не содержит сегмент клиентской стороны.

Листинг 15.11. Раздел <template> файла App.vue

```
<template>
    <div id="app">
        <div id="nav">
            <router-link to="/">Home</router-link>
            <router-link to="/about">About</router-link>
        </div>
        <router-view/>
    </div>
</template>
```

Отображает компонент по умолчанию

Отображает компонент, настроенный для URL-сегмента /about

Здесь маршрутизатор должен отобразить Home или About

ПРИМЕЧАНИЕ В листинге 15.11 оба тега `<router-link>` имеют в атрибутах `to` статические значения. Это не обязательно должно быть именно так, и вы можете привязать к атрибуту `:to` переменную (обратите внимание на двоеточие перед `to`).

В следующем разделе мы заменим код, сгенерированный в файле Home.vue, чтобы он считывал и отображал список товаров. Мы также заменим код в About.view, чтобы он показывал описание выбранного товара. Параллельно с этим мы продемонстрируем, как вы можете передавать данные в процессе перемещения к представлению описания товара.

15.3.2. Отображение списка товаров в представлении Home

В этом разделе мы рассмотрим файл products.json из директории public, содержащий следующее:

```
[
    { "id":0, "title": "First Product", "price": 24.99 },
    { "id":1, "title": "Second Product", "price": 64.99 },
    { "id":2, "title": "Third Product", "price": 74.99}
]
```

В этом файле расположены данные о товаре в формате JSON, имеющие достаточно простую структуру, что позволяет с легкостью написать TS-интерфейс для представления этого товара. Но вы также можете сгенерировать соответствующий интерфейс TS, используя сторонний инструмент вроде MakeTypes (https://jvilk.com/MakeTypes). На рис. 15.10 показан скриншот сайта MakeTypes. Вы просто вставляете JSON-данные в текстовое поле слева, и он генерирует соответствующий интерфейс TS в поле справа. Мы использовали MakeTypes для генерации интерфейса `Product`, расположенного в файле product.ts.

Наше приложение должно считывать этот файл, как только пользователь переходит по маршруту `Home`, который является маршрутом по умолчанию. Но где именно в компоненте `Home` нам следует поместить код для получения данных? Знаем ли мы точно, когда завершается создание компонента `Home`? Да, знаем. Vue предлагает ряд обратных вызовов, которые активируются на разных стадиях жизненного цикла компонента. В файле объявлений типов TS, options.d.ts, расположенном в директории node_modules/vue/types, вы найдете объявление интерфейса `ComponentOptions`. В нем содержатся объявления всех хуков жизненного цикла.

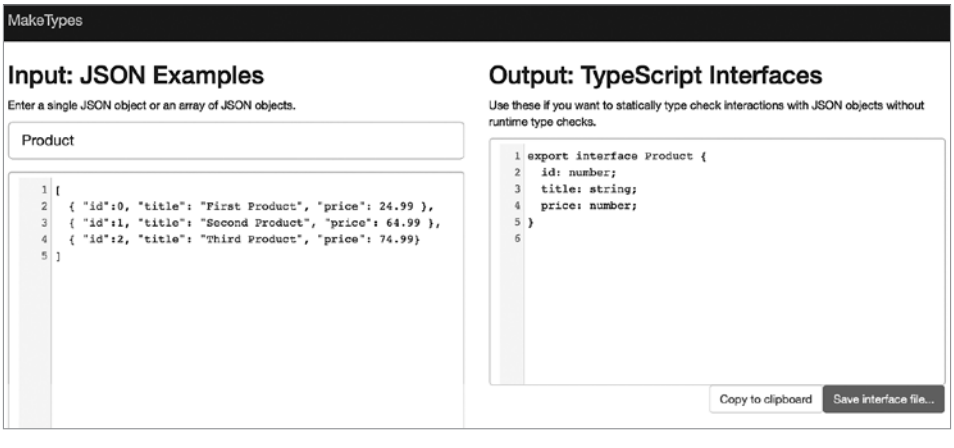


Рис. 15.10. Генерация интерфейса TypeScript из JSON при помощи MakeTypes

Листинг 15.12. Хуки жизненного цикла компонента

```
beforeCreate?(this: V): void;
created?(): void;
beforeDestroy?(): void;
destroyed?(): void;
beforeMount?(): void;
mounted?(): void;
beforeUpdate?(): void;
updated?(): void;
```

```
activated?(): void;
deactivated?(): void;
errorCaptured?(err: Error, vm: Vue, info: string): boolean | void;
serverPrefetch?(this: V): Promise<void>;
```

Вы можете найти описание каждого из этих методов документации к Vue.js (http://mng.bz/omBZ), мы же просто скажем, что для наших нужд подходит метод `created()`. Он вызывается, когда компонент инициализирован и готов к получению данных и обработке событий.

Хуки жизненного цикла вызывает Vue, следовательно, нам нужно лишь поместить наш получающий данные код внутрь метода `created()`. В следующем листинге показана первая измененная версия компонента `Home`.

Листинг 15.13. Добавление хука жизненного цикла компонента created()

```
<template>
    <div class="home">
        <h1>I'm the Home component</h1>
    </div>
</template>

<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';

@Component
export default class Home extends Vue {

    created() {  ◄—— Этот хук жизненного цикла вызывается Vue
        console.log("Home created!");
    }
}

</script>
```

ПРИМЕЧАНИЕ Vue Router имеет собственные хуки жизненного цикла и защиты, которые позволяют вам перехватывать важные события в процессе перемещения по маршруту. Все они описаны в документации по ссылке https://router.vuejs.org/guide/.

Запустите приложение, и вы увидите в консоли браузера сообщение «Home created!». Теперь, когда мы уверены, что хук `created()` вызывается, мы поручим ему получение товаров.

Листинг 15.14. Получение товаров

```
<template>
    <div class="home">
        <h1>I'm the Home component</h1>
    </div>
```

```
</template>

<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';
import { Product } from '@/product';  <-- Импортирует интерфейс Product

@Component
export default class Home extends Vue {

    products: Product[]=[];

    created() {
        fetch("/products.json")  <-- Инициирует получение данных, используя Promise
        .then(response => response.json())  <-- Преобразует ответ в формат JSON
        .then(json => {
          this.products=json;  <-- Заполняет массив товаров данными

          console.log(this.products);  <-- Выводит полученные данные в консоль
        },
        error => {
            console.log('Error loading products.json:', error);
        });
    }
}
</script>
```

В этой версии компонента Home мы использовали API Fetch браузера, чтобы считать файл products.json и просто вывести полученные данные в консоль. Здесь мы применили синтаксис промисов, а позднее в приложении router-product-detail мы используем ключевые слова `async` и `await`, чтобы вы могли сравнить эти подходы.

В листинге 15.14 есть важная инструкция `import`, которая использует знак `@` в качестве сокращения для ./src. Это возможно в связи с тем, что в tsconfig.json опция `paths` определена следующим образом:

```
"paths": {
    "@/*": [
        "src/*"
    ]
}
```

Знак `@` может также быть сокращением для директивы `v-on`, которую мы использовали для обработки событий. Например, вместо написания `<button v-on:click="doSomething()">` вы можете написать `<button @click="doSomething()">`.

Следующим шагом будет добавление тега `<ul>` для отображения списка товаров в компоненте Home. Vue содержит ряд директив, которые говорят экземпляру Vue, что делать с элементом DOM. Директивы могут использоваться в шаблоне

и хуке в виде аналогичного HTML-атрибутам префикса: `v-if`, `v-show`, `v-for`, `v-bind`, `v-on` и т. д.

Здесь мы используем директиву `v-for`, чтобы произвести итерацию по массиву `products`, отображая `<li>` для каждого его элемента. Vue нужна возможность отслеживать все элементы списка, поэтому вы должны предоставить для каждого элемента атрибут уникального ключа, и мы используем директиву `v-bind:key`, определяющую ID товара в качестве такого уникального ключа. В следующем листинге показана очередная версия компонента `Home`, отображающая список продуктов.

Листинг 15.15. Отображение списка товаров в компоненте Home

```
<template>
    <div class="home">
        <h1>Products</h1>
        <ul id="prod">
            <li v-for="product in products"
                v-bind:key="product.id">
              {{ product.title }}
            </li>
        </ul>
    </div>
</template>

<style>
    ul {
        text-align: left;
    }
</style>

<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';
import {Product} from '@/product';

@Component
export default class Home extends Vue {

    products: Product[]=[];

    created() { fetch("/products.json")
        .then(response => response.json())
        .then(json => {
          this.products=json;
        },
        error => {
            console.log('Error loading products.json:', error);
        });
    }
}

</script>
```

Перебирает товары при помощи директивы v-for

Присваивает каждому элементу <li> уникальный ключ

Отображает только название товара

Выравнивает текст списка элементов

Запуск приложения приведет к отображению компонента `Home`, как показано на рис. 15.11.

Рис. 15.11. Отображение товаров

Мы реализуем в этом приложении еще одну функцию — пользователь должен иметь возможность выбирать товар из списка, а приложение должно знать, какой товар был выбран. В следующей версии компонента `Home` мы обработаем событие клика и выделим выбранный продукт голубым фоном. В листинге 15.16 показан шаблон с добавленной директивой `v-on:click`, в котором вместо `v-on` мы использовали сокращение `@`.

Листинг 15.16. Новый шаблон компонента Home

```
<template>
    <div class="home">
        <h1>Products</h1>
        <ul id="prod">
            <li v-for="product in products" v-bind:key="product.id"
                  v-bind:class="{selected: product === selectedProduct}"
                  @click = "onSelect(product)">
                {{ product.title }}
            </li>
        </ul>
    </div>
</template>
```

Использует привязку для динамического применения к выбранному элементу другого стиля

Вызывает метод onSelect, передавая данные выбранного продукта

В шаблоне листинга 15.16 есть два дополнения. Во-первых, мы добавили директиву `v-bind`, чтобы привязать CSS-селектор `selected` к элементу `<li>`, который имеет то же значение, что и свойство класса `selectedProduct`. Во-вторых, мы добавили обработчик события клика для вызова метода `onSelect()`, в котором будем устанавливать значение для `selectedProduct`, чтобы механизм привязки мог выделять соответствующий элемент списка.

В следующем листинге показан раздел `<style>` компонента `Home`, где определен класс `selected`.

Листинг 15.17. Новый стиль компонента Home

```
<style>

.home {
    display: flex;
    flex-direction: column;
}
    ul {
        text-align: left;
        display: inline-block;
        align-self: start;
    }

    .selected {   <-- Определяет стиль для выделения выбранного продукта
        background-color: lightblue
    }
</style>
```

В следующем листинге показано содержимое раздела `<script>` компонента `Home`, где расположено новое свойство `selectedProduct`.

Листинг 15.18. Раздел <script> компонента Home

```
<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';
import {Product} from '@/product';

@Component
export default class Home extends Vue {

    products: Product[]=[];
    selectedProduct: Product | null = null;   <-- Свойство selectedProduct хранит выбранный продукт

    created() {
        fetch("/products.json")
        .then(response => response.json())
        .then(json => {
          this.products=json;
        },
        error => {
            console.log('Error loading products.json:', error);
        });
    }

    onSelect(prod: Product): void {   <-- Функция-обработчик для события клика
        this.selectedProduct = prod;   <-- Устанавливает значение для selectedProduct
        }
}
</script>
```

Обратите внимание на тип свойства `selectedProduct` класса `Home`. Нам пришлось инициализировать это свойство, иначе TS стал бы ругаться: «Property

'selectedProduct' has no initializer and is not definitely assigned in the constructor» (Свойство 'selectedProduct' не имеет инициализатора и не определено в конструкторе). Эта проверка может быть либо отключена для всего проекта посредством установки в tsconfig.json опции `strictPropertyInitialization: false`, либо заглушена на уровне свойства с помощью восклицательного знака, расположенного сразу после его имени:

```
selectedProduct!: Product;
```

Объявление этого свойства как `selectedProduct: Product = null` также не будет работать, поскольку TS будет жаловаться, что вы не можете присваивать типу `Product` значение `null`. Именно поэтому мы явно разрешили `selectedProperty` быть `null`, применив тип объединения `selectedProperty: Product | null = null;`. Нам нужно инициализировать `selectedProperty` со значением, потому что иначе свойство не будет существовать в сгенерированном коде, Vue не сделает его реактивным, и мы не сможем использовать его в шаблоне компонента.

Теперь, когда пользователь кликает по продукту, вызывается метод `onSelect()`, устанавливая значение в свойстве `selectedProduct`, который используется с директивой `v-bind: class` для изменения CSS-селектора выбранного элемента списка. Когда мы устанавливаем значение для `selectedProperty`, отображается весь UL (это происходит, когда изменяется значение любого свойства класса), очищая стиль ранее выбранного элемента. На рис. 15.12 показан отображенный список, где выбран второй товар.

Выше в этой главе вы видели, как родительский компонент может передавать данные своему потомку при помощи `props`. В следующем разделе мы покажем вам, как передавать данные в процессе перехода по маршруту.

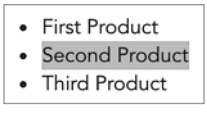


Рис. 15.12. Товар выделяется при щелчке

15.3.3. Передача данных с помощью Vue Router

Когда пользователь переходит по маршруту, ваше приложение может передавать данные целевому компоненту, используя параметры этого маршрута. В текущем разделе мы рассмотрим еще одну версию нашего приложения, отображающего список товаров. В этом случае при выборе пользователем товара приложение будет переходить к представлению его описания, показывая соответствующую информацию.

Расположено это приложение в директории router-product-detail. Выполните `npm install`, а затем `npm run serve`, и будет показан список товаров. Щелкните по одному из них, и приложение перейдет к его описанию. На рис. 15.13 показан скриншот, сделанный после того, как пользователь щелкнул по Second Product в списке. Текст `Second Product` будет выделен зеленым цветом.

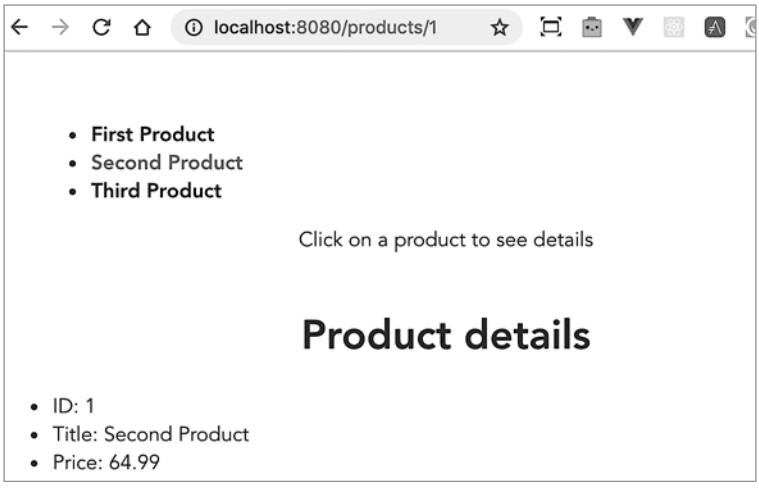


Рис. 15.13. Показ подробностей выбранного товара

В этом приложении есть только два компонента: App и `ProductDetails`. Верхняя часть изображения — это UI компонента App, а нижняя — это `ProductDetails`. Обратите внимание на сегмент URL /products/1. Маршрут, ведущий к представлению описания товара, определен как путь '/products/:productId'.

Листинг 15.19. Файл router.ts: маршрут для /products/:productId

```
import Vue from 'vue';
import Router from 'vue-router';
import ProductDetails from './views/ProductDetails.vue';

Vue.use(Router);

export default new Router({
    base: process.env.BASE_URL,
    mode: 'history',
    routes: [
        {
            path: '/products/:productId',
            component: ProductDetails,
        },
    ],
});
```

Настраивает навигацию для URL «products», сопровождаемого значением

Переходит к ProductDetails, передавая значение :productId как productid

На рис. 15.13 это не очевидно, но элементы списка товаров представлены HTML-тегами привязки, и каждая ссылка имеет URL, включающий ID выбранного товара. В следующем листинге показан шаблон компонента App, отображающий ссылку для каждого товара.

Листинг 15.20. Шаблон компонента App

```
<template>
    <div id="app">
        <div id="nav">
            <ul>
                <li v-for="product in products"
                    v-bind:key="product.id">
                    <router-link v-bind:to="'/products/' + product.id">
                        {{ product.title }}
                    </router-link>
                </li>
            </ul>
            <p>Click on a product to see details</p>
        </div>
        <router-view/>
    </div>
</template>
```

Формирует ссылку, включающую ID выбранного товара

Здесь будет отображен компонент ProductDetail

Сравните это содержимое динамически сгенерированного элемента <li> с версией из листинга 15.16. Тогда мы просто отображали текст product.title, здесь же мы отображаем следующее:

```
<router-link v-bind:to="'/products/' + product.id">
    {{ product.title }}
</router-link>
```

Код отображает заголовок, но добавляет в URL ID товара. В процессе компиляции Vue заменит <router-link> на стандартный тег привязки <a>, и весь список будет показан в области, определенной тегом <router-view>.

Раздел <script> в App.vue содержит только код для чтения файла products.json, как показано в следующем листинге. Сам же этот код мы объясняли ранее.

Листинг 15.21. Раздел <script> в App.vue

```
<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';
import { Product } from '@/product';

@Component

export default class App extends Vue {
    private products: Product[] = [];

    private created() {
```

```
        fetch('/products.json')
            .then((response) => response.json())
            .then(
                (data) => this.products = data,
                (error) => console.log('Error loading products.json:', error),
            );
    }
}
</script>
```

Тот факт, что свойство `products` компонента `App` объявлено как `private`, но все равно может использоваться в шаблоне, показывает, что Vue нужно улучшать свою поддержку TypeScript. К примеру, Angular бы не позволил вам обратиться к приватным переменным класса из шаблона.

Теперь давайте рассмотрим код компонента `ProductDetails`, которому нужно извлечь значение `productId` из маршрутизатора и отобразить описание продукта. В следующем листинге показан раздел `<template>` компонента `ProductDetails`.

Листинг 15.22. Раздел <template> из ProductDetails.vue

```
<template>
    <div>
        <h1>Product details</h1>
        <ul v-if="product">  ◄—— Условное отображение <ul>
            <li>ID: {{ product.id }}</li>
            <li>Title: {{ product.title }}</li>
            <li>Price: {{ product.price }}</li>
        </ul>
    </div>
</template>
```

Здесь мы используем директиву `v-if`, которая позволяет нам контролировать отображение элемента DOM, основываясь на некотором условии. В данном случае выражение `v-if="product"` означает «Отобрази этот `<ul>`, только если переменная `product` будет иметь утвердительное значение». Свойство `product` объявлено в классе `ProductDetails`, и оно будет иметь значение только после получения данных товара. Чтобы это произошло, пользователь должен выбрать товар в компоненте `App`. Когда он это сделает, маршрутизатор перейдет к `ProductDetails`, передавая ID товара в качестве своего параметра. Затем метод `fetchProductByID()` заполнит свойство `product`, и его информация будет отображена при помощи шаблона компонента.

В следующем листинге показан код класса `ProductDetails`, который содержит свойство `product` и три метода: `beforeRouteEnter()`, `beforeRouteUpdate()` и `fetchProductByID()`. Первые два являются навигационными хуками маршрутизатора, а последнее находит товар по ID.

Хук `beforeRouteEnter()` вызывается до подтверждения маршрута, отображающего этот компонент. Vue передает для этого хука три аргумента:

- `to` — целевой объект `Route`, к которому осуществляется переход.
- `from` — текущий `Route`, от которого происходит переход.
- `next` — функция, которая должна быть вызвана для продолжения навигации.

Листинг 15.23. Раздел <script> из ProductDetails.vue

```
<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';
import { Route } from 'vue-router';
import { Product } from '@/product';

@Component({
    async beforeRouteEnter(to: Route, from: Route, next: Function) {
        const product = await fetchProductByID(to.params.productId);
        next((component) => component.product = product);
    },

    async beforeRouteUpdate(to: Route, from: Route, next: Function) {
            this.product = await fetchProductByID(to.params.productId);
        next();
    },
})
export default class ProductDetails extends Vue {
    private product: Product | null = null;
}

async function fetchProductByID(id: string): Promise<Product> {
    const productId = parseInt(id, 10);
    const response = await fetch('/products.json');
    const products = await response.json();
    return products.find((p) => p.id === productId);
}
</script>
```

Получает данные для переданного ID товара

Навигационный хук beforeRouterEnter

Разрешает навигационный хук

Навигационный хук beforeRouterUpdate

Объявляет свойство product

Метод, получающий описание продукта

Аргумент `to` содержит свойство `params`, которое хранит значение параметра, переданного в маршрут. В данном случае в файле router.ts мы использовали имя `productId`, поэтому для получения значения этого параметра в целевом маршруте нужно использовать это же имя.

Хук `beforeRouteEnter()` не имеет доступа к экземпляру компонента `this`, поскольку на момент его вызова этот компонент еще не был создан. Тем не менее вы можете обратиться к этому экземпляру, передав обратный вызов в `next()`. Обратный вызов будет активирован после подтверждения перемещения, и экземпляр компонента будет передан в этот обратный вызов в качестве аргумента:

```
next((component) => component.product = product);
```

Здесь мы инициализируем свойство `product` экземпляра компонента. Хук `beforeRouteUpdate()` вызывается при изменении маршрута, отображающего этот компонент. В нашем приложении это происходит, когда компонент `ProductDetails` уже отображен, но пользователь кликает по другому товару

в списке. Этот хук имеет доступ к экземпляру компонента, поэтому мы можем легко присвоить значение товара к `this.product`, и аргументы для обратного вызова `next()` не потребуются.

Для простоты мы находим описание товара при помощи метода `fetchProductByID()`. Он использует API Fetch для считывания всего файла products.json и находит в нем один объект с совпадающим ID товара. Мы используем ключевые слова `async` и `await`, и вы можете сравнить этот синтаксис с его аналогом, основанным на промисе, который был показан в листинге 15.14.

На этом наше знакомство с библиотекой/фреймворком Vue завершается. В следующей главе мы создадим еще одну версию блокчейна UI, на этот раз при помощи Vue.

ИТОГИ

- Vue — это библиотека, позволяющая вам создавать компоненты UI для отображения. Она также включает организующий навигацию пользователя маршрутизатор и инструменты для генерации новых проектов, а также создания `dev` или `prod` связок для развертывания.
- Если вы предпочитаете разрабатывать компоненты UI, содержащие в одном файле и HTML, и стили, и код, то Vue для этого отлично подойдет, так как один файл содержит три раздела: `<template>` для разметки, `<script>` для кода и `<style>` для CSS.
- Осваивать Vue легче, чем React или Angular. К тому же этот инструмент предлагает аналогичные возможности.
- JS-разработчики работают с Vue, используя объектный API, но TS-разработчики могут найти использование классовых компонентов более естественным, чем создание компонентов в качестве JS-объектов. В Vue 3 у нас также будет возможность создавать и функциональные компоненты, как мы это делаем в React.
- Vue 3 находится в разработке. По ее завершении этот продукт предложит новый Composition API, который позволит вам разрабатывать функциональные компоненты UI. На момент написания книги Composition API находился на стадии запроса комментариев, при этом команда разработчиков Vue обещает простое и по большей части автоматизированное обновление с Vue 2 до Vue 3. Composition API будет дополнением к существующему объектному API.
- Как и React.js, Vue не принуждает вас преобразовывать существующее приложение в одностраничное (SPA). Вы можете поэтапно вводить Vue в имеющийся фронтенд-код, не переписывая всю базу кода за раз.

16 Разработка блокчейн-клиента на Vue.js

В этой главе:

- ✓ Обзор Vue.js-версии блокчейн-клиента.
- ✓ Запуск приложения Vue, работающего с двумя серверами в dev-режиме.
- ✓ Разбор потока данных, начиная с ввода транзакции и заканчивая генерацией блока.
- ✓ Организация взаимодействия между клиентскими компонентами блокчейна.

В предыдущей главе вы изучили основы Vue, а теперь мы рассмотрим новую версию блокчейн-приложения, где клиентская сторона будет написана в Vue. Исходный код этого веб-клиента расположен в директории blockchain/client, а сервер обмена сообщениями в blockchain/server.

Код для серверной стороны остается таким же, как и в главе 14. Функциональность самого приложения — тоже. Тем не менее его UI-часть была полностью переписана в Vue и TypeScript.

В этой главе мы не будем рассматривать функциональность блокчейн-приложения, поскольку уже делали это ранее. Рассмотрим же мы только код, специфичный для библиотеки Vue. Можете вернуться к главе 10, чтобы освежить в памяти информацию о функциональности блокчейн-клиента и сервера сообщений.

Коротко говоря, когда пользователь любого узла кликает по кнопке GENERATE BLOCK, клиентский код объявляет о начале процесса добычи, но это еще не гарантирует скорейшее ее завершение именно этим узлом. Другие узлы также могут начать добычу блока с теми же транзакциями, и все они будут использовать сервер сообщений для обмена информацией о длиннейших цепочках и достижения консенсуса относительно узла-победителя.

Сначала мы покажем вам, как запустить сервер сообщений и клиент Vue для блокчейн-приложения, а затем представим код классовых компонентов Vue.

16.1. ЗАПУСК КЛИЕНТА И СЕРВЕРА ОБМЕНА СООБЩЕНИЯМИ

Для запуска сервера откройте терминал в директории server, выполните команду `npm install` для установки зависимостей сервера, а затем команду `npm start`. Вы увидите сообщение: `Listening on http://localhost:3000`. Оставьте сервер запущенным.

Для запуска клиента откройте другое окно терминала в директории client, также выполните команду `npm install` для установки Vue и ее зависимостей, а затем команду `npm start serve`. Откройте браузер на localhost:8080, и вы увидите знакомую блокчейн-страницу, показанную на рис. 16.1. Файл App.vue содержит код для компонента верхнего уровня App, а другие файлы *.vue содержат дочерние компоненты `TransactionForm`, `PendingTransactionsPanel` и `BlocksPanel`.

Клиентская часть этого приложения была сгенерирована Vue CLI, и из списка опций мы выбрали Babel, TypeScript и классовые компоненты. На рис. 16.2 показана файловая структура директории client. Компоненты UI расположены в поддиректории components, а поддиректория lib содержит другие сценарии, которые отвечают за создание блокчейн-узлов и за обмен данными с сервером сообщений. Директория public содержит незавершенный файл index.html (он будет дополнен в процессе сборки) и файл стилей .css, в котором находятся все стили. Vue Router в этом приложении мы не использовали.

В процессе генерации проекта мы выбрали и TypeScript, и Babel. Вы видите, что файл конфигурации включает пресет `@vue/app`. Кроме этого, Babel содержит TS-плагин, поэтому наше приложение использует единый процесс компиляции, контролируемый Babel.

Использование Babel рационально по следующим причинам:

- *Режим* Modern — аналог дифференциальной загрузки в Angular. Генерируется два набора связок: один в формате ES5, а второй в ES2015. В итоге если

браузер пользователя поддерживает синтаксис ES2015, то будут загружены только соответствующие ему связки.

- *Автоопределение полифилов* — возможность, аналогичная предыдущему пункту, но решающая иную задачу. Если режим Modern касается языковых возможностей JS, то эта функция связана с API браузеров. С ее помощью Babel автоматически определяет, какие нужны полифилы, основываясь на возможностях языка, заданных в исходном коде. Это гарантирует, что в итоговую связку будет включено их минимальное число.
- *Поддержка* JSX — в отсутствие Babel мы можем использовать только HTML-шаблоны.

Директория lib содержит код, который генерирует новые блоки, запрашивает длиннейшую цепочку, уведомляет другие узлы о только что созданных блоках и приглашает других членов блокчейна начать добычу новых блоков для заданных транзакций. Все эти процессы были описаны в разделах 10.1 и 10.2. Так как код в директории lib не содержит компонентов UI, он остается в точности таким же, как в блокчейн-клиенте React из главы 14.

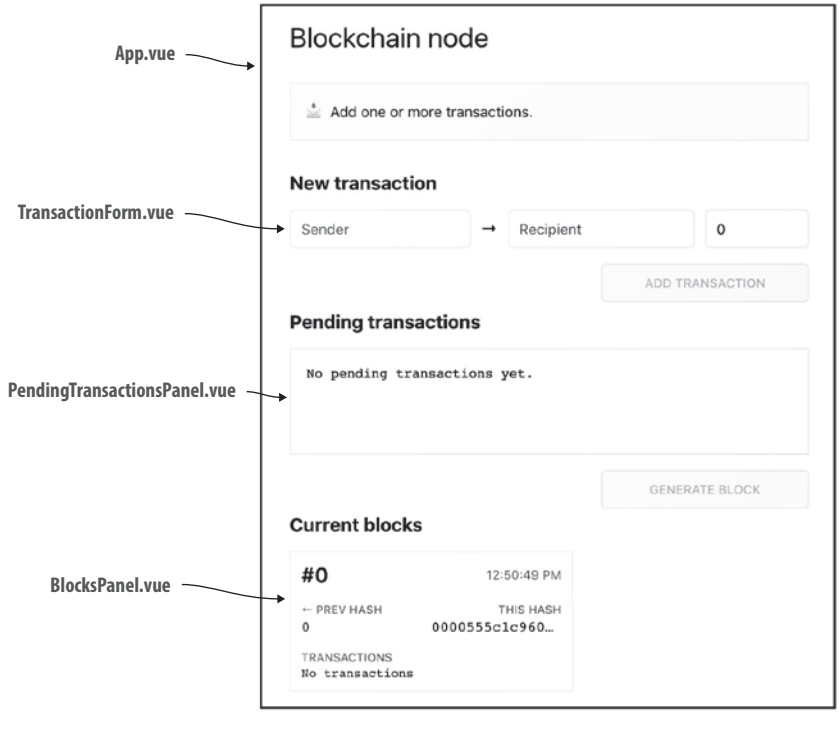


Рис. 16.1. Блокчейн-клиент запущен

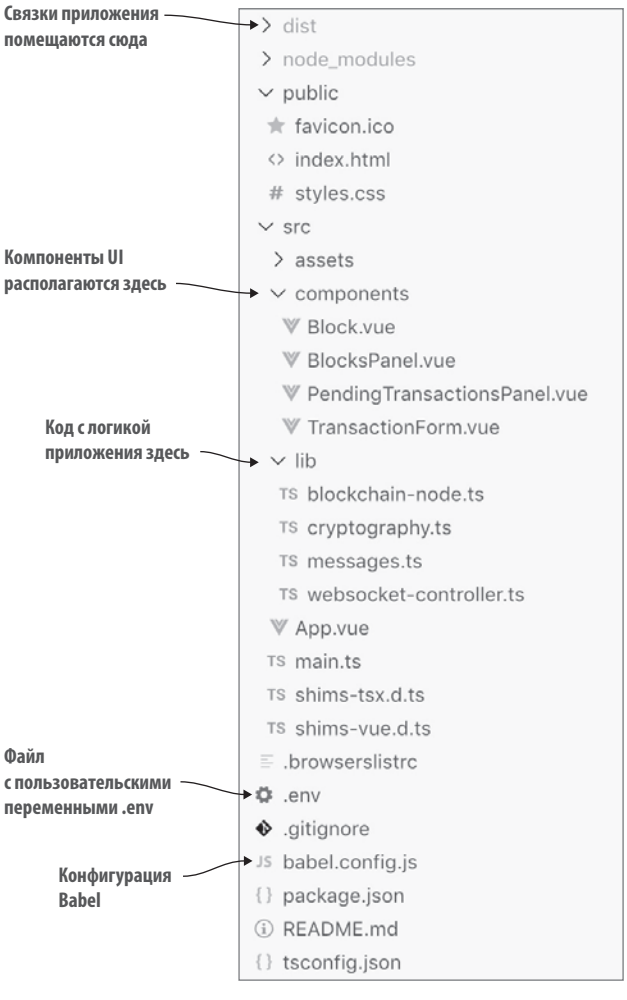


Рис. 16.2. Структура проекта

ПРИМЕЧАНИЕ Сравните рис. 16.1 с рис. 14.1, и вы увидите, что посадочная страница блокчейн-клиента разделена на компоненты UI одинаково в Vue и React.

Когда вы запускаете приложение, сгенерированное CLI, оно использует сценарий `vue-cli-service`. Вы можете найти эти команды в файле package.json:

```
"scripts": {
    "serve": "vue-cli-service serve",
    "build": "vue-cli-service build"
}
```

Сценарий `vue-cli-service` всегда считывает файл .env, который можно использовать для конфигурирования пользовательских переменных вроде имен хостов и имен портов. На рис. 16.3 показано, как dev-сервер Webpack (порт 8080) проксирует запросы к серверу сообщений (порт 3000). Это осуществляется практически так же, как и в React или Angular. Прокси определен в файле .env как VUE_APP_PROXY_HOSTNAME=localhost:3000.

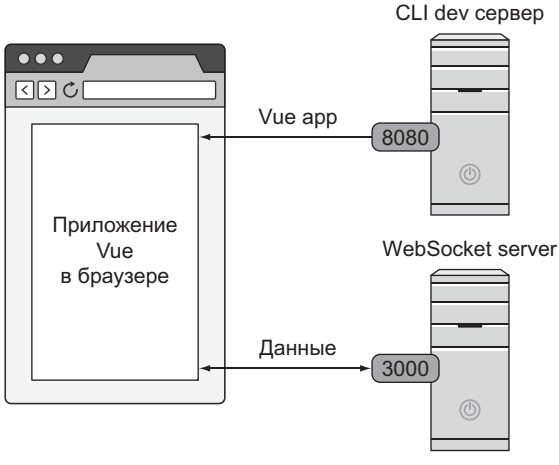


Рис. 16.3. Одно приложение, два сервера

Точка входа в блокчейн-клиент находится в файле main.ts, который встраивает экземпляр Vue в элемент DOM с ID app.

```
new Vue({
    render: h => h(App),
}).$mount('#app')
```

Как и React, Vue использует виртуальную DOM, и функция `render()` возвращает экземпляр виртуального узла VNode, который фактически является деревом элементов VNode, корневой элемент которого имеет ID app.

Именно здесь будет отображаться компонент верхнего уровня, и мы рассмотрим его код в следующем разделе.

16.2. КОМПОНЕНТ APP

Файл App.vue содержит код компонента класса App, а в его разделе `<template>` размещены три дочерних компонента: `TransactionForm`, `PendingTransactionsPanel` и `BlocksPanel`.

Листинг 16.1. Раздел <template> файла App.vue

```
<template>
    <main id="app">
        <h1>Blockchain node</h1>
        <aside><p>{{ status }}</p></aside>
        <section>
            <transaction-form
                :disabled="shouldDisableForm()"
                @add-transaction="addTransaction">
            </transaction-form>
        </section>
        <section>
            <pending-transactions-panel
                :transactions="transactions()"
                :disabled="shouldDisableGeneration()"
                @generate-block="generateBlock">
            </pending-transactions-panel>
        </section>
        <section>
            <blocks-panel :blocks="blocks()"></blocks-panel>
        </section>
    </main>
</template>
```

Компонент TransactionForm

Привязывает свойство disabled

Обрабатывает событие add-transaction

Компонент ThePendingTransactionsPanel

Компонент BlocksPanel

Выражение `:disabled="shouldDisableForm()"` является сокращением для `v-bind :disabled="shouldDisableForm()"`, и в своем контексте оно управляет свойством `disabled` компонента `TransactionForm`. Скобки после имени метода означают, что здесь мы вызываем метод `shouldDisableForm()`. Этот метод объявлен в классе `App` (см. листинг 16.2). Из шаблона могут быть вызваны все методы компонента.

Выражение `@add-transaction="addTransaction"` сообщает нам, что компонент `TransactionForm` может отправить событие `add-transaction`, и когда это произойдет, должен быть вызван метод `addTransaction()` компонента `App`. В данном случае после метода скобок не было, так как это просто ссылка на метод, который может быть вызван позже.

> **ПРИМЕЧАНИЕ** Если класс компонента назван с использованием нотации в верблюжьем регистре, вы можете использовать его в шаблоне другого компонента без изменений или применить в качестве разделителя тире. В листинге 16.1 для представления компонента `TransactionForm` мы использовали тег `<transaction-form>`, но также могли использовать тег с именем, написанным в верблюжьем регистре `<TransactionForm>`.

Выражение `@generate-block="generateBlock"` означает, что компонент `PendingTransactionsPanel` может отправить событие `generate-block`, и когда он это сделает, компонент `App` вызовет метод `generateBlock()`.

В выражении `:blocks="blocks()"` мы вызываем метод `block()`, и возвращаемое им значение будет присвоено свойству `blocks` компонента `BlocksPanel`, который отмечен декоратором `@Props` (как вы увидите в листинге 16.7).

В листинге 16.2 показан код раздела `<script>` компонента `App`. В нем мы опустили большинство методов, так как они содержали код блокчейна, уже объясненный нами ранее. Прокомментируем только часть кода, относящуюся к Vue:

Листинг 16.2. Часть раздела <script> файла App.vue

```
<script lang="ts">
// Импорты опущены в целях сокращения.
const node = new BlockchainNode();
const server = new WebsocketController();

@Component({
    components: {  <-- Перечисляет все дочерние компоненты в декораторе @Component
        BlocksPanel, PendingTransactionsPanel, TransactionForm
    }
})
export default class App extends Vue {
    status: string = '';

    blocks(): Block[] {  <-- Эта функция вызывается из шаблона
        return node.chain;
    }

    transactions(): Transaction[] {                    <-- Эта функция вызывается
        return node.pendingTransactions;                   из шаблона
    }

    shouldDisableForm(): boolean {                     <-- Эта функция вызывается
        return node.isMining || node.chainIsEmpty;         из шаблона
    }

    shouldDisableGeneration(): boolean {               <-- Эта функция вызывается
        return node.isMining || node.noPendingTransactions;    из шаблона
    }

    created() {  <-- Хук (обратный вызов) жизненного цикла компонента created()
        this.updateStatus();
        server
            .connect(this.handleServerMessages.bind(this))
            .then(this.initializeBlockchainNode.bind(this));
    }

    destroyed() {  <-- Хук (обратный вызов) жизненного цикла компонента destroyed()
        server.disconnect();
    }

    updateStatus() {  <-- Обновляет свойство status

      this.status = node.chainIsEmpty      ? '⌛Initializing
➥ the blockchain...' :
                node.isMining              ? '⌛ Mining a new block...' :
                node.noPendingTransactions ? '✉ Add one or
➥ more transactions.' :
```

```
                                        `✓ Ready to mine a new block
➥ (transactions: ${node.pendingTransactions.length}).`;
    }

    async initializeBlockchainNode(): Promise<void> {...}
    addTransaction(transaction: Transaction): void {...}
    async generateBlock(): Promise<void> {...}
    async addBlock(block: Block, notifyOthers = true): Promise<void> {...}
    handleServerMessages(message: Message) {...}
    handleGetLongestChainRequest(message: Message): void {...}
    async handleNewBlockRequest(message: Message): Promise<void> {...}
    handleNewBlockAnnouncement(message: Message): void {...}
}
</script>
```

Параметр декоратора `@Component()` является объектным литералом, и здесь мы используем синтаксис сокращения, добавленный в версии ES6. Если имя значения свойства в объектном литерале совпадает с именем идентификатора свойства, вам нет необходимости его повторять. Длинная версия объекта, представляющего дочерний компонент, выглядела бы так:

```
{
    BlocksPanel: BlocksPanel,
    PendingTransactionsPanel: PendingTransactionsPanel,
    TransactionForm: TransactionForm
}
```

ПРИМЕЧАНИЕ В листинге 16.7 в компоненте `BlocksPanel` мы будем использовать длинную нотацию. Позже объясним почему.

Вместо того чтобы объявлять `node` и `server` как свойства класса, мы держим их вне класса компонента, чтобы Vue не расширял объекты геттерами и сеттерами, необходимыми для обнаружения изменения. Мы хотели написать метод, возвращающий все узлы из блокчейна как геттер (например, `get blocks() { return node.chain; }`), но Vue не позволила шаблонам компонента работать с геттерами, и поэтому мы написали его как метод класса. То же касается и нескольких других методов класса `App`.

Хук жизненного цикла компонента `created()` вызывается Vue, когда данные и события готовы к использованию, но шаблон еще не отображен. В этом методе мы подключаемся к серверу сообщений, передавая обратный вызов `handleServerMessages()`. Когда WebSocket-соединение установлено, код инициализирует узел блокчейна (запрашивая длиннейшую цепочку, как объяснялось в главе 10) и инициализирует узел либо с уже существующими блоками, либо с первичным блоком.

ПРИМЕЧАНИЕ Здесь так же присутствует и хук жизненного цикла `mounted()`, который вызывается после отображения шаблона компонента.

Хук жизненного цикла `destroyed()` вызывается Vue, когда все внутренности компонента были уничтожены и вам просто нужно произвести финальную очистку. В нашем случае мы отключаемся от WebSocket-сервера, чтобы в памяти не осталось висячего соединения, продолжающего получать сообщения от других блоков. Обратный вызов `beforeDestroy()` мог бы стать альтернативным местом для выполнения очистки данных. После вызова `beforeDestroy()` компонент все еще остается полностью работоспособным, и вы можете применить функциональную логику для осуществления процедуры очистки.

И СНОВА О ПРОГРАММИРОВАНИИ ЧЕРЕЗ ИНТЕРФЕЙСЫ

В главе 3 мы уделили время объяснению преимуществ программирования через интерфейсы, а теперь хотели бы продемонстрировать, что происходит, если этого не делать. В Vue есть хук под названием `created()`, который является обратным вызовом, активируемым объектом Vue. Попробуйте допустить ошибку в его имени, добавив лишнюю `t`, — `creatted()`. В итоге приложение перестанет исправно работать, поскольку метод `created()`, обменивающийся данными с сервером сообщений и обновляющий переменную класса `status`, не будет существовать.

Если подобные ошибки всплывают только во время выполнения, то в использовании TypeScript нет смысла. Этот конкретный случай ясно показывает, что TS-поддержка была добавлена в Vue задним числом. А как можно было сделать иначе?

Для сравнения давайте посмотрим, как хуки жизненного цикла компонента спроектированы в Angular, где TypeScript рассматривался в роли основного языка изначально. Angular объявляет для каждого такого хука интерфейс. К примеру, есть интерфейс `OnInit`, который объявляет один метод — `ngOnInit()`. Если вы хотите, чтобы ваш компонент реализовывал этот хук, то начинаете с объявления реализации вашим классом *OnInit*, а затем пишете в этом классе реализацию `ngOnInit()`:

```
export class App implements OnInit() {OnInit

    ngOnInit() {...}
}
```

Попробуйте сделать опечатку в имени хука, добавив лишнюю `t`, — `ngOnInitt()`. Статический анализатор кода TS выделит его как ошибку, утверждая, что вы обещали реализовать методы, объявленные в интерфейсе `OnInit`, но где же `ngOnInit()`? Очевидно, что, программируя через интерфейсы, вы избегаете подобных багов.

ПРИМЕЧАНИЕ Документация Vue включает диаграмму, которая показывает все хуки жизненного цикла. Найти ее вы можете по адресу http://mng.bz/9wnx.

Метод `updateStatus()` вызывается из нескольких других методов вроде `generateBlock()` и `addBlock()`. Он обновляет свойство `status`, что приводит к повторному отображению UI, поскольку `status` является свойством компонента. Vue обертывает каждое свойство компонента в геттер и сеттер, благодаря чему и знает, когда нужно повторно отобразить UI. В документации Vue свойства компонента названы реактивными, потому что все они становятся сеттерами и геттерами, способными реагировать на изменения.

Теперь давайте рассмотрим код дочерних компонентов, начиная с `TransactionForm`.

16.3. КОМПОНЕНТ ПРЕДСТАВЛЕНИЯ TRANSACTIONFORM

На рис. 16.4 показан UI компонента `TransactionForm`, который позволяет пользователю вводить имена отправителя и получателя, а также сумму транзакции. Когда пользователь кликает по кнопке `ADD TRANSACTION`, информация должна отправляться умному родительскому компоненту `App`, знающему, как эти данные обработать. Описываемая кнопка станет доступна после заполнения формы.

Рис. 16.4. UI компонента TransactionForm

Шаблон компонента верхнего уровня `App` использует `TransactionForm` следующим образом:

```
<transaction-form
    :disabled="shouldDisableForm()"
    @add-transaction="addTransaction">
</transaction-form>
```

В листинге 16.3 показан шаблон `TransactionForm`, являющийся HTML-формой, в которой каждое поле ввода использует свойство `disabled`, управляемое родительским методом `shouldDisableForm()`. Вернитесь к листингу 16.2, и вы увидите, что `shouldDisableForm()` возвращает `true`, если узел добывается или если в блокчейне еще нет блоков.

Листинг 16.3. Раздел <template> компонента TransactionForm

```
<template>
    <div>
        <h2>New transaction</h2>
        <form class="add-transaction-form"
                @submit.prevent="handleFormSubmit">
            <input
              type="text"
              name="sender"
              placeholder="Sender"
              autoComplete="off"
              v-model.trim="formValue.sender"
              :disabled="disabled">

            <span class="hidden-xs">•</span>

            <input
              type="text"
              name="recipient"
              placeholder="Recipient"
              autoComplete="off"
              :disabled="disabled"
              v-model.trim="formValue.recipient">

            <input
              type="number"
              name="amount"
              placeholder="Amount"
              :disabled="disabled" 3((CO3-6))
              v-model.number="formValue.amount">

            <button type="submit"
                    class="ripple"
                    :disabled="!isValid() || disabled">
              ADD TRANSACTION
            </button>
        </form>
    </div>
</template>
```

Препятствует предустановленной перезагрузке страницы при событии формы submit

formValue.sender привязан к этому полю формы

Это поле контролирует переменная класса disabled

formValue.recipient привязан к этому полю формы

formValue.amount привязан к этому полю формы

По условию активирует кнопку формы submit

За этой формой стоит модель данных, хранящая все значения, введенные пользователем. Vue содержит директиву `v-model`, которая используется для создания двусторонних привязок данных между элементами формы `input`, `textarea` и `select`. «Двусторонние» означает, что если пользователь вводит или изменяет данные в поле формы, то переменной, указанной в директиве `v-model` этого поля, будет присвоено новое значение. Если значение этой переменной изменяется программно, то поле формы также обновляется.

Vue предлагает несколько модификаторов событий, и здесь мы используем `.prevent`. В Vue выражение `@submit.prevent="handleFormSubmit"` означает

«Предотвратить предустановленную обработку кнопки формы submit. Вызвать вместо этого метод handleFormSubmit()».

Каждое поле ввода привязано к одному из свойств объекта formValue, который играет роль модели формы и определен в разделе сценария этого компонента как formValue: Transaction. Тип Transaction определен так:

```
export interface Transaction {
    readonly sender: string;
    readonly recipient: string;
    readonly amount: number;
}
```

Например, следующая строка использует директиву v-model для отображения поля формы sender (отправитель) в свойство sender объекта formValue:

```
v-model="formValue.sender"
```

Но директива v-model поддерживает модификаторы, поэтому мы написали ее так:

```
v-model.trim="formValue.sender"
```

Модификатор trim автоматически обрезает пустое пространство при пользовательском вводе. Мы также использовали модификатор number в v-model.number="formValue.amount", чтобы обеспечить автоматическое приведение типа значения к числу в процессе синхронизации значения из поля amount со свойством formValue.amount.

В листинге 16.4 показан раздел <script> файла TransactionForm.vue. Он определяет и инициализирует объект formValue. У него также есть метод isValid() для проверки действительности формы и метод handleFormSubmit(), который вызывается, когда пользователь кликает по кнопке ADD TRANSACTION.

Здесь мы используем декоратор @Prop с аргументом Boolean, давая Vue команду привести переданное значение (строчные HTML-данные) к этому типу.

Метод isValid() возвращает true, только если пользователь ввел в форму все три значения. В результате чего будет активирована кнопка ADD TRANSACTION, и если пользователь по ней кликнет, то метод handleFormSubmit() отправит событие add-transaction родительскому компоненту App, который вызовет метод addTransaction().

Дочерний компонент может отправлять данные своему родителю, используя метод $emit(), и мы вызываем его с полезной нагрузкой (…this.formValue). Здесь мы клонируем объект formValue, используя JS-оператор распространения. Метод addTransaction() в компоненте App получит объект типа Transaction

и добавит его в список ожидающих транзакций, обслуживаемый компонентом `PendingTransactionsPanel`.

Листинг 16.4. Раздел <script> файла TransactionForm.vue

```
<script lang="ts">
import { Component, Prop, Vue } from 'vue-property-decorator';
import { Transaction } from '../lib/blockchain-node';

@Component
export default class TransactionForm extends Vue {

    @Prop(Boolean) readonly disabled: boolean;

    formValue: Transaction = this.defaultFormValue();

    isValid() {
        return (
            this.formValue.sender &&
            this.formValue.recipient &&
            this.formValue.amount > 0
        );
    }

    handleFormSubmit() {
        this.$emit('add-transaction', { ...this.formValue });

        this.formValue = this.defaultFormValue();
    }

    private defaultFormValue(): Transaction {
        return {
            sender: '',
            recipient: '',
            amount: 0
        };
    }
}
</script>
```

Значение prop передается родителем

Инициализирует модель формы со значениями по умолчанию

Действительна ли форма?

Обрабатывает клик по кнопке ADD TRANSACTION

Отправляет событие родителю

Сбрасывает форму

Значение модели формы по умолчанию

Мы призываем вас прогнать это приложение через отладчик браузера, установив точку останова в методе `handleFormSubmit()`, принадлежащем `TransactionFrom`. На рис. 16.5 показан скриншот, сделанный после того, как мы в качестве отправителя, получателя и суммы ввели `"Alex"`, `"Mary"` и `"100"`, после чего кликнули на кнопке `ADD TRANSACTION`. Отладчик Chrome остановил выполнение на точке останова в методе `handleFormSubmit()`. По умолчанию в файле tsconfig.json опция генерации карт кода включена, следовательно, вы также можете производить отладку и TS-кода.

Чтобы найти в отладчике исходники TS, откройте вкладку Source в инструментах разработчика и найдите раздел Webpack в панели слева. Затем найдите иконку папки с именем-точкой, а в ней перейдите в подпапку src. Вы можете увидеть

множество файлов с одинаковым именем, заканчивающимся на разные числа, как показано на рис. 16.5. Причина этого в горячей замене модулей: каждый раз, когда вы меняете файл, Webpack передает его новую версию, но с другим суффиксом в имени, поэтому вам потребуется несколько секунд, чтобы найти именно файл с TypeScript.

В середине рис. 16.5 показана точка останова на строке 59. В правой панели мы добавили `this.formValue` в раздел Watch, и вы можете видеть там значения `100`, `Mary` и `Alex`.

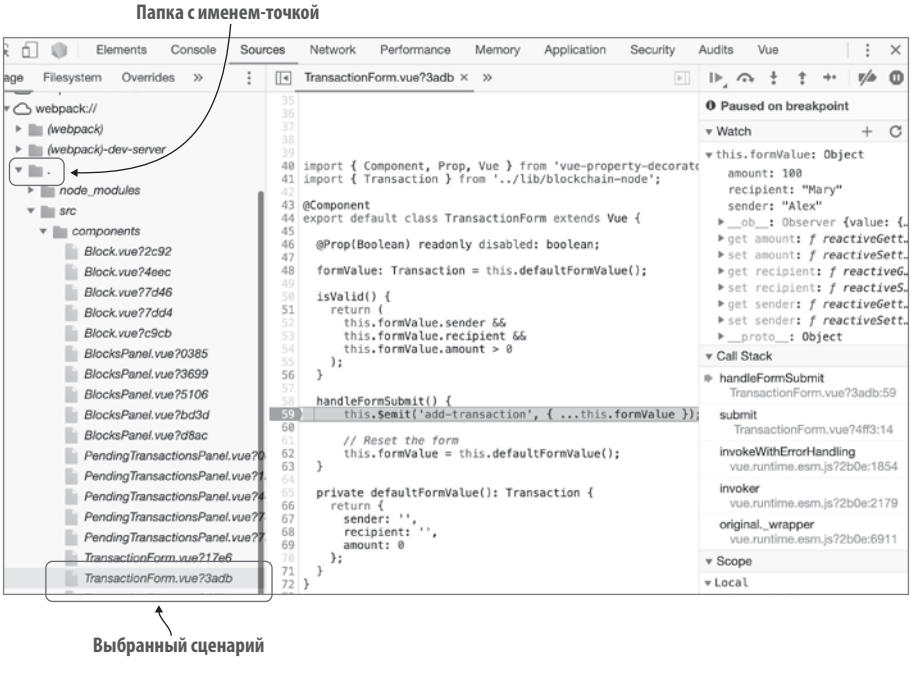


Рис. 16.5. Отладка компонента TransactionForm

Щелкните на иконке `Step Over`, и отладчик перенесет вас к методу `addTransaction()` в компоненте App, где вы увидите, что объект с этими значениями получен. После этого текущая транзакция добавляется в Node список ожидающих транзакций, как показано в следующем листинге.

Листинг 16.5. Метод addTransaction компонента App

```
addTransaction(transaction: Transaction): void {
    node.addTransaction(transaction);
    this.updateStatus();
}
```

Вызов метода `this.updateStatus()` изменяет переменную класса `status`, что вызывает повторное отображение компонента `PendinfTransactionsPanel`, о котором мы и поговорим далее.

16.4. КОМПОНЕНТ ПРЕДСТАВЛЕНИЯ PENDINGTRANSACTIONSPANEL

`PendingTransactionsPanel` является компонентом представления, содержащим `props transactions`. Его родительский компонент `App` передает массив транзакций следующим образом:

```
<pending-transactions-panel
    :transactions="transactions()"  ◄──  Передает массив транзакций
    :disabled="shouldDisableGeneration()"
@generate-block="generateBlock">
```

В шаблоне `PendingTransactionsPanel` мы активируем функцию `formattedTransactions()` (см. листинг 16.6), которая перебирает массив `Transactions[]`, форматируя и отображая его элементы в качестве строк в элементе `<pre>`.

Компонент `PendingTransactionsPanel` помимо этого может инициировать генерацию блока, когда пользователь щелкает по кнопке `GENERATE BLOCK`. Поскольку это компонент представления, он не знает, как генерировать блок, но может отправить событие `generate-block` своему родителю, который уже решит, что с ним делать. В следующем листинге показан код компонента `PendingTransactionsPanel`.

Листинг 16.6. Файл PendingTransactionsPanel.vue

```
<template>
    <div>
        <h2>Pending transactions</h2>
        <pre class="pending-transactions list">{{
➥ formattedTransactions() || 'No pending transactions yet.' }}
        </pre>  ◄──  Здесь показываются отформатированные транзакции
        <div class="pending-transactions form">
            <button class="ripple"
                    type="button"
                    :disabled="disabled"
                    @click="generateBlock()">  ◄──┐
                GENERATE BLOCK                    │  Клик по кнопке GENERATE BLOCK
            </button>                             │  вызывает generateBlock()
        </div>
        <div class="clear"></div>
    </div>
</template>

<script lang="ts">
```

```
import { Component, Prop, Vue } from 'vue-property-decorator';
import { Transaction } from '@/lib/blockchain-node';

@Component
export default class PendingTransactionsPanel extends Vue {

    @Prop(Boolean) readonly disabled: boolean;
    @Prop({ type: Array, required: true }) readonly transactions: Transaction[];

    formattedTransactions(): string {
        return this.transactions
            .map((t: any) =>`${t.sender} • ${t.recipient}: $${t.amount}`)
            .join('\n');
    }

    generateBlock(): void {
        this.$emit('generate-block');
    }
}
</script>
```

Значение для этих props представлено в виде Boolean

Форматирует ожидающие транзакции

Отправляет событие generate-block

Один из декораторов `@Prop` содержит параметр `{ type: Array, required: true }`. Так мы даем Vue команду считывать переданное значение как массив, и значение этого `props` является обязательным

На рис. 16.6 показано отображение компонента `PendingTransactionsPanel` с двумя ожидающими транзакциями, полученными от компонента `TransactionForm`.

Рис. 16.6. UI компонента PendingTransactionsPanel

Щелчок по кнопке `GENERATE BLOCK` в `PendingTransactionsPanel` должен запустить процесс генерации блока. Поскольку компонент `App` имеет доступ к массиву `Transactions[]`, метод `generateBlock()` просто отправляет событие. `App` является хостом для `PendingTransactionsPanel`:

```
<pending-transactions-panel
    :transactions="transactions()"
    :disabled="shouldDisableGeneration()"
    @generate-block="generateBlock">
</pending-transactions-panel>
```

Когда событие `generate-block` отправлено, вызывается метод `generateBlock()` компонента `App`, который, в частности, обновляет свойство `status`. Это приводит к повторному отображению UI, поскольку `status` является свойством компонента. Компонент `BlocksPanel` получает все блоки через свой `props blocks`.

Теперь давайте посмотрим, что происходит в `BlocksPanel`.

16.5. КОМПОНЕНТЫ ПРЕДСТАВЛЕНИЯ BLOCKSPANEL И BLOCK

Когда пользователь нажимает кнопку `GENERATE BLOCK` в компоненте `PendingTransactionsPanel`, все активные узлы блокчейна начинают процесс добычи. После достижения консенсуса новый блок будет добавлен в блокчейн и отображен в компоненте `BlocksPanel`, который играет роль контейнера дочерних компонентов `Block`. На рис. 16.7 показано отображение `BlocksPanel` с блокчейном из двух блоков.

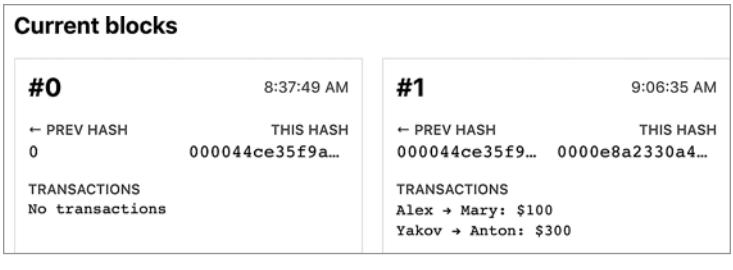


Рис. 16.7. UI компонента BlocksPanel

В процессе добычи блока и достижения консенсуса задействованы экземпляры `BlockchainNode` и `WebsocketController`, но поскольку `BlocksPanel` является компонентом представления, он не общается напрямую ни с одним из этих объектов. Эта работа делегируется умному компоненту `App`.

Компонент `BlocksPanel` не отправляет данные своему родителю, его цель отобразить блокчейн, полученный через `props blocks`. Компонент `App` вызывает свой метод `blocks()` и привязывает возвращаемое значение (коллекцию существующих в блокчейне блоков) к свойству `blocks` компонента `BlocksPanel` (двоеточие используется для привязки):

```
<blocks-panel :blocks="blocks()"></blocks-panel>
```

В следующем листинге показан код компонента `BlocksPanel`. Обратите внимание, что объявление свойства `blocks` оформлено декоратором `@Prop`, что говорит о поступлении значений от родителя.

Листинг 16.7. BlocksPanel.vue

```
<template>
    <div>
        <h2>Current blocks</h2>
        <div class="blocks">
            <div class="blocks ribbon">
                <block v-for="(b, i) in blocks"
                       :key="b.hash"
                       :index="i"
                       :block="b">
                </block>
            </div>
            <div class="blocks overlay"></div>
        </div>
    </div>
</template>

<script lang="ts">
import { Component, Prop, Vue } from 'vue-property-decorator';
import { Block } from '@/lib/blockchain-node';

import BlockComponent from './Block.vue';

@Component({
    components: {
        Block: BlockComponent
    }
})
export default class BlocksPanel extends Vue {
    @Prop({ type: Array, required: true }) readonly blocks: Block[];
}
</script>
```

Перебирает массив блоков и отображает BlockComponents

Присваивает каждому отображенному блоку уникальный ключ

Передает значение в props компонента Block

Импортирует интерфейс Block

Импортирует компонент Block

Регистрирует потомка BlockComponent под именем Block

Объявляет декорированное свойство blocks

Компонент представления `BlocksPanel` использует директиву `v-for` для перебора массива `blocks`, отображая компонент `Block` для каждого его элемента. В React-версии приложения для отображения блоков мы использовали метод `Array.map()` (см. листинг 14.22 в главе 14). Почему же здесь для этого мы использовали специальный атрибут HTML-элемента `v-for`? Причина в том, что в React мы задействовали JSX, который расширил наши возможности использования JavaScript. Здесь же HTML-шаблоны позволяют нам использовать только специальные атрибуты тегов. Другими словами, в React вы можете использовать для отображения JS, в то время как в Vue это будет статическая строка.

> **ПРИМЕЧАНИЕ** В этой главе мы не использовали JSX, чтобы показать вам, как работать с HTML-шаблонами, но в документации к Vue даны рекомендации относительно использования JSX: http://mng.bz/j58z. Рассмотрите применение JSX, если предпочитаете использовать в шаблонах не HTML, а JavaScript.

Обратите внимание, что здесь для определения дочернего компонента мы не задействовали сокращенный синтаксис объектного литерала:

```
components: {
    Block: BlockComponent
}
```

Имя свойства слева (`Block`) определяет имя компонента, который вы можете использовать в шаблоне: `<block>`. HTML нечувствителен к регистру, и если в сценарии имя компонента указано как `Block`, то в HTML на него можно сослаться как на `<block>`.

> **ПРИМЕЧАНИЕ** Подробнее с этой темой можно ознакомиться в дискуссии на GitHub «HTML case sensitivity workaround» (Обхождение проблемы нечувствительности регистра в HTML) по ссылке http://mng.bz/WOv4.

Нам пришлось использовать здесь длинный вариант синтаксиса из-за конфликта имен: мы объявили интерфейс `Block` в директории lib/blocikchain-node, и как вы увидите в листинге 16.8, файл Block.vue объявляет компонент с таким же именем `Block`. Сначала мы попытались использовать сокращенный ES6-синтаксис для объектных литералов:

```
components: {
    Block
}
```

В итоге Vue начал жаловаться, что не узнает тег `<block>`, но поскольку мы использовали для экспорта `class Block` ключевое слово `default`, то могли импортировать его под любым именем. Как вы видели ранее в листинге 16.7, мы назвали его `BlockComponent`. В нем мы говорим Vue о том, что у нас есть компонент под названием `Block`, но его код мы импортировали под именем `BlockComponent`.

> **ПРИМЕЧАНИЕ** Более простым решением этого конфликта имен было бы изменение тега шаблона с `<block>` на `<block-component>`, но мы хотели использовать этот конфликт, чтобы показать случай, где сокращенный синтаксис для объектных литералов бы не сработал.

В следующем листинге показан код компонента `Block`.

Листинг 16.8. Block.vue

```
<template>
    <div class="block">
        <div class="block header">
            <span class="block index">#{{ index }}</span>
            <span class="block timestamp">{{ timestamp() }}</span>
        </div>
        <div class="block hashes">
            <div class="block hash">
                <div class="block label">• PREV HASH</div>
                <div class="block hash-value">{{ block.previousHash }}</div>
            </div>
            <div class="block hash">
                <div class="block label">THIS HASH</div>
                <div class="block hash-value">{{ block.hash }}</div>
            </div>
        </div>
        <div>
            <div class="block label">TRANSACTIONS</div>
            <pre class="block transactions">{{ formattedTransactions() ||
➥ 'No transactions' }}</pre>
        </div>
    </div>
</template>

<script lang="ts">
import { Component, Prop, Vue } from 'vue-property-decorator';
import { Block as ChainBlock, Transaction }
➥ from '@/lib/blockchain-node';  ◄── Импортирует, присваивая Block псевдоним

@Component
export default class Block extends Vue {
    @Prop(Number) readonly index: number;  ◄── Порядковый номер блока

    @Prop({ type: Object, required: true }) readonly block: ChainBlock;  ◄──
                                                                  Данные блока
    timestamp() {
        return new Date(this.block.timestamp).toLocaleTimeString();
    }

    formattedTransactions(): string {
        return this.block.transactions
            .map((t: Transaction) =>
                  `${t.sender} • ${t.recipient}: $${t.amount}`)
            .join('\n');
    }
}
</script>
```

В листинге 16.8 нам также пришлось разрешать конфликт имен между классом компонента `Block` и интерфейсом, имеющим такое же имя. Здесь мы используем другой синтаксис:

```
import { Block as ChainBlock} from '@/lib/blockchain-node';
```

В этом случае интерфейс `Block` был экспортирован как проименованный экспорт файла blockchain-node.ts, поэтому мы не могли использовать любое имя. Нам пришлось написать `import { Block as ChainBlock }`, чтобы ввести псевдоним `ChainBlock`. Обратите внимание на фигурные скобки — их нужно использовать при импорте именованных экспортов.

`Block` является самым простым компонентом этого приложения. Как вы видели на рис. 16.7, он просто отображает данные одного блока. На этом наше рассмотрение блокчейн-клиента, написанного в Vue и TypeScript, завершается.

ИТОГИ

- Поскольку Vue генерирует сеттеры и геттеры для каждого свойства компонента, процесс обнаружения изменений существенно упрощен. Изменение значения свойства компонента служит сигналом для повторного отображения UI.
- Как и в случае с React или Angular, UI приложения Vue состоит из умных компонентов и компонентов представления. Не размещайте логику приложения в последних, так как они предназначены для представления данных или обеспечения взаимодействия с пользователем (отправка пользовательского ввода другим компонентам).
- В Vue родительский компонент передает данные своему потомку посредством props. Потомок отправляет данные родителю через события с полезной нагрузкой или без нее.
- Vue CLI генерирует проекты, внутренне использующие для связывания Webpack. В разработке Webpack dev-сервер поддерживает автоматическую перекомпиляцию и горячую замену модулей, при которой новый код отправляется браузеру без перезагрузки страницы.

ЭПИЛОГ

В этой книге мы показали вам основные синтаксические конструкции TypeScript наряду с несколькими приложениями, использующими этот язык. Все основные веб-фреймворки поддерживают TS, и вам не нужно ждать новых проектов, чтобы начать его использовать, — можно просто постепенно вводить этот язык в уже существующие проекты JavaScript. В качестве бонуса мы объяснили основы технологии блокчейн, показав несколько версий TS-приложений, ее использующих.

Надеемся, что после прочтения этой книги вы поймете, почему TS так стремительно завоевывает популярность. Мы верим, что этот язык еще долго будет в лидерах, и желаем вам приятного опыта его использования!

Приложение А
Современный JavaScript

ECMAScript — это стандарт для скриптовых языков, эволюция которого регулируется комитетом TC39. Синтаксис ECMAScript реализован в нескольких языках, в наибольшей же степени в JavaScript. Начиная с шестой редакции (она же ES6, или ES2015), TC39 выпускают новые спецификации ежегодно.

Вы можете ознакомиться с последней версией по ссылке http://mng.bz/8zoZ, но именно ECMAScript2015 внесла существенные дополнения в JavaScript. Большая часть синтаксиса, рассмотренного в этом приложении, была добавлена в этой спецификации, и большинство браузеров полноценно поддерживают именно ее (см. http://mng.bz/ao59). Даже если пользователи вашего приложения работают в более старых браузерах, вы все равно можете производить разработку в ES6/7/8/9 и затем просто использовать транспилятор вроде TypeScript или Babel, чтобы преобразовать код, использующий последний синтаксис ECMAScript, в версию ES5.

Мы предполагаем, что вы знакомы с версией ES5 синтаксиса, и рассмотрим только избранные возможности, добавленные в ECMAScript, начиная с 2015 года.

А.1. КАК ЗАПУСКАТЬ ОБРАЗЦЫ КОДА

Образцы кода для этого приложения оформлены в виде JS-файлов с расширением .js, и мы будем использовать для их запуска сайт CodePen (https://codepen.io). Этот ресурс позволят быстро писать, тестировать и обмениваться приложениями,

использующими HTML, CSS и JavaScript. Мы предоставим ссылки CodePen на большинство образцов, чтобы вы могли проследовать по ним, увидеть выбранный образец в действии и при желании внести в него изменения. Если образец производит вывод в консоль, просто щелкните по Console в нижней части CodePen, чтобы его увидеть.

Теперь давайте рассмотрим, как некоторые возможности ECMAScript реализованы в JavaScript.

A.2. КЛЮЧЕВЫЕ СЛОВА LET И CONST

Ключевые слова `let` и `const` предназначены для использования вместо их коллеги `var`. Давайте начнем с рассмотрения сложностей, связанных с этим ключевым словом.

A.2.1. Ключевое слово var и поднятие

В ES5 и более старых версиях JavaScript вы бы использовали `var` для объявления переменных, а движок JS перемещал бы это объявление вверх контекста их выполнения (например, функции). Этот процесс называется поднятием (подробнее читайте по ссылке http://mng.bz/3x9w).

Из-за поднятия, если вы объявляли переменную внутри блока кода (например, внутри фигурных скобок инструкции `if`), эта переменная также становилась видимой вне этого блока. Посмотрите на следующий пример, где мы объявляем переменную `i` внутри цикла `for`, но также используем ее извне:

```
function foo() {

    for (var i=0; i<10; i++) {

    }

    console.log("i=" + i);

}

foo();
```

Выполнение этого кода выведет `i=10`. Переменная `i` по-прежнему доступна вне цикла, хотя и кажется, что подразумевалось ее использование только внутри него. JavaScript автоматически поднимает объявление переменной в верхнюю часть функции.

В предыдущем примере поднятие не привело к появлению проблем, поскольку использовалась всего одна переменная `i`. Тем не менее если бы внутри и вне функции были объявлены две переменные с одинаковым именем, это могло бы привести к запутанному поведению. Рассмотрите следующий листинг, объявляющий переменную `customer` в глобальной области. Чуть позже мы введем другую переменную `customer`, но уже в локальной области, но пока что оставим ее закомментированной.

Листинг А.1. Поднятие объявления переменной

```
var customer = "Joe";
(function () {
    console.log("The name of the customer inside the function is " +
➥ customer);
    /* if (true) {
            var customer = "Mary";
        } */
})();
console.log("The name of the customer outside the function is " + customer);
```

Глобальная переменная `customer` видима внутри и вне функции, и выполнение этого кода приведет к следующему выводу:

```
The name of the customer inside the function is Joe
The name of the customer outside the function is Joe
```

Раскомментируйте инструкцию `if`, объявляющую и инициализирующую переменную `customer` внутри фигурных скобок. Теперь у нас есть две переменные с одинаковым именем — одна в глобальной области и другая в области функции. В данном случае вывод консоли будет уже иным:

```
The name of the customer inside the function is undefined
The name of the customer outside the function is Joe
```

Все потому, что в ES5 объявления переменных поднимаются в верхнюю часть области (в данном случае это выражение в самых верхних скобках), но с инициализацией этого не происходит. Когда переменная создана, ее начальное значение — `undefined`. Объявление второй неопределенной переменной `customer` было поднято в верхнюю часть функции, и `console.log()` вывела в консоль значение переменной, объявленной внутри функции, которая перекрыла значение глобальной переменной `customer`.

ПРИМЕЧАНИЕ Этот пример на CodePen: http://mng.bz/cK9y.

Объявления функций также поднимаются, поэтому фактически вы можете вызвать функцию до ее объявления:

```
doSomething();

function doSomething() {
    console.log("I'm doing something");
}
```

С другой стороны, выражения функций считаются инициализациями переменных, поэтому не поднимаются. Следующий фрагмент кода произведет `undefined` для переменной `doSomething`:

```
doSomething();

var doSomething = function() {
    console.log("I'm doing something");
}
```

Теперь давайте посмотрим, как ключевые слова `let` и `const` могут посодействовать вам в разделении областей видимости.

A.2.2. let и const для работы в области блока

ES6 устраняет путаницу, связанную с поднятием, при помощи ключевых слов `let` и `const`. `let` используется, когда вам нужно объявить переменную, которая может быть инициализирована с одним значением, а затем получить присвоением другое. Используя же ключевое слово `const`, вы можете присвоить значение к идентификатору только один раз, после чего повторное присваивание не допускается.

При этом не стоит считать, что `const` представляет неизменяемые значения. Квалификатор `const` просто означает, что оно может быть инициализировано только раз. Но это не значит, что свойство объекта, присвоенного идентификатору `const`, не может быть изменено. Например, следующее выражение `const products` представляет массив продуктов, и вы можете изменять отдельные свойства этих объектов после инициализации `const products`:

```
const products = [
    { id: 1, description: 'Product 1' },
     {id: 2, description: 'Product 2'}
]

products[0].id = 111;
products[1].description = 'Product 222';
```

Объявление переменных с ключевыми словами `let` и `const` вместо `var` ограничивает область видимости переменных их блоком. В следующем листинге показан пример.

Листинг А.2. Ограничение области видимости переменных блоком

```
const customer = "Joe";

    (function () {
        console.log("1. Inside the function " + customer);
        if (true) {
            const customer = "Mary";
            console.log("2. Inside the block " + customer);
        }
    })();

console.log("3. In the global scope " + customer);
```

Теперь две переменные `customer` имеют разные области видимости и значения, поэтому программа выведет в консоль следующее:

```
The name of the customer inside the function is Joe
The name of the customer inside the block is Mary
The name of the customer in the global scope is Joe
```

Проще говоря, если вы разрабатываете новое приложение, не используйте `var`. Используйте вместо него `const` или `let`.

> **СОВЕТ** Если вы попытаетесь использовать переменные, определенные с `let` или `const`, до их объявления, то получите ошибку среды выполнения `ReferenceError`. Это называется *временной мертвой зоной*, где вы не можете обращаться к переменной до ее определения.

В предыдущем примере кода мы должны были использовать `const` вместо `let`, так как мы не присваивали повторно значения идентификатору `customer`.

> **ПРИМЕЧАНИЕ** Этот пример на CodePen: http://mng.bz/fkJd.

> **СОВЕТ** Если вам нужно объявить идентификатор, определяйте его как `const`. Никогда не поздно изменить его на `let`, если ему вдруг потребуется присвоить значение.

А.3. ШАБЛОННЫЕ ЛИТЕРАЛЫ

Строчные литералы теперь могут содержать вложенные выражения. Эта возможность известна как *интерполяция*. В ES5 же для создания строки, содержащей строчные литералы совместно со значениями переменных, вам бы пришлось использовать конкатенацию:

```
const customerName = "John Smith";
console.log("Hello" + customerName);
```

Теперь вы можете использовать шаблонные литералы, являющиеся строками, заключенными в обратные кавычки. Вы можете вкладывать выражения внутрь литерала, размещая их внутри фигурных скобок, имеющих в виде префикса значок доллара. В следующем фрагменте кода значение переменной `customerName` вложено в строчный литерал:

```
const customerName = "John Smith";
console.log(`Hello ${customerName}`);

function getCustomer() {
    return "Allan Lou";
}
console.log(`Hello ${getCustomer()}`);
```

Вывод этого кода показан здесь:

```
Hello John Smith
Hello Allan Lou
```

ПРИМЕЧАНИЕ Этот пример на CodePen: http://mng.bz/Ey30.

В предыдущем образце кода мы вложили значение переменной `customerName` в шаблонный литерал, а затем вложили значение, возвращенное функцией `getCustomer()`. Вы можете использовать в фигурных скобках любое рабочее выражение JavaScript.

Строки могут занимать по несколько строчек кода. Используя же обратные кавычки, вы можете писать многострочные строки, не прибегая к их конкатенации:

```
const message = `Please enter a password that
                has at least 8 characters and
                includes a capital letter`;

console.log(message);
```

Итоговая строка будет рассматривать все пробелы как часть строки, поэтому вывод будет выглядеть так:

```
Please enter a password that
               has at least 8 characters and
               includes a capital letter
```

ПРИМЕЧАНИЕ Этот пример на CodePen: http://mng.bz/1SSP.

А.3.1. Размеченные шаблонные строки

Если шаблонной строке предшествует имя функции, эта строка вычисляется первой, а затем передается в функцию для дальнейшей обработки. Строчные

части шаблона передаются в функцию в виде массива, а все вычисленные в шаблоне выражения передаются в виде отдельных аргументов. Синтаксис выглядит несколько необычно, потому что вы не используете скобки как в стандартных вызовах JS-функций.

В следующем фрагменте кода функция-тег `mytag` сопровождена шаблонной строкой:

```
mytag`Hello ${name}`;
```

Значение переменной `name` будет вычислено и передано в функцию `mytag`.

Давайте напишем простой размеченный шаблон, выводящий сумму со знаком валюты, который будет зависеть от переменной `region`. Если значение `region` будет `1`, мы оставим сумму неизменной и поставим перед ней значок доллара. Если же значение `region` будет `2`, нам нужно конвертировать сумму, применив в качестве обменного курса `0,9`, и поставить перед полученным результатом значок евро. Шаблонная строка будет выглядеть так:

```
`You've earned ${region} ${amount}!`
```

Давайте вызовем функцию-тег `currencyAdjustment`. Размеченная шаблонная строка будет выглядеть так:

```
currencyAdjustment`You've earned ${region} ${amount}!`
```

Наша функция `currncyAdjustment` будет получать три аргумента: первый будет представлять все строчные части из шаблонной строки, второй — регион, а третий — сумму. Вы же можете добавить любое число аргументов после первого. Законченный пример выглядит так:

```
function currencyAdjustment(stringParts, region, amount) {
        console.log( stringParts);
        console.log( region );
        console.log( amount );

    let sign;
    if (region === 1){
      sign="$"
    } else{
      sign='\u20AC';     // Значок евро.
      amount=0.9*amount; // Преобразует в евро, на основе обменного курса 0.9.
    }
    return `${stringParts[0]}${sign}${amount}${stringParts[2]}`;
}

const amount = 100;
const region = 2; // Europe: 2, USA: 1
```

```
const message = currencyAdjustment`You've earned ${region} ${amount}!`
console.log(message);
```

Функция `currencyAdjustment` получит строку со вложенными `region` и `amount`, а затем считает шаблон, отделив от этих значений строчные части (пробелы также считаются частями строки). Сначала мы выведем эти значения для наглядности. Затем эта функция проверит регион, применит конвертацию и вернет новую шаблонную строку. Выполнение предыдущего примера кода произведет следующий вывод:

```
["You've earned "," ","!"]
2
100
You've earned ?90!
```

ПРИМЕЧАНИЕ Этот пример на CodePen: http://mng.bz/E1Yo.

В разделе 10.6.2 мы рассмотрели код веб-клиента, использующий lit-html, который сам использует размеченные шаблонные строки.

А.4. ОПЦИОНАЛЬНЫЕ ПАРАМЕТРЫ И ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ

Вы можете указать значения для параметров функции (аргументы), которые будут использованы, если в процессе вызова функции значение передано не будет. Предположим, вы пишете функцию для вычисления налога, получающую два аргумента: ежегодный доход и штат проживания гражданина. Если штат проживания не будет передан, мы хотим использовать по умолчанию Флориду.

В ES5 нам бы пришлось начать тело функции с проверки, был ли передан аргумент штата, и только в противном случае мы бы использовали Флориду:

```
function calcTaxES5(income, state) {

    state = state || "Florida";

    console.log("ES5. Calculating tax for the resident of " + state +
                                        " with the income " + income);
}

calcTaxES5(50000);
```

Вот вывод этого кода:

```
"ES5. Calculating tax for the resident of Florida with the income 50000"
```

Начиная же с версии ES6, вы можете указать значение по умолчанию прямо в сигнатуре функции:

```
function calcTaxES6(income, state = "Florida") {

    console.log("ES6. Calculating tax for the resident of " + state +
                                        " with the income " + income);
}

calcTaxES6(50000);
```

ПРИМЕЧАНИЕ Этот пример на CodePen: http://mng.bz/U51z.

А.5. ВЫРАЖЕНИЯ СТРЕЛОЧНЫХ ФУНКЦИЙ

Стрелочные функции предоставляют сокращенную нотацию для анонимных функций и добавляют для переменной `this` лексическую область. Синтаксис стрелочных функций состоит из аргументов, знака стрелки (`=>`) и тела функции. Если тело функции состоит из одного выражения, фигурные скобки использовать не нужно. Если такая функция из одного выражения возвращает значение, нет необходимости писать инструкцию `return`, поскольку результат будет возвращен неявно:

```
let sum = (arg1, arg2) => arg1 + arg2;
```

Тело же многострочного выражения стрелочной функции должно быть заключено в фигурные скобки, и инструкцию `return` нужно использовать явно:

```
(arg1, arg2) => {
    // делает что-то.
    return someResult;
}
```

Если стрелочная функция не имеет аргументов, используйте пустые скобки:

```
() => {
    // делает что-то.
    return someResult;
}
```

Если функция имеет всего один аргумент, скобки использовать не обязательно:

```
arg1 => {
    // делает что-то.
}
```

В следующем фрагменте кода мы передаем выражения стрелочных функций в виде аргументов в JS Array методы `reduce()` для вычисления суммы и `filter()` для вывода в консоль только четных чисел:

```
const myArray = [1, 2, 3, 4, 5];

console.log( "The sum of myArray elements is " +
                myArray.reduce((a,b) => a+b)); // prints 15

console.log( "The even numbers in myArray are " +
                myArray.filter( value => value % 2 === 0)); // prints 2 4
```

Теперь, когда вы знакомы с синтаксисом стрелочных функций, давайте посмотрим, как они упрощают работу с объектной ссылкой `this`.

В ES5 выяснить, на какой объект ссылается ключевое слово `this`, не всегда легко. Поищите в Google *JavaScript this, that*, и вы найдете множество постов, в которых люди жалуются на `this`, указывающий «не на тот» объект. Ссылка `this` может иметь различные значения, в зависимости от того, как была вызвана функция и используется ли режим `strict` (см. документацию для «Strict Mode» в Mozilla Developer Network по ссылке http://mng.bz/VNVL).

Сначала мы продемонстрируем проблему, а затем покажем ее решение, предлагаемое ES6.

Ознакомьтесь с кодом из следующего листинга, который вызывает анонимную функцию каждую секунду. Эта функция выводит случайные сгенерированные цены для тикера (биржевого символа), передаваемого в функцию-конструктор `StockQuoteGenerator()`.

Листинг А.3. this указывает на разные объекты

```
function StockQuoteGenerator(symbol){
    this.symbol = symbol;  <-- this.symbol является свойством StockQuoteGenerator()
    console.log(`this.symbol=${this.symbol}`);

    setInterval( function () {
        console.log(`The price of ${this.symbol}  <-- Здесь this.symbol не определен
        is ${Math.random()}`);
    }, 1000);
}
const stockQuoteGenerator = new StockQuoteGenerator("IBM");
```

В первом включении `this` указывал на объект функции, и `this.symbol` имел значение `IBM`. Во втором включении из-за `setInterval()` значение `this.symbol` уже `undefined`. Вы увидите такое поведение не только если функция вызывается внутри `setInterval()`, но и если функция вызывается в любом обратном вызове. Внутри обратного вызова при отключенном режиме `strict` `this` будет указывать на глобальный объект, который не совпадает с `this`, определенным функцией-конструктором `StockQouteGenerator()`. Если же режим `strict` включен, объект `this` будет `undefined`.

ПРИМЕЧАНИЕ В предыдущем фрагменте кода мы могли просто использовать `symbol` вместо `this.symbol`. Но нашей целью было показать вам, как переменная `this` указывает на разные объекты. Этот пример на CodePen: http://mng.bz/NeEN.

Другим решением для обеспечения выполнения функции в конкретном объекте `this` будет использование JS-функций `call()`, `apply()` или `bind()`.

ПРИМЕЧАНИЕ Если вы не знакомы с проблемой `this` в JavaScript, ознакомьтесь со статьей Ричарда Бовелла (Richard Bovell) «Understand JavaScript "this" with Clarity, and Master It» в блоге «JavaScript is Sexy» по адресу http://mng.bz/ZQfz.

В следующем листинге показано решение со стрелочной функцией, предлагающей недвусмысленный `this`. Здесь мы просто заменили анонимную функцию, передаваемую в `setInterval()`, на стрелочную.

Листинг А.4. Использование стрелочной функции

```
function StockQuoteGenerator(symbol){
    this.symbol = symbol; // this.symbol is undefined inside getQuote()
    console.log("this.symbol=" + this.symbol);
    setInterval(() =>
        console.log(`The price of ${this.symbol} is ${Math.random()}`)
    , 1000);
}
const stockQuoteGenerator = new StockQuoteGenerator("IBM");
```

В предыдущем примере кода ссылка `this` разрешится правильно. Стрелочная функция, переданная в качестве аргумента в `setInterval()`, использует значение `this` окружающего контекста, поэтому она распознает в качестве значения `this.symbol` — `"IBM"`.

ПРИМЕЧАНИЕ Этот пример на CodePen: http://mng.bz/DNOn.

А.6. ОПЕРАТОР ОСТАТКА (REST)

В ES5 написание функции с переменным числом параметров требовало использования специального объекта `arguments`. Этот объект аналогичен массиву и содержит значения, соответствующие аргументам, переданным в функцию.

Начиная с ES6, для переменного числа аргументов функции вы можете использовать оператор остатка. Он представлен тремя точками (...) и должен выступать в роли последнего аргумента функции. Если имя аргумента начинается с трех точек, то функция получит оставшиеся аргументы в массиве.

Например, вы можете передать в функцию несколько покупателей, используя имя переменной с оператором остатка:

```
function processCustomers(...customers) {
    // Здесь располагается реализация функции.
}
```

Внутри этой функции вы можете обработать данные `customers` аналогично любому массиву.

Представьте, что вам нужно написать функцию для вычисления налогов, которая должна вызываться с первым аргументом `income`, сопровождаемым любым числом аргументов, представляющих имена покупателей. В листинге A.5 показано, как можно обработать переменное число аргументов, используя синтаксис ES5 и ES6. Функция `calcTaxES5()` использует объект `arguments`, а функция `calcTaxES6()` — оператор остатка.

Листинг A.5. Использование оператора остатка

```
// ES5 и объект arguments
    function calcTaxES5() {

        console.log("ES5. Calculating tax for customers with the income ",
                            arguments[0]); // доход является первым элементом.

        //Начиная со второго элемента, извлекает массив
        var customers = [].slice.call(arguments, 1);
        customers.forEach(function (customer) {
            console.log("Processing ", customer);
        });
    }

    calcTaxES5(50000, "Smith", "Johnson", "McDonald");
    calcTaxES5(750000, "Olson", "Clinton");

// ES6 и оператор остатка.
  function calcTaxES6(income, ...customers) {
        console.log(`ES6. Calculating tax for customers with the income
➥ ${income}`);

        customers.forEach( (customer) => console.log(`Processing ${customer}`));
    }

    calcTaxES6(50000, "Smith", "Johnson", "McDonald");
    calcTaxES6(750000, "Olson", "Clinton");
```

Обе функции (`calcTaxES5` и `calcTaxES6`) производят одинаковый результат:

```
ES5. Calculating tax for customers with the income 50000
Processing Smith
Processing Johnson
```

```
Processing McDonald
ES5. Calculating tax for customers with the income 750000
Processing Olson
Processing Clinton
ES6. Calculating tax for customers with the income 50000
Processing Smith
Processing Johnson
Processing McDonald
ES6. Calculating tax for customers with the income 750000
Processing Olson
Processing Clinton
```

ПРИМЕЧАНИЕ Этот пример на CodePen: http://mng.bz/I2zq.

Тем не менее в обработке покупателей разница присутствует. Поскольку объект `arguments` не является реальным массивом, нам пришлось создать в ES5-версии массив, используя методы `slice()` и `call()`, чтобы извлечь имена покупателей, начиная со второго элемента в `arguments`. Версия ES6 не требует использования подобных трюков, потому что оператор остатка предоставляет стандартный массив покупателей. Использование этого оператора сделало код более простым и читаемым.

А.7. ОПЕРАТОР РАСПРОСТРАНЕНИЯ

Оператор распространения из версии ES6 также представлен в виде трех точек (...), но в то время как оператор остатка может преобразовывать переменное число параметров в массив, оператор расширения может, наоборот, преобразовывать массив в список значений или параметров функции.

Предположим, у вас есть два массива и вам нужно добавить элементы второго в конец первого. Используя оператор распространения, это можно сделать в одной строчке кода:

```
let array1= [...array2];
```

Здесь этот оператор извлекает каждый элемент `array2` и добавляет его в новый массив (квадратные скобки означают здесь `создать новый массив`). Вы также можете создавать копии массивов:

```
array1.push(...array2);
```

Находить максимальное значение при помощи оператора распространения тоже очень легко:

```
const maxValue = Math.max(...myArray);
```

В некоторых случаях вам понадобится клонировать объект. Предположим, у вас есть объект, хранящий состояние приложения, и вы хотите создать новый объект, когда одно из свойств состояния изменится. Вам нужно не изменить первичный объект, а именно создать его дубликат с изменением одного или нескольких свойств. Один из способов реализации неизменяемых объектов — это использование функции `Object.assign()`. Код в следующем листинге сперва создает один клон, а затем еще один клон, одновременно изменяя значение `lastName`.

Листинг A.6. Дублирование при помощи assign()

```
// Дублирует при помощи Object.assign().
const myObject = {name: "Mary" , lastName: "Smith"};
const clone = Object.assign({}, myObject);
console.log(clone);

// Дублирует с изменением свойства lastName.
const cloneModified = Object.assign({}, myObject, {lastName: "Lee"});
console.log(cloneModified);
```

При этом, как можно видеть из следующего листинга, оператор распространения для достижения той же цели предлагает более лаконичный синтаксис.

Листинг A.7. Клонирование при помощи распространения

```
// Клонирует при помощи распространения.
const myObject = { name: "Mary" , lastName: "Smith"};
const cloneSpread = {...myObject};
console.log(cloneSpread);

// Клонирует с изменением `lastName`.
const cloneSpreadModified = {...myObject, lastName: "Lee"};
console.log(cloneSpreadModified);
```

Наш объект `myObject` содержит два свойства: `name` и `lastName`. Строка, клонирующая `myObject`, параллельно изменяя `lastName`, будет работать, даже если вы или кто-либо еще добавите в `myObject` больше свойств.

ПРИМЕЧАНИЕ Этот пример на CodePen: http://mng.bz/X2pL.

Клонирование при помощи `Object.assign()` или посредством оператора распространения создает упрощенную копию объекта. Происходит копирование всех значений, имеющихся в объекте на момент клонирования, но если какие-то из свойств объекта сами являются объектами, то на такие вложенные свойства будут скопированы только ссылки. Если после упрощенного клонирования значения вложенных свойств изменятся в оригинальном объекте, в клоне эти изменения также отразятся.

В листинге А.8 показан объект, имеющий вложенный объект — `birth`. Изначально дата рождения определена как `18 Jan 2019`. После клонирования объект-клон будет иметь такую же дату. Но если вы измените ее в оригинальном объекте, клон также получит новое значение. Это доказывает, что была скопирована только ссылка на вложенный объект, но не его значения.

Листинг А.8. Упрощенное клонирование

```
const myObject = { name: 'Mary', lastName: 'Smith', birth: { date:
➥ '18 Jan 2019' }};
const clone = {...myObject};  ◄—— Клонирует myObject
console.log(clone.birth.date);  ◄—— Дата рождения клона 18 Jan 2019
myObject.birth.date = '20 Jan 2019';  ◄—— Изменяет дату рождения оригинального объекта
console.log(clone.birth.date);  ◄—— Дата рождения клона изменена на 20 Jan 2019
```

А.8. ДЕСТРУКТУРИЗАЦИЯ

Создание экземпляров означает конструирование их в памяти. Термин *деструктуризация* означает изменение структуры или разделение объектов. В ES5 вы могли деструктурировать любой объект или коллекцию, написав для этого функцию. В ES6 был добавлен синтаксис деструктурирующего присваивания, позволяющий извлекать данные из свойств объекта или массива в простом выражении посредством указания *сопоставления с образцом* (matching pattern). Будет легче объяснить это на примере, что мы далее и сделаем.

А.8.1. Деструктуризация объектов

Предположим, что функция `getStock()` возвращает объект `Stock`, имеющий атрибуты `symbol` и `price`. В ES5 если вы хотели присвоить значения этих атрибутов отдельным переменным, вам сначала нужно было создать переменную для хранения объекта `Stock`, а затем написать две инструкции, присваивающие атрибуты объекта соответствующим переменным:

```
var stock = getStock();
var symbol = stock.symbol;
var price = stock.price;
```

Начиная с версии ES6, вам просто нужно написать шаблон для сопоставления слева и присвоить ему объект `Stock`:

```
let {symbol, price} = getStock();
```

Несколько непривычно видеть фигурные скобки слева от знака равенства, но такова часть синтаксиса сопоставляющего выражения. Когда вы видите

фигурные скобки слева, представляйте их как блок кода, а не объектный литерал.

В следующем листинге продемонстрировано получение от функции `getStock()` объекта `Stock` и его деструктуризация на две переменные.

Листинг А.9. Деструктуризация объекта

```
function getStock() {

    return {
        symbol: "IBM",
        price: 100.00
    };
}

let {symbol, price} = getStock();

console.log(`The price of ${symbol} is ${price}`);
```

Выполнение этого сценария выведет следующее:

```
The price of IBM is 100
```

Иначе говоря, мы присваиваем набор данных (в данном случае свойств объекта) к набору переменных (`symbol` и `price`) в одном выражении присваивания. Даже если бы объект `Stock` имел больше свойств, предыдущее деструктурирующее выражение все равно бы работало, потому что `symbol` и `price` совпадали бы с шаблоном. Выражение сопоставления перечисляет только переменные для интересующих вас атрибутов объекта.

ПРИМЕЧАНИЕ Этот пример на CodePen: http://mng.bz/CI47.

Вы также можете деструктурировать вложенные объекты. Код в следующем листинге создает вложенный объект, представляющий фонд акций Microsoft, и передает его в функцию `printStockInfo`, которая берет из этого объекта тикер и название биржи.

Листинг А.10. Деструктуризация вложенного объекта

```
const msft = {
    symbol: "MSFT",
    lastPrice: 50.00,
    exchange: {   <-- Вложенный объект
        name: "NASDAQ",
        tradingHours: "9:30am-4pm"
    }
};
```

```
function printStockInfo(stock) {
    let {symbol, exchange: {name}} = stock;
    console.log(`The ${symbol} stock is traded at ${name}`);
}

printStockInfo(msft);
```

Деструктурирует вложенный объект для получения названия биржи

Выполнение предыдущего сценария выведет в консоль следующее:

```
The MSFT stock is traded at NASDAQ
```

ПРИМЕЧАНИЕ Этот пример на CodePen: http://mng.bz/Xauq.

Предположим, вы пишете функцию для обработки события DOM браузера. В HTML-части вы вызываете эту функцию, передавая объект события в качестве аргумента. Этот объект имеет несколько свойств, но вашей функции-обработчику для идентификации объекта, отправившего это событие, нужно лишь свойство `target`. Синтаксис деструктуризации облегчает эту задачу:

```
<button id="myButton">Click me</button>
...
document
    .getElementById("myButton")
    .addEventListener("click", ({target}) =>
                              console.log(target));
```

Обратите внимание на синтаксис деструктуризации `{target}` в аргументе функции.

ПРИМЕЧАНИЕ Этот пример на CodePen: http://mng.bz/Dj24.

Начиная с версии ES2018, в процессе деструктуризации вы можете использовать синтаксис, аналогичный операторам остатка и распространения. Например, следующий код присвоит переменной `lastPrice` значение `50`, а оставшиеся свойства объекта `msft` будут помещены в объект `otherInfo`.

ПРИМЕЧАНИЕ Этот пример на CodePen: http://mng.bz/loN6.

Листинг А.11. Совмещение деструктуризации и оператора остатка

```
const msft = {
    symbol: "MSFT",
    lastPrice: 50.00,
    exchange: {
        name: "NASDAQ",
        tradingHours: "9:30am-4pm"
    }
};
```

```
const { lastPrice, ...otherInfo } = msft;
```

Деструктуризация и оператор остатка

```
console.log(`lastPrice= ${lastPrice}`);
console.log(`otherInfo=`, otherInfo);
```

A.8.2. Деструктуризация массивов

Деструктуризация массивов работает практически так же, как и деструктуризация объектов, но вместо фигурных скобок вы используете квадратные. Помимо этого, если при деструктуризации объектов вам нужно указывать переменные, соответствующие свойствам объекта, то в случае с массивами вы указываете переменные, соответствующие индексам массива.

Следующий код извлекает значения двух элементов массива в две переменные:

```
let [name1, name2] = ["Smith", "Clinton"];
console.log(`name1 = ${name1}, name2 = ${name2}`);
```

Вывод будет следующим:

```
name1 = Smith, name2 = Clinton
```

Если бы вы хотели извлечь только второй элемент массива, сопоставление выглядело бы так:

```
let [, name2] = ["Smith", "Clinton"];
```

Если функция возвращает массив, синтаксис деструктуризации преобразует его в функцию, возвращающую несколько значений, как показано в функции `getCustomers()` ниже:

```
function getCustomers() {
    return ["Smith", , , "Gonzales"];
}

let [firstCustomer, , , lastCustomer] = getCustomers();
console.log(`The first customer is ${firstCustomer} and the last one is
➥ ${lastCustomer}`);
```

Теперь совместим деструктуризацию массива с оператором остатка. Предположим, есть массив нескольких покупателей, но нам нужно обработать только первых двух. В следующем фрагменте кода показано, как это сделать:

```
let customers = ["Smith", "Clinton", "Lou", "Gonzales"];

let [firstCust, secondCust, ...otherCust] = customers;

console.log(`The first customer is ${firstCust} and the second one is
➥ ${secondCust}`);
console.log(`Other customers are ${otherCust}`);
```

А вот итоговый вывод консоли:

```
The first customer is Smith and the second one is Clinton
Other customers are Lou, Gonzales
```

Аналогичным образом вы можете передать шаблон для сопоставления с параметром остатка в функцию:

```
var customers = ["Smith", "Clinton", "Lou", "Gonzales"];

function processFirstTwoCustomers([firstCust, secondCust, ...otherCust]) {

  console.log(`The first customer is ${firstCust} and the second one is
➥ ${secondCust}`);
  console.log(`Other customers are ${otherCust}`);

}

processFirstTwoCustomers(customers);
```

Вывод будет таким же:

```
The first customer is Smith and the second one is Clinton
Other customers are Lou,Gonzales
```

Подытоживая, можно сказать, что преимущество деструктуризации — в возможности писать меньше кода при необходимости инициализации переменной с данными, расположенными в свойствах объекта или массивах.

А.9. КЛАССЫ И НАСЛЕДОВАНИЕ

Несмотря на наличие в ES5 поддержки ООП и наследования, классы в ES6 облегчают чтение и написание кода.

В ES5 объекты можно создать либо с нуля, либо наследованием от других объектов. По умолчанию объекты в JS наследуют от `Object`. Такое объектное наследование, а в данном случае именно *наследование через прототипы*, реализуется через специальное свойство `prototype`, которое указывает на предка объекта. В ES5 для создания объекта `NJTax`, наследующего от объекта `Tax`, вы можете написать следующее:

```
function Tax() {
    // Здесь размещается код налогового объекта.
}

function NJTax() {
```

```
    // Здесь размещается код налогового объекта Нью-Джерси.
}

NJTax.prototype = new Tax();  ◄—— Наследует NJTax от Tax

var njTax = new NJTax();
```

В ES6 для приведения синтаксиса к соответствию с другими ООП языками вроде Java и C#, были введены ключевые слова `class` и `extends`. Далее показан ES6-эквивалент предыдущего кода:

```
class Tax {
    // Здесь размещается код налога.
}

class NJTax extends Tax {
    // Здесь размещается код налогового объекта Нью-Джерси.
}

let njTax = new NJTax();
```

Класс `Tax` является предком, или суперклассом, а `NJTax` является потомком, или подклассом. Вы также можете сказать, что класс `NJTax` имеет отношения формата «является» с классом `Tax`: `NJTax` является `Tax`. Вы можете реализовать в `NJTax` дополнительную функциональность, но он по-прежнему будет «являться» или «быть вариацией» `Tax`.

Аналогично, если вы создадите класс `Employee`, наследующий от `Person`, то можете сказать, что `Employee` является `Person`.

Можно создать один или несколько экземпляров объектов:

```
var tax1 = new Tax();  ◄—— Первый экземпляр объекта Tax
var tax2 = new Tax();  ◄—— Второй экземпляр объекта Tax
```

> **ПРИМЕЧАНИЕ** В противоположность объявлениям функций, объявления классов не поднимаются. То есть вы должны объявлять класс до того, как его используете, иначе получите `ReferenceError`.

Каждый из этих объектов будет содержать свойства и методы, существующие в классе `Tax`, но они будут иметь другое *состояние*. Например, первый экземпляр может быть создан для гражданина с годовым доходом $50 000, а второй — для гражданина, заработавшего $75 000. Каждый экземпляр будет использовать одну и ту же копию методов, объявленных в классе `Tax`, поэтому код дублироваться не будет.

В ES5 вы также можете избегать дублирования кода, объявляя методы не внутри объектов, а в их прототипах:

```
function Tax() {
    // Здесь размещается код налогового объекта.
}

Tax.prototype = {
    calcTax: function() {
        // Здесь размещается код для вычисления налога.
    }
}
```

JavaScript по-прежнему остается языком с наследованием через прототипы, но ES6 позволяет писать код более изящно:

```
class Tax() {

    calcTax() {
        // Здесь размещается код для вычисления налога.
    }
}
```

ПЕРЕМЕННЫЕ ЧЛЕНОВ КЛАССА НЕ ПОДДЕРЖИВАЮТСЯ

На момент написания книги JS не позволял объявлять переменные членов классов (они же поля классов или свойства), как это можно делать в Java, C# или TypeScript.

В данный момент поля классов находятся на стадии 3 предложения ECMAScript и уже поддерживаются в Chrome v76 и Babel. Вы можете посмотреть, как объявлять переменные членов в классах TypeScript в разделе 2.2.2.

A.9.1. Конструкторы

В процессе инстанцирования классы выполняют код, расположенный в специальных методах, называемых *конструкторами*. В языках вроде Java и C# имя конструктора должно совпадать с именем класса, но в JS для определения конструктора класса используется ключевое слово `constructor`:

```
class Tax {

    constructor(income) {
        this.income = income;
    }
}

const myTax = new Tax(50000);
```

Конструктор — это особый метод, выполняемый только один раз — при создании объекта. Класс `Tax` не объявляет отдельную классовую переменную `income`,

но создает ее динамически в объекте `this`, инициализируя `this.income` со значениями аргумента конструктора. Переменная `this` указывает на экземпляр текущего объекта.

В следующем примере показано, как создать экземпляр подкласса `NJTax`, предоставив его конструктору доход в размере 50 000.

```
class Tax {
    constructor(income) {
        this.income = income;
    }
}

class NJTax extends Tax {
    // Здесь размещается код, относящийся к налогу в Нью-Джерси.
}

const njTax = new NJTax(50000);

console.log(`The income in njTax instance is ${njTax.income}`);
```

Вывод этого фрагмента кода:

```
The income in njTax instance is 50000
```

Так как подкласс `NJTax` не определяет свой конструктор, то в процессе его инстанцирования конструктор будет вызван из суперкласса `Tax`. Этого не происходит в случае, когда подкласс определяет собственный конструктор. Такой пример вы увидите в следующем разделе.

> **ПРИМЕЧАНИЕ** Пока JS не начнет поддерживать поля классов, для добавления новой переменной члена в подклассе вам нужно объявлять в нем конструктор. В предстоящем релизе ECMAScript поля классов уже должны поддерживаться, и вам не потребуется объявлять конструктор подкласса только ради этого.

Классы в JS являются просто синтаксическим сахаром, повышающим читаемость. Изнутри JS по-прежнему использует наследование через прототипы, которое позволят вам динамически заменять предка во время выполнения, учитывая, что класс может иметь только одного прямого предка.

А.9.2. Ключевое слово super и функция super()

Функция `super()` позволяет подклассу (потомку) вызывать конструктор из суперкласса (предка). Ключевое же слово `super` используется для вызова метода, определенного в суперклассе.

Следующий листинг иллюстрирует как функцию `super()`, так и ключевое слово `super`. Класс `Tax` имеет метод `calculateFederalTax()`, и его подкласс `NJTax` добавляет метод `calculateStateTax()`. Оба этих класса имеют собственные версии метода `calcMinTax()`.

Листинг А.12. Использование super() и super

```
class Tax {
    constructor(income) {
        this.income = income;
    }

    calculateFederalTax() {
        console.log(`Calculating federal tax for income ${this.income}`);
    }

    calcMinTax() {
        console.log("In Tax. Calculating min tax");
        return 123;
    }
}

class NJTax extends Tax {
    constructor(income, stateTaxPercent) {
        super(income);
        this.stateTaxPercent=stateTaxPercent;
    }

    calculateStateTax() {

        console.log(`Calculating state tax for income ${this.income}`);
    }

    calcMinTax() {
        let minTax = super.calcMinTax();
        console.log(`In NJTax. Will adjust min tax of ${minTax}`);
    }
}

const theTax = new NJTax(50000, 6);

theTax.calculateFederalTax();
theTax.calculateStateTax();

theTax.calcMinTax();
```

Выполнение этого кода произведет следующий вывод:

```
Calculating federal tax for income 50000
Calculating state tax for income 50000
In Tax. Calculating min tax
In NJTax. Will adjust min tax of 123
```

ПРИМЕЧАНИЕ Этот пример на CodePen: http://mng.bz/6e9S.

Класс `NJTax` имеет собственный явно определенный конструктор с двумя аргументами: `income` и `stateTaxPercent`, которые вы передаете в процессе инстанцирования `NJTax`. Чтобы убедиться, что конструктор `Tax` вызван (он устанавливает в объекте атрибут `income`), вы явно вызываете его из конструктора подкласса: `super(income)`. В отсутствие этой строки предыдущий сценарий выдал бы ошибку. Вы должны активировать конструктор суперкласса из производного конструктора вызовом функции `super()`.

Другой способ вызова кода суперкласса — это использование ключевого слова `super`. И `Tax`, и `NJTax` имеют методы `calcMinTax()`. Расположенный в `Tax` вычисляет минимальную базовую сумму согласно федеральным налоговым законам, в то время как этот же метод в подклассе использует базовое значение, внося в него коррективы. Оба метода имеют одинаковую сигнатуру, то есть здесь вы видите случай *переопределения метода*.

Вызывая `super.calcMinTax()`, вы гарантируете, что при вычислении налога штата учитывается базовый федеральный налог. Если же вы не вызовете `super.calcMinTax()`, будет применен метод `calcMinTax()` подкласса. Переопределение методов часто используется для замены функциональности метода в суперклассе без внесения изменений в код.

А.9.3. Статические члены класса

Если вам нужно свойство класса, совместно используемое несколькими его экземплярами, нужно объявить его при помощи ключевого слова `static`. Такое свойство будет создано не в каком-то конкретном экземпляре, но в самом классе.

В листинге А.13 вы можете обратиться к статической переменной `counter` из обоих экземпляров объекта `A`, вызвав метод `printCounter()`. Но если вы попытаетесь обратиться к `counter` напрямую при помощи ссылки на экземпляр объекта (например, `a1.counter`), ее тип будет `undefined`.

Листинг А.13. Совместное использование свойства класса

```
class A {
    static counter = 0;  <-- Объявляет статическое свойство

    printCounter(){
        console.log("static counter=" + A.counter);  <-- Обращается к статическому свойству по имени класса
    };
}

const a1 = new A();  <-- Создает первый экземпляр класса A
```

```
A.counter++;  ◄── Увеличивает статический счетчик
a1.printCounter(); // выводит 1

A.counter++;  ◄── Увеличивает статический счетчик

const a2 = new A();  ◄── Создает второй экземпляр класса A
a2.printCounter(); // выводит 2

console.log("On the a1 instance, counter is " + a1.counter);
console.log("On the a2 instance, counter is " + a2.counter);
```

В этом примере кода мы увеличиваем счетчик вне экземпляра класса, используя в качестве ссылок имена класса: `A.counter`. Оба экземпляра класса `A` видят одинаковое значение счетчика. Обратите внимание, что даже если мы вызываем метод `printCounter()` в конкретном экземпляре, он обращается к статическому свойству, используя имя класса.

Этот код производит следующий вывод:

```
static counter=1
static counter=2
On the a1 instance, counter is undefined
On the a2 instance, counter is undefined
```

В двух последних строках этого вывода мы пытаемся обратиться к свойству `counter`, используя ссылки экземпляра `a1` и `a2`, но ни в одном из экземпляров нет такого свойства, следовательно, их тип `undefined`.

ПРИМЕЧАНИЕ Этот пример на CodePen: http://mng.bz/BYQ0.

С помощью ключевого слова `static` вы также можете создавать статические методы. Такие методы тоже вызываются не в экземпляре класса, но в самом классе. Мы часто используем их в классах, выступающих в роли коллекции служебных функций, и создание экземпляров не требуется.

Листинг А.14. Класс Helper является коллекцией служебных функций

```
class Helper {

static convertDollarsToEuros() {  ◄── Объявляет первый статический метод

        console.log("Converting dollars to euros");
    }

    static convertCelsiusToFahrenheit() {  ◄── Объявляет второй статический метод

        console.log("Converting Celsius to Fahrenheit");
    }
```

```
}

Helper.convertDollarsToEuros();
Helper.convertCelsiusToFahrenheit();
```

Вызывает статический метод, не инстанцируя класс

ПРИМЕЧАНИЕ Этот пример на CodePen: http://mng.bz/dxaN.

В листинге 2.2, где мы реализовывали шаблон Одиночка, вы можете увидеть практическое применение статических членов класса.

А.10. АСИНХРОННАЯ ОБРАБОТКА

Для организации асинхронной обработки в ES5 вам требовалось использовать *обратные вызовы* — функции, передаваемые в качестве аргументов в другие функции для дальнейшего их вызова. Обратные вызовы можно активировать синхронно либо асинхронно.

Например, вы можете передать обратный вызов в метод массива `forEach()` для синхронной активации. При создании AJAX-запросов к серверу вы можете передавать функцию обратного вызова для асинхронной активации при получении результата с сервера.

А.10.1. Ад обратных вызовов

Рассмотрим пример получения с сервера данных о заказанных товарах. Начинается он с асинхронного вызова сервера для получения информации о покупателях, а затем для каждого покупателя вам нужно сделать еще один вызов, чтобы получить их заказы. Далее для каждого заказа нужно получить товары, и заключительный вызов будет получать описание этих товаров. В асинхронной обработке вы не знаете, когда завершится каждая из этих операций, поэтому нужно написать функции обратных вызовов, которые будут активироваться при завершении каждой предыдущей.

Используем функцию `setTimeout()` для эмуляции задержек, представив, что каждой операции для завершения требуется одна секунда. На рис. А.1 показан пример такого кода.

ПРИМЕЧАНИЕ Использование обратных вызовов считается антипаттерном, также известным как пирамида гибели, или ад обратных вызовов (рис. А.1 слева). В коде на этом изображении представлены четыре обратных вызова, при этом подобный уровень вкладывания существенно усложняет чтение. В реальных приложениях пирамида может расти очень быстро, существенно усложняя чтение и отладку кода.

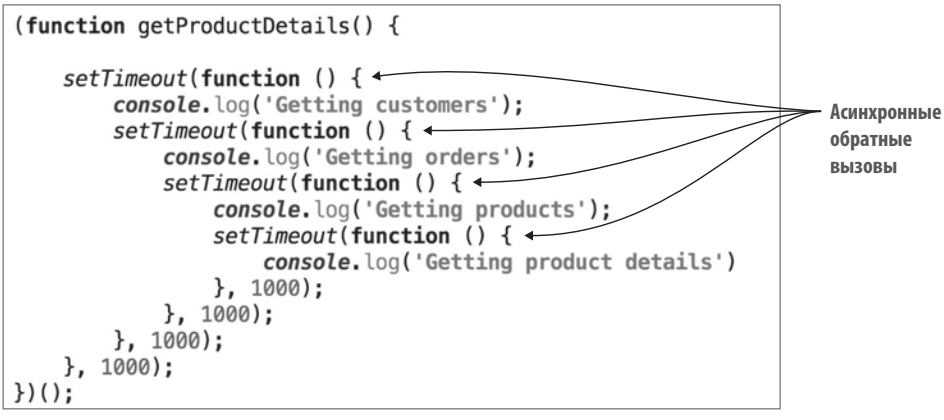


Рис. А.1. Ад обратных вызовов, или пирамида гибели

Выполнение кода на рис. А.1 выведет в консоль следующие сообщения с задержкой в одну секунду:

```
Getting customers
Getting orders
Getting products
Getting product details
```

ПРИМЕЧАНИЕ Этот пример на CodePen: http://mng.bz/DAX5.

А.10.2. Промисы

Когда вы нажимаете кнопку на кофемашине, то не получаете чашку кофе в ту же секунду. Вы получаете обещание (промис), что чашка кофе будет предоставлена вам чуть позже. Если вы не забудете предоставить воду и молотый кофе, этот промис *разрешится* и вы сможете насладиться кофе через примерно минуту. Если же в кофемашине закончились вода или кофе, промис будет *отвергнут*. Весь этот процесс асинхронен, и пока ваш кофе варится, вы можете заниматься другими делами.

Промисы в JS позволяют вам избегать вложенных вызовов и делать асинхронный код более читаемым. Объект `Promise` представляет конечное завершение или провал асинхронной операции. После своего создания объект `Promise` ожидает и прослушивает результат асинхронной операции, сообщая вам о ее успехе либо в провале, чтобы вы могли перейти к последующим шагам соответственно.

Объект `Promise` представляет будущий результат операции и может находиться в одном из трех состояний:

- Fulfilled (*выполнен*) — операция успешно завершилась.
- Rejected (*отвергнут*) — операция провалилась и вернула ошибку.
- Pending (*ожидание*) — операция еще выполняется.

Вы можете инстанцировать объект `Promise`, передав в его конструктор две функции: одну для вызова при успешном завершении операции и вторую для вызова в случае провала. Рассмотрите сценарий с функцией `getCustomers()`, показанный в следующем листинге.

Листинг А.15. Использование промиса

```
function getCustomers() {

    return new Promise(
        function (resolve, reject) {

        console.log("Getting customers");
            // Здесь эмулируется асинхронный вызов сервера
        setTimeout(function() {
            const success = true;
            if (success) {
              resolve("John Smith");  <-- Получение покупателя
            } else {
              reject("Can't get customers"); <-- Вызывается при возникновении ошибки
            }
         }, 1000);
      }
   );
}

getCustomers()
    .then((cust) => console.log(cust))  <-- Вызывается при выполнении промиса
    .catch((err) => console.log(err));  <-- Вызывается, если промис отвергнут
console.log("Invoked getCustomers. Waiting for results");
```

Функция `getCustomers()` возвращает объект `Promise`, который инстанцируется с функцией, имеющей аргументы конструктора `resolve` и `reject`. В коде вы активируете `resolve()`, если получаете информацию о покупателе. Для простоты `setTimeout()` эмулирует асинхронный вызов продолжительностью в одну секунду. Кроме этого, мы жестко закодировали флаг `success` со значением `true`. В реальном сценарии вы могли сделать запрос с объектом `XMLHttpRequest` и вызвать `resolve()` при успешном извлечении результата либо `reject()` в случае ошибки.

В нижней части предыдущего листинга мы прикрепили методы `then()` и `catch()` к экземпляру `Promise()`. Вызван будет только один из них. Когда вы вызываете `resolve("John Smith")` изнутри функции, это приводит к вызову метода `then()`, получившего `"John Smith"` в качестве аргумента. Если вы измените значение

success на false, будет вызван метод `catch()` с аргументом, содержащим `"Can't get customers"`:

```
Getting customers
Invoked getCustomers. Waiting for results
John Smith
```

Обратите внимание на сообщение `Invoked getCustomers. Waiting for Results`, выведенное перед `John Smith`. Это доказывает, что функция `getCustomers()` сработала асинхронно.

ПРИМЕЧАНИЕ Этот пример на CodePen: http://mng.bz/5rf3.

Каждый промис представляет одну асинхронную операцию, и вы можете соединять их в цепочку, гарантируя определенный порядок выполнения. Давайте добавим в следующем листинге функцию `getOrders()`, которая может находить заказы для переданного покупателя, и соединим ее с `getCustomers()`.

Листинг А.16. Объединение промисов в цепочки

```
function getCustomers() {

    return new Promise(
        function (resolve, reject) {

        console.log("Getting customers");
            // Здесь эмулируется асинхронный вызов сервера.
        setTimeout(function() {
            const success = true;
            if (success){
                resolve("John Smith");
            }else{
                reject("Can't get customers");
            }
        }, 1000);
      }
  );
}

function getOrders(customer) {

    return new Promise(
        function (resolve, reject) {

            // Здесь эмулируется асинхронный вызов сервера.
        setTimeout(function() {
            const success = true;
            if (success) {
              resolve(`Found the order 123 for ${customer}`);
            } else {
              reject("Can't get orders");
            }
```

Вызывается при успешном получении покупателя

Вызывается при успешном нахождении заказа покупателя

```
        }, 1000);
      }
  );
}
getCustomers()
    .then((cust) => {
                console.log(cust);
                return cust;
            })
    .then((cust) => getOrders(cust))  <-- Соединяет с getOrders()
    .then((order) => console.log(order))
    .catch((err) => console.error(err));  <-- Обрабатывает ошибки
console.log("Chained getCustomers and getOrders. Waiting for results");
```

Этот код не только объявляет и соединяет две функции в цепочку, но также демонстрирует, как вы можете вывести промежуточные результаты в консоль. Вывод предыдущего листинга приведен ниже (обратите внимание, что покупатель, возвращенный `getCustomers()`, был правильно передан в `getOrders()`):

```
Getting customers
Chained getCustomers and getOrders. Waiting for results
John Smith
Found the order 123 for John Smith
```

ПРИМЕЧАНИЕ Этот пример на CodePen: http://mng.bz/6z5k.

Вы можете соединять вызовы функций в цепочки, используя `then()`, и иметь всего один скрипт обработки ошибок для всех вызовов цепочки. Если произойдет ошибка, она распространится по всей цепочке `then`, пока не встретит обработчик ошибок. После ошибки дальнейших вызовов `then` не последует.

Изменение значения переменной `success` на `false` в предыдущем листинге приведет к выводу сообщения «Can't get customers», и метод `getOrders()` вызван не будет. Если удалить эти выводы консоли, код, извлекающий покупателей и заказы, будет выглядеть чистым и понятным:

```
getCustomers()
    .then((cust) => getOrders(cust))
    .catch((err) => console.error(err));
```

Добавление дополнительных then не усложнит читаемость этого кода (сравните его с пирамидой гибели из рис. А.1).

А.10.3. Разрешение нескольких промисов одновременно

Еще один случай, который стоит рассмотреть, — это асинхронные функции, не зависящие друг от друга. Предположим, вам нужно вызвать две функции

в определенном порядке, но при этом также нужно выполнить некоторое действие только после их завершения. Объект `Promise` имеет метод `all()`, получающий итерируемую коллекцию промисов и выполняющий (разрешающий) их все. Поскольку метод `all()` возвращает объект `Promise`, вы можете добавить в результат `then()` или `catch()` (или оба).

Представьте себе веб-портал, которому нужно сделать несколько асинхронных вызовов для получения данных о погоде, новостей с фондовой биржи и данных о дорожной обстановке. Если вы хотите отразить страницу портала только после завершения всех этих вызовов, нужно использовать `Promise.all()`:

```
Promise.all([getWeather(),
             getStockMarketNews(),
             getTraffic()])
.then( (results) => { /* render the portal's UI here */ })
.catch(err => console.error(err)) ;
```

Имейте в виду, что `Promise.all()` разрешается только после разрешения всех промисов. Если один из них будет отвергнут, управление перейдет к обработчику `catch()`.

В сравнении с функциями обратных вызовов промисы делают код более линейным и читаемым, представляя при этом несколько состояний приложения. С другой стороны, промисы нельзя отменить. Представьте нетерпеливого пользователя, который для получения данных с сервера щелкает по кнопке несколько раз. Каждый щелчок создает промис и инициирует HTTP-запрос. В этом случае нет способа сохранить только последний запрос и отменить незавершенные.

JS-код с промисами легче читать, но если вы внимательно посмотрите на функцию `then()`, то увидите, что вам по-прежнему нужно передать функцию обратного вызова, которая будет вызвана позже. Здесь стоит перейти к знакомству с ключевыми словами `async` и `await`, которые являются следующим шагом в эволюции синтаксиса JS для асинхронного программирования.

A.10.4. async-await

Ключевые слова `async/await` были добавлены в версии ES8 (она же ES2017). Они позволяют вам рассматривать возвращающие промисы функции как синхронные. Следующая строка кода выполняется, только когда завершается предыдущая, но ожидание завершения асинхронного кода происходит на заднем плане и не блокирует выполнение других частей программы:

- `async` — ключевое слово, отмечающее асинхронные функции.
- `await` — ключевое слово, помещаемое перед вызовом `async` функции. Это дает команду движку JavaScript не переходить к следующей строке, пока асин-

хронная функция не вернет результат или не выбросит ошибку. Движок JS внутренне обернет выражение справа от `await` в промис, а оставшуюся часть метода — в обратный вызов `then()`.

Чтобы продемонстрировать применение `async/await`, следующий листинг повторно использует вызовы `getCustomers()` и `getOrders()`, использующие промисы для эмуляции асинхронного выполнения.

Листинг А.17. Объявление двух функций, использующих промисы

```
function getCustomers() {

    return new Promise(
        function (resolve, reject) {

            console.log("Getting customers");
            // Эмулирует асинхронный вызов продолжительностью в одну секунду.
            setTimeout(function() {
                const success = true; if (success){
                    resolve("John Smith");
                } else {
                    reject("Can't get customers");
                }
            }, 1000);
        }
    );
}

function getOrders(customer) {

    return new Promise(
        function (resolve, reject) {

            // Эмулирует асинхронный вызов продолжительностью в одну секунду.
            setTimeout(function() {
                const success = true; // change it to false

                if (success){
                    resolve( `Found the order 123 for ${customer}`);
                } else {
                    reject(`getOrders() has thrown an error for ${customer}`);
                }
            }, 1000);
        }
    );
}
```

Нужно соединить в цепочку эти вызовы функций, но на этот раз мы не будем использовать вызовы `then()`, как мы делали с промисами. Мы создадим новую функцию `getCustomerOrders()`, которая внутренне вызывает `getCustomers()`, а по завершении — `getOrders()`.

Будем использовать ключевое слово `await` в строках, где вызываем `getCustomers()` и `getOrders()`, чтобы код ожидал завершения этих функций, прежде чем

продолжать выполнение. Отметим функцию `getCustomerOrders()` ключевым словом `async`, потому что `await` она будет использовать внутри. В следующем листинге объявляется и вызывается функция `getCustomerOrders()`:

Листинг А.18. Объявление и вызов async функции

```
(async function getCustomersOrders() {
    try {
        const customer = await getCustomers();
        console.log(`Got customer ${customer}`);
        const orders = await getOrders(customer);
        console.log(orders);
    } catch(err) {
        console.log(err);
    }
})();

console.log("This is the last line in the app. Chained getCustomers() and
➥ getOrders() are still running without blocking the rest of the app.");
```

Объявляет функцию с ключевым словом async

Вызывает асинхронную функцию getCustomers() с await, чтобы код ниже не выполнялся до ее завершения

Вызывает асинхронную функцию getOrders() с await, чтобы код ниже не выполнялся до ее завершения

Обрабатывает ошибки

Этот код выполняется вне асинхронной функции

Здесь можно заметить, что этот код выглядит как синхронный. У него нет обратных вызовов, и выполняется он строка за строкой. Обработка ошибок выполняется стандартным способом при помощи блока `try/catch`.

Выполнение этого кода произведет следующий вывод:

```
Getting customers
This is the last line in the app. Chained getCustomers() and getOrders()
are still running without blocking the rest of the app.
Got customer John Smith
Found the order 123 for John Smith
```

Обратите внимание, что сообщение о последней строке кода выводится перед именем покупателя и номером заказа. Несмотря на то что эти значения извлекаются асинхронно несколько позже, выполнение этого небольшого приложения не остановилось и сценарий достиг последней строки до того, как асинхронные функции `getCustomers()` и `getOrders()` закончили свое выполнение.

ПРИМЕЧАНИЕ Пример с `async/await` на CodePen: http://mng.bz/pSV8.

А.11. МОДУЛИ

В любом языке программирования разделение кода помогает организовать приложение в логические и, возможно, повторно используемые модули. Разбитые

на модули приложения позволяют более эффективно разделять задачи программирования между разработчиками. Создатели модулей решают, какие API должны быть раскрыты модулем для внешнего использования, а какие должны использоваться внутренне.

В ES5 нет языковых конструкций для создания модулей, поэтому нам нужно прибегнуть к одному из двух вариантов:

- Вручную реализовать шаблон проектирования модуля как немедленно инициализируемой функции.
- Использовать сторонние реализации модулей вроде стандарта асинхронного определения модуля (AMD; http://mng.bz/JKVc) или CommonJS (http://mng.bz/7Lld).

CommonJS был создан для модульных JavaScript-приложений, выполняемых вне браузера (вроде написанных на Node.js и развернутых под движком Google V8). AMD в первую очередь используется для приложений, выполняемых в браузере.

Вам стоит разделить ваше приложение на модули, чтобы облегчить обслуживание кода. Кроме того, следует минимизировать количество JS-кода, загружаемого в клиент при запуске. Представьте типичный онлайн-магазин. Нужно ли вам загружать код для обработки платежей, когда пользователь открывает главную страницу приложения? Что, если он в итоге так и не нажмет на кнопку Сформировать заказ? Будет правильным разбить такое приложение на модули, чтобы код загружался в подходящее время. RequireJS, вероятно, наиболее популярная сторонняя библиотека, реализующая стандарт AMD. Она позволяет вам определять зависимости между модулями и загружать их в браузер по требованию.

Начиная с версии ES6, модули стали частью языка. Если сценарий использует ключевые слова `import` или `export`, то он становится модулем. Например, следующий сценарий shipping.js экспортирует функцию `ship()`, которая в итоге может быть импортирована другими сценариями. Функция же `calculateShippingCost()` остается невидимой для внешних сценариев.

Листинг А.19. Модуль shipping.js экспортирует только член ship()

```
export function ship() {
    console.log("Shipping products...");
}

function calculateShippingCost(){
    console.log("Calculating shipping cost");
}
```

Следующий сценарий main.js импортирует и использует функцию `ship()` из shipping.js:

Листинг А.20. Импорт члена ship из модуля main.js

```
import {ship} from './shipping.js';

ship();
```

С точки зрения синтаксиса выглядит он чистым и простым. Тем не менее в ES6 наличие инструкции `import` не производит загрузку модуля. Загрузка модулей в ней не стандартизована, и разработчики использовали сторонние загрузчики вроде SystemJS или Webpack (подробности в главе 6).

JAVASCRIPT-МОДУЛИ И ГЛОБАЛЬНАЯ ОБЛАСТЬ ВИДИМОСТИ

Предположим, у вас есть проект со множеством файлов, и один из них содержит следующее:

```
class Person {}
```

Так как мы ничего не экспортировали из этого файла, это не модуль ES6, и экземпляр класса `Person` будет создан в глобальной области. Если у вас уже есть другой сценарий в этом же проекте, который также объявляет класс `Person`, компилятор TS укажет на ошибку в предыдущем коде, утверждая, что вы пытаетесь повторно объявить уже существующий элемент.

Добавление инструкции `export` в предыдущий код изменит ситуацию, и этот сценарий станет модулем:

```
export class Person {}
```

Теперь объекты типа `Person` не будут создаваться в глобальной области. Их область будет ограничена только теми сценариями (другими модулями ES6), которые импортируют `Person`.

Модули ES6 позволяют избегать загрязнения глобальной области и ограничивают видимость сценариев и их членов (классов, функций, переменных и констант) импортирующими их модулями.

Теперь можно указать в качестве типа сценария `module`, как показано в следующем листинге. Все современные браузеры поддерживают `module` как допустимый тип в теге `<script>`, значит, вы можете дать команду браузеру загрузить сценарий как модуль ES6.

Листинг A.21. index.html: использование сценария типа module

```
<!DOCTYPE html>
<head>
    <title>My modules</title>
</head>
<body>
    <h1>Hello modules!</h1>
    <script type="module" src="./main.js"></script>  ◄—— Загрузка первого модуля
</body>
</html>
```

Обратите внимание: несмотря на то что мы не упоминали сценарий, расположенный в файле shipping.js, он все равно будет загружен, потому что сценарий из main.js его импортирует. Для более старых браузеров вы можете использовать атрибут `nomodule` и предоставить резервный сценарий:

```
<script type="module" src="./main.js"></script>
<script nomodule src="./main_fallback.js"></script>
```

Если браузер поддерживает тип `module`, он проигнорирует строку с `nomodule`.

> **ПРИМЕЧАНИЕ** В листинге 9.4 вы видите HTML-файл, использующий тег `<script>` с атрибутом `type="module"`.

A.11.1. Импорты и экспорты

Модуль — это просто JS-файл, реализующий конкретную функциональность и экспортирующий (либо импортирующий) публичный API, чтобы другие JS-программы могли его использовать. Нет никакого специального ключевого слова, объявляющего, что код в конкретном файле является модулем. Вы преобразуете сценарий в модуль ES6, просто используя `import` и `export`.

Ключевое слово `import` позволяет одному сценарию объявить, что ему нужно использовать экспортированные члены другого сценария. Аналогичным образом ключевое слово `export` позволяет объявлять переменные, функции или классы, которые модуль должен предоставить для использования другими сценариями. Иначе говоря, используя `export`, вы можете сделать выбранные API доступными для других модулей. Функции, переменные и классы модуля, не экспортируемые явно, остаются приватными для этого модуля.

ES6 предлагает два вида использования `export`: с именованием и по умолчанию. В случае с именованными экспортами вы можете использовать ключевое слово `export` перед несколькими членами модуля (вроде классов, функций и переменных). Код в следующем файле tax.js экспортирует переменную `taxCode`, а также

функции `calcTaxes()` и `fileTaxes()`, но функция `doSomethingElse()` остается скрытой от внешних сценариев:

```
export let taxCode = 1;

export function calcTaxes() { }

function doSomethingElse() { }

export function fileTaxes() { }
```

ES6 В СРАВНЕНИИ С МОДУЛЯМИ NODE

Важно отметить, что модули ES6 разрешаются статически, что является существенным преимуществом перед модулями Node, которые используют функцию `require()`. В модулях ES путь к модулю должен быть строчным литералом. В вызове же `require()` мы можем передать выражение, вычисляемое в строку, во время выполнения.

В случае с выражениями инструменты вроде IDE, статических анализаторов и бандлеров не могут избавиться от неиспользуемого кода из импортируемого модуля, потому что инструменты не знают, какой еще код использует модуль и что используется из модуля.

Но если мы передадим в вызов `require()` строчный литерал (что мы и делаем в большинстве случаев), модули CommonJS можно также «перетряхнуть», хотя и не в той же степени, что модули ES, которые явно обозначены ключевыми словами `export` и `import`. В их случае инструменты могут ясно видеть, какие символы доступны для импорта другими модулями и что нам нужно импортировать. Инструменты могут также лучше выполнять анализ и перетряхивание дерева исходного кода, использующего модули ES6, чем того, что использует commonJS.

Когда сценарий импортирует именованные экспортированные члены модуля, имена этих членов должны быть заключены в фигурные скобки. Следующий файл main.js это демонстрирует:

```
import {taxCode, calcTaxes} from 'tax';

if (taxCode === 1) { // делает что-то }

calcTaxes();
```

Здесь `tax` относится к имени файла модуля без его расширения. Фигурные скобки представляют деструктуризацию. Модуль из файла tax.js экспортирует три члена, но нам нужно импортировать только `taxCode` и `calcTaxes`.

Один из экспортируемых членов модуля может быть отмечен как `default`, что охарактеризует его как анонимный экспорт, и другой модуль сможет в своей инструкции `import` дать ему любое имя.

Так выглядит файл my_module.js, экспортирующий функцию:

```
export default function() { // делает что-то }  ◄—— Нет точки с запятой

export let taxCode;
```

Файл main.js импортирует и именованный экспорт, и экспорт по умолчанию, одновременно присваивая имя coolFunction последнему.

```
import coolFunction, {taxCode} from 'my_module';
coolFunction();
```

Обратите внимание, что вам не нужно использовать фигурные скобки вокруг `coolFunction` (экспорт по умолчанию), но нужно вокруг `taxCode` (именованный экспорт). Сценарий, импортирующий класс, переменную или функцию, экспортированные с ключевым словом `default`, может дать им новые имена, не используя специальные ключевые слова:

```
import aVeryCoolFunction, {taxCode} from 'my_module';
aVeryCoolFunction();
```

А вот чтобы присвоить псевдоним именованному экспорту, вам придется написать, к примеру, следующее:

```
import coolFunction, {taxCode as taxCode2016} from 'my_module';
```

Инструкции модуля `import` не копируют экспортированный код, а служат в качестве ссылок. Сценарий, импортирующий модули или члены, не может их изменять, и если значения в импортированных модулях изменятся, то эти изменения тут же отразятся везде, где они были импортированы.

А.12. ТРАНСПИЛЯТОРЫ

Если вы собираетесь начать новый JS-проект, не используйте устаревший синтаксис, а применяйте синтаксис из последних спецификаций ECMAScript. Если пользователи вашего приложения должны работать со старыми браузерами, не поддерживающими последний стандарт ECMAScript, вы можете транспилировать код в ES5 или другой поддерживаемый синтаксис.

Транспиляторы (часто называемые «компиляторы») преобразуют исходный код из одного языка в исходный код другого. В контексте этого приложения вам

может потребоваться транспилировать ваш JS-код из ES6 (или более поздней версии) в ES5, прежде чем развертывать приложение в продакшене.

В экосистеме JavaScript наиболее популярный транспилятор — это Babel (http://babeljs.io). Вы можете испытать любые примеры кода из этого приложения в утилите REPL Babel (http://babeljs.io/repl), которая позволяет вводить фрагмент кода в одной из новейших версий ECMAScript и транспилировать его в ES5. На рис. А.2 показан скриншот вкладки Babel «Try it out», показывающий, как код ES2015 из листинга А.12 (слева) может быть транспилирован в ES5 (справа). Вы можете увидеть его в действии, скопировав пример кода из листинга А.12 в http://babeljs.io/repl.

Рис. А.2. Использование REPL Babel

Babel можно использовать не только для транспиляции новейшего JavaScript-синтаксиса в более старые версии, но также для преобразования TypeScript-кода в JavaScript (как можно видеть в разделе 6.4). Но как правило, вы будете транспилировать TS (в любую версию JS), используя его собственный компилятор, рассмотренный нами в разделе 1.3.

На этом завершается наше знакомство с некоторыми из наиболее важных возможностей, введенных в последних спецификациях ECMAScript. При этом также очень хорошо, что вы можете использовать все эти возможности в своих TS-программах, не дожидаясь добавления их поддержки во все браузеры.

Яков Файн, Антон Моисеев

TypeScript быстро

Перевел с английского *Д. Акуратер*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*Н. Римицан*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*М. Петруненко*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

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: 02.2021. 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 11.02.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 42,570. Тираж 700. Заказ 0000.

Борис Черный

ПРОФЕССИОНАЛЬНЫЙ TYPESCRIPT. РАЗРАБОТКА МАСШТАБИРУЕМЫХ JAVASCRIPT-ПРИЛОЖЕНИЙ

Любой программист, работающий с языком с динамической типизацией, подтвердит, что задача масштабирования кода невероятно сложна и требует большой команды инженеров. Вот почему Facebook, Google и Microsoft придумали статическую типизацию для динамически типизированного кода.

Работая с любым языком программирования, мы отслеживаем исключения и вычитываем код строку за строкой в поиске неисправности и способа ее устранения. TypeScript позволяет автоматизировать эту неприятную часть процесса разработки.

TypeScript, в отличие от множества других типизированных языков, ориентирован на прикладные задачи. Он вводит новые концепции, позволяющие выражать идеи более кратко и точно, и легко создавать масштабируемые и безопасные современные приложения.

Борис Черный помогает разобраться со всеми нюансами и возможностями TypeScript, учит устранять ошибки и масштабировать код.

Марейн Хавербеке

ВЫРАЗИТЕЛЬНЫЙ JAVASCRIPT. СОВРЕМЕННОЕ ВЕБ-ПРОГРАММИРОВАНИЕ

3-е издание

«Выразительный JavaScript» позволит глубоко погрузиться в тему, научиться писать красивый и эффективный код. Вы познакомитесь с синтаксисом, стрелочными и асинхронными функциями, итератором, шаблонными строками и блочной областью видимости.

Марейн Хавербеке — практик. Получайте опыт и изучайте язык на множестве примеров, выполняя упражнения и учебные проекты. Сначала вы познакомитесь со структурой языка JavaScript, управляющими структурами, функциями и структурами данных, затем изучите обработку ошибок и исправление багов, модульность и асинхронное программирование, после чего перейдете к программированию браузеров.

Дэн Вандеркам

ЭФФЕКТИВНЫЙ TYPESCRIPT: 62 СПОСОБА УЛУЧШИТЬ КОД

«Эффективный TypeScript» необходим тем, кто уже имеет опыт работы с JavaScript. Цель этой книги — не научить пользоваться инструментами, а помочь повысить профессиональный уровень.

TypeScript представляет собой не просто систему типов, а набор служб языка, удобных в использовании. Он повышает безопасность разработки в JavaScript, делает работу увлекательнее и проще.

Дэн Вандеркам работает главным инженером в Sidewalk Labs, а также является соучредителем митапа TypeScript NYC. Долгое время был участником открытых проектов. Принимал участие в разработке поисковой системы Google, которой пользуются миллионы людей во всем мире.