O'REILLY®

# Профессиональный TypeScript

## Разработка масштабируемых JavaScript-приложений

Борис Черный

# Programming TypeScript

## *Making Your JavaScript Applications Scale*

***Boris Cherny***

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

# Профессиональный TypeScript

## Разработка масштабируемых JavaScript-приложений

Борис Черный

ПИТЕР®

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

2021

ББК 32.988.02-018
УДК 004.738.5
Ч-49

**Борис Черный**

Ч-49 Профессиональный TypeScript. Разработка масштабируемых JavaScript-приложений. — СПб.: Питер, 2021. — 352 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1651-5

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.

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

ББК 32.988.02-018
УДК 004.738.5



ISBN 978-1492037651 англ. Authorized Russian translation of the English edition of Programming TypeScript ISBN 9781492037651 © 2019 Boris Cherny. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-1651-5 © Перевод на русский язык
ООО Издательство «Питер», 2021
© Издание на русском языке,
оформление ООО Издательство «Питер», 2021
© Серия «Бестселлеры O'Reilly», 2021

# Краткое содержание

# Оглавление

# Отзывы

Отличная книга для углубленного изучения TypeScript. Она демонстрирует все преимущества использования системы типов и помогает обрести уверенность при работе с JavaScript.

*Минко Гечев,*
*инженер команды Angular в Google*

Книга «Профессиональный TypeScript. Разработка масштабируемых JavaScript-приложений» помогла мне быстро освоить инструменты и внутреннее устройство этого языка. Она дала ответы на все мои вопросы с помощью реальных примеров. Глава «Продвинутые типы» сломала терминологические барьеры и показала, как TypeScript позволяет создать безопасный и удобный код.

*Шон Гров,*
*сооснователь OneGraph*

Борис создал обширное руководство по TypeScript. Прочтите его вдоль и поперек. А затем еще разочек.

*Блейк Эмбри, инженер в Opendoor,*
*автор TypeScript Node and Typings*
*(«Типизации и Node в TypeScript»)*

*Посвящается Саше и Михаилу.*
*Возможно, и они однажды полюбят типы.*

# Пролог

Эта книга предназначена для программистов всех направлений: JavaScript-инженеров, C#-разработчиков, сторонников Java, любителей Python, Ruby и Haskell. На каком бы языке вы ни писали, если вам известны функции, переменные, классы и связанные с ними ошибки, то эта книга для вас. Наличие некоторого опыта в JavaScript, включая базовые знания об объектной модели документа (DOM) и обмене информацией по сети, поможет вам освоить представленный материал. Хоть мы и не сильно углубляемся в эти понятия, на них основаны приведенные примеры.

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.

Если раньше вы не встречались со статической типизацией, то узнаете о типах из этой книги. Я расскажу, как с их помощью снизить риск программного сбоя, улучшить документирование кода и масштабировать его для большего числа пользователей, инженеров и серверов. Без сложных терминов я объясню материал, подключая вашу интуицию и память, и в этом мне помогут примеры.

TypeScript, в отличие от множества других типизированных языков, преимущественно практический. Он вводит новые концепции, позволяющие выражать идеи более кратко и точно и играючи создавать современные приложения безопасным способом.

## Структура книги

Я постарался передать вам теоретическое понимание работы TypeScript и достаточное количество практических советов по написанию кода.

TypeScript — практический язык, поэтому в книге теория и практика в основном дополняют друг друга, но в первых двух главах преимущественно освещается теория, а ближе к концу представлена только практика.

Мы рассмотрим такие основы, как компилятор, модуль проверки типов и сами типы. Далее обсудим их разновидности и операторы, после чего перейдем к углубленным темам, таким как особенности системы типов, обработка ошибок и асинхронное программирование. В завершение я расскажу, как использовать TypeScript с вашими любимыми фреймворками (фронтенд и бэкенд), производить миграцию существующего JavaScript-проекта в TypeScript и запускать TypeScript-приложение в продакшене.

Большинство глав завершаются набором упражнений. Попробуйте выполнить их самостоятельно, чтобы лучше усвоить материал. Ответы к ним доступны по адресу https://github.com/bcherny/programming-typescript-answers.

## Стиль

В коде я старался придерживаться единого стиля, поэтому отмечу, какие мои личные предпочтения он содержит.

- Точка с запятой только при необходимости.
- Отступы двумя пробелами.
- Короткие имена переменных вроде `a`, `f` или `_` там, где программа является фрагментом кода или ее структура важнее деталей.

Я считаю, что некоторых аспектов этого стиля стоит придерживаться и вам. К ним относятся:

- Использование синтаксиса JavaScript и новейших возможностей этого языка (последняя версия JavaScript обычно называется esnext). Так ваш код будет соответствовать последним стандартам и иметь хорошую функциональную совместимость с поисковыми системами. Это поможет снизить временные затраты на набор новых сотрудников и позволит оценить все преимущества таких мощных инструментов, как стрелочные функции, промисы и генераторы.
- Использование оператора расширения `spread` для длительного сохранения неизменности структур данных[1].

[1] Если у вас нет опыта работы в JavaScript, то вот пример: чтобы объекту `o` добавить свойство `k` со значением `3`, можно либо изменить `o` напрямую — `o.k = 3`, либо применить это изменение к `o`, создав тем самым *новый* объект — `let p = {...o, k: 3}`.

- Привычка убеждаться, что у любого элемента есть тип, по возможности выведенный. Старайтесь не злоупотреблять явными типами. Так вы сделаете код лаконичным и чистым, а также повысите его безопасность, выявляя неверные типы, а не используя временные решения.
- Стремление сохранить код переиспользуемым и обобщенным (см. раздел «Полиморфизм» на с. 90).

Конечно, все это не ново, но для TypeScript имеет особую актуальность. Его встроенный низкоуровневый компилятор, поддержка типов `readonly` (только для чтения), мощный интерфейс типов, глубокая поддержка полиморфизма и полностью структурная система типов позволяют добиться хорошего стиля, в то время как сам язык остается выразительным и верным для лежащего в его основе JavaScript.

Еще пара заметок до перехода к основному материалу.

JavaScript не предоставляет указатели и ссылки. Вместо них используются значения и ссылочные типы. Значения являются неизменными и включают такие элементы, как *strings (*строки), *numbers* (числа) и *booleans* (логические значения), в то время как ссылочные типы часто указывают на изменяемые структуры данных вроде массивов, объектов и функций. Когда я использую слово «значение», то обычно имею в виду либо значение JavaScript, либо ссылку.

И последнее. Написание идеального TypeScript-кода затрудняется взаимодействием с JavaScript, некорректно типизированными сторонними библиотеками и наследованным кодом. Также ему мешает спешка. В книге я буду часто советовать вам избегать компромиссов. В реальности достаточную корректность кода будете определять только вы и ваша команда.

## Типографские соглашения

Ниже приведен список используемых обозначений.

*Курсив*

Используется для обозначения новых терминов.

`Моноширинный`

Применяется для оформления листингов программ и программных элементов в обычном тексте, таких как имена переменных и функций, базы данных, типы данных, переменные окружения, инструкции и ключевые слова.

**`Моноширинный жирный`**

Обозначает команды или другой текст, который должен вводиться пользователем. Иногда используется в листингах для привлечения внимания.

Так обозначаются примечания общего характера.

Так выделяются советы и предложения.

Так обозначаются предупреждения и предостережения.

## Использование примеров кода

Вспомогательный материал (примеры кода, упражнения и пр.) доступен для загрузки по адресу: https://github.com/bcherny/programming-typescript-answers.

В общем случае все примеры кода из этой книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабаты-

ваете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам следует получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, получение разрешения не требуется. Но при включении существенных объемов программного кода примеров из этой книги в вашу документацию вам необходимо будет получить разрешение издательства.

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу permissions@oreilly.com.

## Благодарности

Написание этой книги заняло год и потребовало множества бессонных ночей, ранних подъемов и потерянных выходных. В ее основе лежат многолетние наработки — сниппеты и заметки, собранные воедино, доработанные и обогащенные теорией.

Хочу поблагодарить O'Reilly за предоставленную возможность работы над книгой и лично редактора Анжелу Руфино за поддержку на протяжении всего процесса. Благодарю Ника Нэнса за помощь в создании раздела «Типобезопасные API» и Шьяма Шешадри за помощь с подразделом «Angular» (см. с. 256). Спасибо моим научным редакторам: Даниэлю Розенвассеру из команды TypeScript, который провел огромное количество времени, вычитывая рукопись и знакомя меня с нюансами системы типов, а также Джонатану Кримеру, Якову Файну, Полу Байингу и Рэйчел Хэд за технические правки и обратную связь. Спасибо моей семье — Лизе и Илье, Вадиму, Розе и Алику, Фаине и Иосифу — за вдохновение на реализацию этого проекта.

Самую большую благодарность выражаю Саре Гилфорд, которая поддерживала меня на протяжении всего процесса написания, даже когда отменялись планы на вечер и выходные или я внезапно начинал рассуждать на тему различных деталей системы типов. Я бесконечно благодарен ей за поддержку, без которой точно не справился бы.

# От издательства

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

Мы будем рады узнать ваше мнение!

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

ГЛАВА 1

# Вступление

Итак, вы купили книгу о TypeScript. Зачем она нужна?

Вероятно, затем, что вам изрядно надоели странные ошибки в JavaScript вроде *cannot read property... of undefined* («невозможно прочесть свойство... принадлежащее *undefined*»). Или вы слышали, что TypeScript помогает масштабировать код, и решили изучить этот вопрос. Или вы работаете в C# и подумываете о переходе на JavaScript. А может, занимаетесь функциональным программированием и настало время повысить свой уровень. Или эту книгу вы получили в подарок на Новый год от босса, который устал от проблем, вызванных вашим кодом (если я перегибаю, остановите меня).

TypeScript — это язык будущих веб- и мобильных приложений, проектов NodeJS и IoT (систем интернет-управления устройствами). Он позволяет создавать более безопасные программы, обеспечивать их документацией, полезной и вам, и будущим инженерам, поддерживает безболезненный рефакторинг, а также избавляет от необходимости проводить половину модульных, или юнит-тестов. (Каких еще модульных тестов?) TypeScript может удвоить вашу продуктивность и даже устроить свидание с той милой бариста из кафе напротив.

Но прежде, чем спешить к ней, разберемся с упомянутыми преимуществами. Что конкретно я имею в виду, когда говорю «более безопасные»? Конечно, речь идет о *безопасности типов*.

**БЕЗОПАСНОСТЬ ТИПОВ**

Использование типов для предотвращения неверного поведения программ[1].

[1] В каждом отдельном статически типизированном языке «неверное поведение» может означать разное, начиная от программ, дающих сбой при запуске, и заканчивая рабочими, но бессмысленными приемами.

Вот несколько примеров неверного поведения кода:

- Перемножение числа и списка.
- Вызов функции со списком строк, когда требуется список объектов.
- Вызов метода для объекта, когда фактически данный метод не существует в этом объекте.
- Импортирование модуля, который недавно был перемещен.

Некоторые языки программирования стремятся извлечь пользу из подобных ошибок — стараются понять, что вы имели в виду. Возьмем, к примеру, такой JavaScript-код:

```
3 + []            // Вычисляется как строка "3"

let obj = {}
obj.foo           // Вычисляется как undefined

function a(b) {
  return b/2
}
a("z")            // Вычисляется как NaN
```

JavaScript делает все возможное, чтобы избежать исключения. Оказывается ли он полезен? Определенно да. Но можете ли вы при этом быстро и точно обнаруживать баги? Скорее всего, нет.

А теперь представьте, что JavaScript выдает больше исключений вместо попытки извлечь максимум из того, что мы ему дали. Тогда получим подобную реакцию:

```
3 + []            // Ошибка: вы точно хотели добавить число и массив?

let obj = {}
obj.foo           // Ошибка: вы забыли определить свойство "foo" в obj.

function a(b) {
  return b/2
}
a("z")            // Ошибка: функция "a" ожидает число,
                  // но вы передали ей строку.
```

Не поймите меня превратно: попытка исправить за нас наши же ошибки — это приятная особенность языка программирования (ох, если бы она работала не только для программ). Но в JavaScript она создает разрыв между моментом, когда вы допускаете ошибку, и временем ее *обнаружения*. Доходит до того, что вы узнаете об ошибке от кого-то другого.

Когда именно JavaScript сообщает об ошибке?

Или при запуске программы для тестирования в браузере, или при посещении сайта пользователем, или в начале модульного тестирования. Если вы пишете множество модульных и полных тестов, проверяя код на работоспособность, то можно рассчитывать на то, что вы увидите ошибки раньше пользователей. Но что, если вы этого не делаете?

Вот здесь-то и появляется TypeScript. И самое крутое в том, что он выдает сообщения об ошибках в момент их появления (во время типизации) в вашем редакторе. Посмотрим, что он скажет по предыдущему примеру:

```
3 + []          // Ошибка TS2365: оператор '+' не может быть применен
                // для типов '3' и 'never[]'.

let obj = {}
obj.foo       // Ошибка TS2339: свойство 'foo' не существует в типе '{}'.

function a(b: number) {
  return b / 2
}
a("z")          // Ошибка TS2345: аргумент типа '"z"' не может быть
                // присвоен параметру типа 'number'.
```

Он не только устраняет целый класс багов, связанных с типами, но и изменяет подход к написанию кода. Вы начнете делать наброски программы на уровне типов еще до начала ее наполнения на уровне значений[1], обдумывать пограничные случаи уже во время ее проектирования. В итоге вы станете проектировать более простые, быстрые, понятные и легко обслуживаемые программы.

Если вы готовы начать путешествие, тогда приступим!

---

[1] Если вы не совсем понимаете, что в данном случае значит «уровень значений», не переживайте. В следующих главах мы раскроем это понятие.

ГЛАВА 2

# TypeScript с высоты птичьего полета

В нескольких следующих главах мы рассмотрим TypeScript, работу его компилятора (TSC), а также функции и паттерны, которые вы можете разрабатывать.

## Компилятор

Ваше представление о программах во многом зависит от того, с какими языками программирования вы работали ранее. TypeScript отличается от большинства основных языков.

Программы — это файлы, содержащие прописанный вами текст. Специальная программа — компилятор — считывает и преобразует ваш текст в абстрактное синтаксическое дерево (АСД). Оно представляет собой структуру данных, игнорирующую пустые области, комментарии и ваше ценное мнение о пробелах или табуляции. Затем компилятор преобразует АСД в низкоуровневую форму — байт-код, который можно запустить в среде выполнения и получить результат. Итак, когда вы запускаете программу, фактически вы просите среду выполнения считать байт-код, сгенерированный компилятором на основе АСД, полученного из исходного кода. Детали этого процесса могут отличаться, но для большинства языков он выглядит так:

1. Программа преобразуется в АСД.
2. АСД компилируется в байт-код.
3. Байт-код считывается средой выполнения.

Особенность TypeScript в том, что вместо компиляции прямо в байт-код он компилирует в код JavaScript. Затем вы просто запускаете его в браузере,

с NodeJS или вручную, с помощью бумаги и ручки (вариант для тех, кто читает книгу во время восстания машин).

«Причем здесь безопасность кода?» — спросите вы.

Отличный вопрос. Я пропустил важный этап: после создания АСД компилятор проверяет типы.

**МОДУЛЬ ПРОВЕРКИ ТИПОВ**

Специальная программа, определяющая типобезопасность кода.

В проверке типов заключена магия TypeScript. С ее помощью он убеждается, что программа работает так, как вы ожидаете, и что приятная бариста из кафе напротив перезвонит вам (на самом деле она сейчас просто занята).

Итак, если мы добавим проверку типов и преобразование в JavaScript, то процесс компиляции TypeScript будет выглядеть примерно так (рис. 2.1).

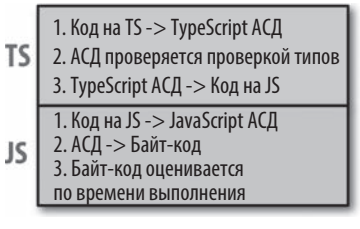


**Рис. 2.1.** Компиляция и запуск TypeScript

Шаги 1–3 производятся компилятором, а шаги 4–6 — средой выполнения JavaScript, находящейся в вашем браузере, или NodeJS, или любым другим JavaScript-движком.

JavaScript-компиляторы и среды выполнения, как правило, представляют собой единую программу, называемую движком. Будучи программистом, с ним вы и будете взаимодействовать. Так работают V8 (движок, лежащий в основе NodeJS, Chrome и Opera), SpiderMonkey (Firefox), JSCore (Safari) и Chakra (Edge). Именно поэтому JavaScript и называют *интерпретируемым* языком.

В течение всего процесса шаги 1–2 используют типы программы, а шаг 3 уже этого не делает. Стоит еще раз повториться: когда TSC компилирует код в JavaScript, он не будет смотреть на типы. Это означает, что типы никогда не смогут повлиять на сгенерированный вывод и будут использованы только для проверки типов. Эта особенность позволяет безопасно с ними экспериментировать — обновлять и улучшать их без риска сломать приложение.

## Система типов

В современных языках реализованы различные системы типов.

> **СИСТЕМА ТИПОВ**
>
> Набор правил, используемых модулем проверки типов для присвоения типов программе.

Главным образом системы типов делятся на два вида: в одних вы должны сообщать компилятору тип каждого элемента посредством явного синтаксиса, другие выводят типы автоматически. Оба вида имеют как плюсы, так и минусы[1].

TypeScript создан на стыке этих двух видов: вы можете явно аннотировать типы либо позволить TypeScript делать их вывод за вас.

Для явного объявления типов используются аннотации, которые сообщают TypeScript, что такое-то значение имеет такой-то тип. Давайте взглянем на несколько примеров (комментарии в соответствующих строках указывают типы, выведенные TypeScript):

```
let a: number = 1                    // a является number
let b: string = 'hello'              // b является string
let c: boolean[] = [true, false]     // c является массивом booleans
```

[1] JavaScript, Python и Ruby выводят типы в среде выполнения. Haskell и OCaml делают вывод типов и проверяют недостающие типы в процессе компиляции. Scala и TypeScript иногда требуют явного указания типов, вывод же и проверку остальных они производят при компиляции. Java и C нуждаются в явных аннотациях практически для всего, что проверяют в среде выполнения.

Если вы хотите, чтобы TypeScript вывел за вас типы, то просто не прописывайте их:

```
let a = 1                          // a является number
let b = 'hello'                    // b является string
let c = [true, false]              // c является массивом booleans
```

Вы сразу убедитесь, насколько хорошо он справляется с этой задачей. Убрав аннотации, вы увидите, что типы остались прежними. На протяжении всей книги мы будем использовать аннотирование только по необходимости и позволим TypeScript демонстрировать свои волшебные способности.

В большинстве случаев лучше позволять TypeScript выводить типы по мере его возможностей и снижать тем самым объем явно аннотированного кода до минимума.

## TypeScript vs JavaScript

Давайте углубимся в систему типов TypeScript и произведем сравнение с ее аналогом в JavaScript (табл. 2.1). Правильное понимание разницы является ключом для построения образной модели функционирования TypeScript.

**Таблица 2.1.** Сравнение систем типов JavaScript и TypeScript

| Система типов | JavaScript | TypeScript |
|---|---|---|
| Как связываются типы | Динамически | Статически |
| Конвертируются ли типы автоматически | Да | Нет (в основном) |
| Когда проверяются типы | Во время выполнения | Во время компиляции |
| Когда вскрываются ошибки | Во время выполнения (в основном) | Во время компиляции (в основном) |

### Как связываются типы

Динамическое связывание типов подразумевает, что JavaScript знакомится с типами в программе только после ее запуска.

TypeScript является языком *с постепенной типизацией* — он работает лучше, если знает все типы программы во время компиляции. Но даже

в нетипизированной программе TypeScript может вывести часть типов и обнаружить малую долю ошибок; остальные же ошибки могут просочиться к конечным пользователям.

Постепенная типизация очень полезна при миграции наследованных баз кода из нетипизированного JavaScript в типизированный TypeScript (см. раздел «Поэтапная миграция из JavaScript в TypeScript» на с. 292), но пока вы не достигнете середины процесса миграции, стремитесь к 100%-ной типизации кода. Именно этот подход используется в книге, за исключением обозначенных случаев.

#### Конвертируются ли типы автоматически

JavaScript является слабо типизированным языком, поэтому если вы произведете недопустимое сложение, например числа и массива (как в главе 1), то он применит множество правил для выяснения вашего намерения, чтобы выдать наилучший результат. Рассмотрим пример того, как JavaScript определяет значение `3 + [1]`:

1. JavaScript замечает, что `3` является числом, а `[1]` — массивом.
2. Увидев `+`, он предполагает, что вы хотите произвести их конкатенацию.
3. Он неявно преобразует `3` в строку, создавая `"3"`.
4. Он также неявно преобразует в строку `[1]`, создавая `"1"`.
5. Производит конкатенацию этих результатов: `"31"`.

Мы могли бы сделать это и более явно (так JavaScript избежит шагов 1, 3 и 4):

```
3 + [1];                          // вычисляется как "31"
(3).toString() + [1].toString()   // вычисляется как "31"
```

В то время как JavaScript старается произвести умные преобразования в стремлении вам помочь, TypeScript начинает указывать на ошибки, как только вы делаете что-либо неверно. Если вы запустите тот же код через TSC, то получите ошибку:

```
3 + [1];                          // Ошибка TS2365: оператор '+'
                                  // не может быть применен к типам '3'
                                  // и 'number[]'.
(3).toString() + [1].toString()   // вычисляется как "31".
```

Как только вы делаете нечто, что выглядит неправильным, TypeScript на это указывает. Если же вы делаете свои намерения явными, он перестает препятствовать. В таком поведении есть смысл: кто бы, находясь в здравом уме, стал складывать число и массив, ожидая в результате получить строку? (Конечно, не считая JavaScript-ведьмы Бавморды, которая пишет код при свечах в гараже, где устроился ваш стартап.)

Неявное преобразование, которое производит JavaScript, может серьезно усложнить обнаружение источника ошибок, особенно при масштабировании проекта крупной командой разработчиков, где каждый инженер будет вынужден понять неявные предположения, формулируемые кодом.

Вывод: если вам необходимо конвертировать типы, делайте это явно.

#### Когда проверяются типы

В большинстве случаев JavaScript не интересует, какие вы предоставляете ему типы, но при этом он старается преобразовать то, что вы предоставили, в то, что он сам ожидает.

TypeScript же, напротив, проверяет типы в процессе компиляции (шаг 2 из списка в начале главы), поэтому вам не нужно запускать код, чтобы увидеть ошибку из предыдущего примера. TypeScript статически анализирует код на наличие подобных ошибок и показывает их еще до запуска. Если код не проходит компиляцию, то это явный признак присутствия ошибки, которую нужно исправить до запуска кода.

Рис. 2.2 показывает, что происходит при типизации последнего примера кода в VSCode (предпочтенный мной редактор).

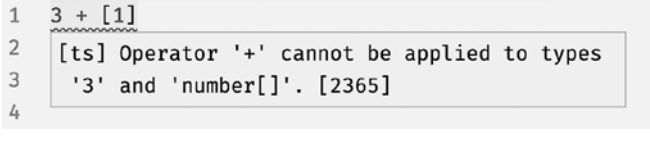


**Рис. 2.2.** VSCode сообщает об ошибке типа

Если в вашем редакторе установлено хорошее расширение TypeScript, то ошибка будет подчеркнута красной волнистой линией при типизации кода. Это существенно ускорит цикл обратной связи между написанием кода, осознанием допущенной ошибки и обновлением кода с исправлением.

#### Когда вскрываются ошибки

JavaScript выбрасывает исключения или производит неявные преобразования типов в среде выполнения[1]. Это означает, что для получения отклика об ошибке необходимо запустить программу. В лучшем случае это станет частью модульного теста, в худшем — вы получите электронное письмо от недовольного пользователя.

TypeScript выдает и синтаксические ошибки, и ошибки типов во время компиляции. Это означает, что эти ошибки будут отображены в редакторе сразу после типизации — это вас удивит, если раньше вы не имели дело с инкрементно компилируемым языком со статической типизацией[2].

К слову, есть множество возможных ошибок, которые TypeScript не может обнаружить при компиляции. К ним относятся переполнения стека, разрывы сетевых соединений и некорректный ввод данных пользователем. Все они по-прежнему будут производить исключения при выполнении. Что же TypeScript действительно делает хорошо, так это вычисляет ошибки при компиляции, которые в противном случае стали бы ошибками при выполнении в среде чистого JavaScript.

## Настройка редактора кода

Теперь, когда вы уже имеете представление о работе компилятора TypeScript и его системе типов, мы можем переходить к настройке редактора и погружению в сам процесс написания кода.

Начните с выбора редактора и его загрузки. Лично мне нравится VSCode, потому что в нем редактирование TypeScript-кода особенно удобно, но вы можете рассмотреть Sublime Text, Atom, Vim, WebStorm и др. Как правило, инженеры весьма требовательны к ИСР (интегрированной среде разработки), поэтому оставлю этот выбор за вами. Если же вы желаете использовать VSCode, то для его установки просто следуйте инструкциям на сайте https://code.visualstudio.com/.

---

[1] Без сомнения, JavaScript вскрывает синтаксические ошибки и способен обнаружить некоторые баги (вроде множественных деклараций `const` с одним именем и в одном диапазоне) после считывания программы, но до ее запуска. Если вы считываете JavaScript-код в процессе сборки (например, в Babel), то эти ошибки тоже обнаружатся.

[2] Инкрементно компилируемые языки позволяют при внесении небольших изменений произвести быструю перекомпиляцию вместо перекомпилирования всей программы (включая незатронутые ее части).

Сам по себе TSC — это приложение командной строки, написанное в TypeScript[1], поэтому для его запуска понадобится NodeJS. Для его установки также есть инструкции на официальном сайте.

NodeJS поставляется с NPM — пакетным менеджером, который нужен для управления зависимостями проекта и сборкой. Мы начнем с его использования для установки TSC и TSLint (линтер для TypeScript). Откройте терминал и создайте новый каталог, затем инициализируйте в нем новый проект NPM:

```
# Создание каталога
mkdir chapter-2
cd chapter-2

# Инициализация нового проекта NPM (следуйте инструкциям)
npm init

# Установка TSC, TSLint и деклараций типов для NodeJS
npm install --save-dev typescript tslint @types/node
```

## tsconfig.json

Каждый TypeScript-проект должен содержать в корневой директории файл tsconfig.json. По нему TypeScript ориентируется, какие файлы и в какую директорию компилировать, а также выясняет, какая версия JavaScript требуется на выходе.

Создайте файл с именем tsconfig.json в корневом каталоге (touch tsconfig.json)[2], затем откройте его в редакторе и внесите следующее:

```
{
    "compilerOptions": {
        "lib": ["es2015"],
        "module": "commonjs",
        "outDir": "dist",
        "sourceMap": true,
```

[1] Таким образом, TSC попадает в разряд так называемых компиляторов с самостоятельным хостингом, которые компилируют свой собственный исходный код.

[2] В данном примере мы создаем tsconfig.json вручную. В дальнейшем вы можете использовать встроенную команду TSC, чтобы генерировать этот файл автоматически: ./node_modules/.bin/tsc --init.

```
        "strict": true,
        "target": "es2015"
    },
    "include": [
        "src"
    ]
}
```

Кратко пройдемся по значениям некоторых из перечисленных опций (табл. 2.2).

**Таблица 2.2.** Опции tsconfig.json

| Опция | Описание |
|---|---|
| `include` | В каких каталогах TSC должен искать файлы TypeScript? |
| `lib` | Наличие каких API в вашей среде разработки должен предполагать TSC? Это касается также таких элементов, как Function.prototype.bind в ES5, Object.assign в ES2015 и document DOM.querySelector? |
| `module` | В какую модульную систему должен производить компиляцию TSC (CommonJS, SystemJS, ES2015 и пр.)? |
| `outDir` | В какой каталог TSC должен помещать сгенерированный JavaScript-код? |
| `strict` | Как производить максимально строгую проверку кода и соблюдать правильную типизацию? Мы будем использовать ее во всех примерах. Активируйте ее в своем проекте |
| `target` | В какую версию JavaScript нужно компилировать код (ES3, ES5, ES2015, ES2016 и пр.)? |

Это лишь несколько из десятков опций, которые поддерживает tsconfig.json, причем постоянно добавляются новые. На деле вы не будете часто их менять, за исключением набора номера в настройках `module` и `target` при переключении на новый бандлер модулей, добавления `"dom"` к `lib` при написании TypeScript для браузера (см. главу 12) или изменения уровня строгости проверки кода при миграции JavaScript-проекта в TypeScript (см. раздел «Поэтапная миграция из JavaScript в TypeScript» на с. 292). Самый полный и актуальный список доступных опций находится в документации на сайте TypeScript (http://bit.ly/2JWfsgY).

Использование файла tsconfig.json для конфигурирования TSC очень удобно, поскольку позволяет проверять конфигурацию в системе контроля версий. Вы также можете установить бóльшую часть опций TSC через командную строку: запустите ./node_modules/.bin/tsc --help для вывода их списка.

### tslint.json

В вашем проекте также должен присутствовать файл tslint.json, содержащий конфигурацию TSLint, определяющую необходимую вам стилистику кода (табуляции, пробелы и пр.).

Использование TSLint не обязательно, но для любого проекта TypeScript настоятельно рекомендуется придерживаться определенной стилистики, чтобы избежать споров с коллегами во время код-ревью.

Следующая команда сгенерирует файл tslint.json со стандартной конфигурацией TSLint:

```
./node_modules/.bin/tslint --init
```

Далее вы можете переписать ее согласно своему стилю. Например, мой файл tslint.json выглядит так:

```
{
    "defaultSeverity": "error",
    "extends": [
        "tslint:recommended"
    ],
    "rules": {
        "semicolon": false,
        "trailing-comma": false
    }
}
```

Полный список доступных правил содержится в документации TSLint. Вы также можете добавлять пользовательские правила или устанавливать дополнительные настройки (как для ReactJS: https://www.npmjs.com/package/tslint-react).

## index.ts

Теперь, когда вы настроили tsconfig.json и tslint.json, создайте каталог src, содержащий ваш первый файл TypeScript:

```
mkdir src
touch src/index.ts
```

Структура каталога проекта должна выглядеть так:

```
chapter-2/
├──node_modules/
├──src/
│  └──index.ts
├──package.json
├──tsconfig.json
└──tslint.json
```

Откройте src/index.ts в редакторе и введите следующий код:

```
console.log('Hello TypeScript!')
```

Затем скомпилируйте и запустите TypeScript-код:

```
# Скомпилируйте код с помощью TSC
./node_modules/.bin/tsc

# Запустите код с помощью NodeJS
node ./dist/index.js
```

Если вы следовали перечисленным шагам, то код запустится и в консоли вы увидите запись:

```
Hello TypeScript!
```

Вот и все — вы только что настроили и запустили ваш первый TypeScript с нуля. Отличная работа!

Для первого запуска проекта TypeScript с нуля я привел пошаговую инструкцию. В дальнейшем для ускорения этого процесса можете использовать пару сокращений.

- Установите ts-node (https://www.npmjs.com/package/ts-node) и используйте его для компиляции и запуска программы посредством всего одной команды.
- Используйте инструмент автоматической генерации typescript-node-starter (https://github.com/Microsoft/TypeScript-Node-Starter) для быстрого создания структуры каталога.

# Упражнения к главе 2

Теперь, когда ваша среда настроена, откройте src/index.ts в редакторе и введите следующий код:

```
let a = 1 + 2
let b = a + 3
let c =
{
    apple: a,
    banana: b
}
let d = c.apple * 4
```

Наведите курсор на a, b, c, d и обратите внимание, как TypeScript выводит типы всех переменных: a является number, b и d также являются number, а c — это объект определенной формы (рис. 2.3).

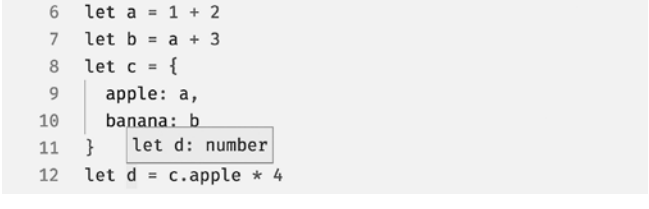


**Рис. 2.3.** TypeScript выводит типы за вас

Поэкспериментируйте с кодом и посмотрите, сможете ли вы:

- Спровоцировать TypeScript использовать красную волнистую линию, указывающую на ошибку (мы зовем это выдачей TypeError (ошибки типа)).
- Прочитать TypeError и понять, что она значит.
- Исправить TypeError, то есть убрать красную линию.

Если вы готовы, попробуйте написать отрезок кода, для которого TypeScript не сможет вывести тип.

ГЛАВА 3

# Подробно о типах

В предыдущей главе мы говорили о системах типов, но не дали определения типам.

> **ТИП**
>
> Набор значений и применимых к ним операций.

Например:

- Тип `boolean` представляет набор всех логических значений (их два: `true` и `false`) и операций, применимых к ним (вроде `||`, `&&` и `!`).
- Тип `number` представляет набор всех чисел и допустимых для них операций (вроде `+`, `-`, `*`, `/`, `%`, `||`, `&&` и `?`), включая методы, которые можно для них вызывать, такие как `.toFixed`, `.toPrecision`, `.toString`.
- Тип `string` представляет набор всех строк и производимых с ними операций (вроде `+`, `||` и `&&`), включая методы, которые можно для них вызывать, такие как `.concat`, `.toUpperCase`.

Встречая тип, вы сразу понимаете, что можно и чего нельзя сделать с его обладателем. Модуль проверки типов не позволяет вам совершать неверные действия, ориентируясь на то, какие типы и как именно вы используете.

В этой главе мы разберем все доступные в TypeScript типы и рассмотрим основные возможности применения каждого из них.

На рис. 3.1 приведена их иерархия.

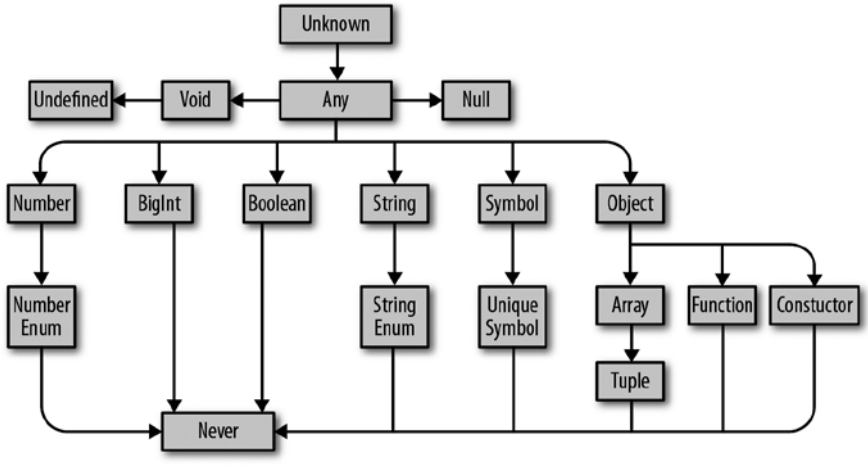


**Рис. 3.1.** Иерархия типов в TypeScript

# О типах

Для обсуждения типов воспользуемся общепринятой у программистов терминологией.

Допустим, функция получает значение и возвращает его умноженным на само себя:

```
function squareOf(n) {
    return n * n
}
squareOf(2)    // вычисляется как 4
squareOf('z')  // вычисляется как NaN
```

Ясно, что эта функция работает только с числами: если передать в `squareOf` не число, то результат будет ошибочным. Итак, в этой ситуации мы прибегаем к аннотированию типа параметра:

```
function squareOf(n: number) {
    return n * n
}
squareOf(2)    // вычисляется как 4
squareOf('z')  // Ошибка TS2345: Аргумент типа '"z"' не может быть
               // присвоен параметру типа 'number'.
```

Теперь, если мы вызовем `squareOf` со значением, отличным от числового, TypeScript будет иметь основание для указания ошибки. Это тривиальный пример (подробнее поговорим о функциях в следующей главе), но его достаточно для демонстрации пары ключевых концепций обсуждения типов в TypeScript. О последнем примере мы можем сказать следующее:

1. Параметр `n`, принадлежащий `squareOf`, ограничен типом `number`.
2. Тип значения `2` может быть присвоен (совместим с) `number`.

Без аннотации типа `squareOf` не ограничен своим параметром и вы можете передать ему аргумент любого типа. Как только вы введете ограничение, TypeScript проверит, во всех ли местах функция вызывается с подходящим аргументом. В нашем примере `2` имеет тип `number`, который совместим с аннотацией `squareOf number`, поэтому TypeScript принимает такой код. Но `'z'` является `string`, которая несовместима с `number`, и TypeScript указывает на ошибку.

Вы также можете интерпретировать это в выражениях пределов: если сообщить TypeScript, что верхний предел `n` — это `number`, любое значение, передаваемое `squareOf`, не будет превышать `number`. В противном случае значение не сможет быть присвоенным `n`.

Более формальное определение совместимости пределов и ограничений вы найдете в главе 6. Сейчас важно понять, что перед вами язык, обсуждаемый в контексте использования некоего типа там, где требуется конкретизировать тип.

## Типы от a до я

Познакомимся с имеющимися в TypeScript типами и содержащимися в них значениями и возможностями и затронем некоторые относящиеся к ним базовые особенности языка, а именно псевдонимы типов, типы объединения и типы пересечения.

### any

`any` выступает в роли крестного отца всех типов. За соответствующую плату он готов, но лишний раз прибегать к его помощи вы вряд ли захотите. В TypeScript в момент компиляции у всего должен быть тип, и `any` стано-

вится типом по умолчанию там, где вы (программист) и TypeScript (модуль проверки) не можете точно определить тип элемента. Можно охарактеризовать его как тип крайнего случая.

Почему же его стоит избегать? Вспомните, чем является тип? Правильно. Это набор значений и доступных для них действий. `any` же представляет собой набор всех возможных значений, и вы можете делать с ним все что угодно: прибавлять к нему, умножать на него, вызывать для него `.pizza()` и т. д.

Значение, имеющее тип `any`, не дает возможности модулю проверки творить свое волшебство. Когда вы разрешаете `any` в коде, вы переходите в режим «слепого полета».

В тех редких случаях, когда действительно необходимо его использовать, делайте это так:

```
let a: any = 666            // any
let b: any = ['danger']     // any
let c = a + b               // any
```

Обратите внимание, что третий тип должен вызвать сообщение об ошибке (ведь вы пытаетесь сложить число и массив), но этого не происходит, потому что вы сообщили TypeScript, что складываете два `any`. Необходимо использовать `any` явно. Когда TypeScript выводит тип некоего значения как `any` (например, вы забыли аннотировать параметр функции или импортировали нетипизированный JavaScript-модуль), то в процессе компиляции он выбросит исключение и подчеркнет ошибку красным. Используя же явную аннотацию `a` и `b` типом `any` (`:any`), вы избежите исключения, потому что таким образом дадите TypeScript понять осознанность этого действия.

**TSC-ФЛАГ NOIMPLICITANY**

По умолчанию TypeScript не жалуется на значения, для которых он вывел тип any (неявные any). Чтобы активировать функцию защиты от неявных any, нужно добавить флаг noImplicitAny в tsconfig.json.

noImplicitAny становится активна при включении режима `strict` в tsconfig.json (см. подраздел «tsconfig.json» на с. 28).

## unknown

Если any — это крестный отец, то unknown — это Киану Ривз в роли Джонни Юты из фильма «На гребне волны» — самоуверенный агент ФБР под прикрытием, который втирается в круг плохих парней, но хранит верность закону. В случае, когда у вас есть значение, чей тип вы узнаете позднее, вместо any примените unknown. Он представляет любое значение, но чтобы использовать это значение, TypeScript потребует уточнить его тип (см. подраздел «Уточнение» на с. 160).

Какие же операции поддерживает unknown? Вы можете сравнивать значения unknown (==, ===, ||, && и ?), отрицать их (!) и уточнять (как и любой другой тип через JavaScript-операторы typeof и instanceof). Применяется же он следующим образом:

```
let a: unknown = 30        // unknown
let b = a === 123          // boolean
let c = a + 10             // Ошибка TS2571: объект имеет тип 'unknown'.
if (typeof a === 'number') {
    let d = a + 10          // number
}
```

Этот пример дает общее представление об использовании unknown.

1. TypeScript никогда не выводит unknown — этот тип нужно явно аннотировать (a)[1].
2. Можно сравнивать значения со значениями типа unknown (b).
3. Нельзя производить действия на основе предположения, что значение unknown имеет конкретный тип (c). Сначала нужно показать TypeScript наличие этого типа (d).

## boolean

Тип boolean (логический) имеет всего два значения: true и false. Такие типы можно сравнивать (==, ===, ||, && и ?) и отрицать (!). Используются они так:

[1] Почти никогда. Если unknown является частью типа объединения, результатом объединения будет unknown (см. подраздел «Типы объединения и пересечения» на с. 50).

```
let a = true              // boolean
var b = false             // boolean
const c = true            // true
let d: boolean = true     // boolean
let e: true = true        // true
let f: true = false       // Ошибка TS2322: тип 'false' не может быть
                          // присвоен типу 'true'.
```

Согласно этому примеру, чтобы сообщить TypeScript, что некий элемент имеет тип `boolean`, можно сделать следующее:

1. Позволить TypeScript вывести тип `boolean` для значения (`a` и `b`).
2. Позволить TypeScript вывести конкретное значение `boolean` (`c`).
3. Явно сообщить TypeScript, что значение является `boolean` (`d`).
4. Явно сообщить TypeScript, что значение имеет конкретное значение `boolean` (`e` и `f`).

В большинстве случаев используются первые два способа, очень редко (если повышается типобезопасность) — четвертый (увидите на примерах) и практически никогда — третий.

Второй и четвертый случаи особенно интересны, поскольку хоть они и делают нечто естественное, поддерживаются они на удивление малым количеством языков и могут оказаться для вас незнакомыми. В том примере я обратился к TypeScript: «Видишь эту переменную `e`? Она не просто какой-то там `boolean`, а совершенно конкретный `Boolean` — `true`». Использовав значение в качестве типа, я существенно ограничил возможные для `e` и `f` значения `boolean`, определив для каждого конкретное, — создал литералы типов.

> **ЛИТЕРАЛ ТИПА**
>
> Тип, представляющий только одно значение и ничто другое.

В четвертом случае я явно аннотировал переменные литералами типов, а во втором TypeScript вывел литералы типов за меня, потому что я использовал `const` вместо `let` или `var`. TypeScript знает, что значение примитива, присвоенного с `const`, не меняется, и выводит для этой переменной

максимально узкий тип. Именно поэтому во втором случае TypeScript вывел тип `c` как `true` вместо `boolean`. Вывод типов для `let` и `const` подробно рассмотрен в подразделе «Расширение типов» на с. 155.

Мы еще будем возвращаться к литералам типов. Они являются мощным инструментом типобезопасности и делают TypeScript уникальным среди других языков программирования, позволяя вам подтрунивать над друзьями, использующими Java.

## number

Тип `number` представляет набор чисел: целочисленные, с плавающей запятой, положительные, отрицательные, `Infinity` (бесконечность), `NaN` и т. д. Для чисел доступно достаточно много действий: сложение (`+`), вычитание (`-`), деление по модулю (`%`) и сравнение (`<`). Рассмотрим примеры:

```
let a = 1234                // number
var b = Infinity * 0.10     // number
const c = 5678              // 5678
let d = a < b               // boolean
let e: number = 100         // number
let f: 26.218 = 26.218      // 26.218
let g: 26.218 = 10          // Ошибка TS2322: тип '10' не может быть
                            // присвоен типу '26.218'.
```

Для типизации с помощь `number` можно сделать следующее:

1. Позволить TypeScript вывести тип значения как `number` (`a` и `b`).
2. Использовать `const`, чтобы TypeScript вывел тип переменной как конкретное число (`number`) (`c`)[1].
3. Явно сообщить TypeScript, что значение имеет тип `number` (`e`).
4. Явно сообщить, что значение имеет конкретный тип `number` (`f` и `g`).

Так же как и с `boolean`, лучше доверить вывод типа TypeScript (первый вариант). Иногда при утонченном описании программы потребуется ограничить тип конкретным значением (второй и четвертый варианты). На практике нет причин явно типизировать значение как `number` (третий вариант).

[1] На момент написания книги применение `NaN`, `Infinity` или `-Infinity` в качестве литералов типов не допускается.

Работая с длинными числами, используйте десятичный разделитель как в значениях, так и в типах для их удобного чтения:

```
let oneMillion = 1_000_000 // Эквивалент 1000000
let twoMillion: 2_000_000 = 2_000_000
```

## bigint

Тип `bigint` является новичком в JavaScript и TypeScript. Он позволяет работать с крупными целочисленными значениями без риска столкнуться с ошибками округления. В то время как `number` может представлять целые числа только до $2^{53}$, `bigint` способен представить и бóльшие значения. Этот тип является набором всех `BigInts` и поддерживает такие действия, как сложение (`+`), вычитание (`-`), умножение (`*`), деление (`/`) и сравнение (`<`). Используется он так:

```
let a = 1234n           // bigint
const b = 5678n         // 5678n
var c = a + b           // bigint
let d = a < 1235        // boolean
let e = 88.5n           // Ошибка TS1353: литерал bigint должен быть
                        // целым числом.
let f: bigint = 100n    // bigint
let g: 100n = 100n      // 100n
let h: bigint = 100     // Ошибка TS2322: тип '100' не может быть
                        // присвоен типу 'bigint'.
```

Аналогично `boolean` и `number` есть четыре варианта объявления `bigint`. Старайтесь позволять TypeScript делать их вывод за вас.

На момент написания книги еще не все движки JavaScript поддерживают `bigint` по умолчанию. Если ваше приложение зависит от него, то стоит убедиться в его поддержке на целевой платформе.

## string

Тип `string` является набором всех строк и доступных для них операций вроде конкатенации (`+`), среза (`.slice`) и т. д. Вот несколько примеров:

```
let a = 'hello'            // string
var b = 'billy'            // string
const c = '!'              // '!'
let d = a + ' ' + b + c    // string
let e: string = 'zoom'     // string let f: 'john' = 'john' // 'john'
let g: 'john' = 'zoe'      // Ошибка TS2322: тип "zoe" не может быть
                           // присвоен типу "john".
```

Как и в случае с `boolean` и `number`, тип `string` можно объявить одним из четырех способов, и лучше, если это сделает сам TypeScript.

## symbol

Тип `symbol` появился с одной из последних ревизий JavaScript (ES2015). На практике символы встречаются нечасто. Они используются в качестве альтернативы строчных ключей в объектах и картах там, где вы хотите быть абсолютно уверены, что пользователи применяют верный, хорошо известный ключ и по случайности не установят другой. Подумайте, устанавливая итератор по умолчанию для объекта (`Symbol.Iterator`) или переназначая ключ в среде выполнения, не является ли объект экземпляром чего-либо (`Symbol.Instance`). Символы обозначаются типом `symbol` и не предполагают много действий:

```
let a = Symbol('a')            // symbol
let b: symbol = Symbol('b')    // symbol
var c = a === b                // boolean
let d = a + 'x'                // Ошибка TS2469: оператор '+' не может
                               // быть применен к типу 'symbol'.
```

Задача `Symbol('a')` в JavaScript заключается в создании нового `symbol` с заданным именем. Этот `symbol` уникален и не будет равен (при сравнении посредством `==` или `===`) никакому другому `symbol` (даже если создать второй одноименный `symbol`). Аналогично тому, как значение `27` приобретает тип `number` при объявлении с помощью `let` и становится конкретным числом при объявлении с `const`, типы символов выводятся как `symbol`, но также могут быть объявлены явно как `unique symbol`:

```
const e = Symbol('e')                 // typeof e
const f: unique symbol = Symbol('f')  // typeof f
let g: unique symbol = Symbol('f')    // Ошибка TS1332: переменная,
                                      // имеющая тип 'unique symbol',
                                      // должна быть 'const'.
```

```
let h = e === e     // boolean
let i = e === f     // Ошибка TS2367: это условие всегда будет
                    // возвращать 'false', поскольку типы 'unique
                    // symbol' и 'unique symbol' не имеют сходства.
```

Этот пример демонстрирует несколько особенностей создания уникального символа:

1. Когда вы объявляете новый `symbol` и присваиваете его переменной `const` (не `var` или `let`), TypeScript выводит ее тип как `unique symbol`. В редакторе она будет отображаться не как `unique symbol`, а как `typeof имяПеременной`.
2. Вы можете явно аннотировать тип `const`-переменной как `unique symbol`.
3. `unique symbol` всегда равен самому себе.
4. TypeScript при компиляции знает, что `unique symbol` никогда не будет равен никакому другому `unique symbol`.

Воспринимайте `unique symbol` так, как и другие типы литералов вроде `1`, `true` или `"literal"`. Они являются способом создать тип, представляющий конкретное значение `symbol`.

## Объекты

Тип `object` в TypeScript определяет форму объекта. Примечательно, что он не отражает разницу между простыми объектами (вроде созданных с помощью `{}`) и более сложными (созданными с помощью `new`). Так и было задумано: JavaScript преимущественно структурно типизирован, поэтому TypeScript избегает номинальной типизации.

> **СТРУКТУРНАЯ ТИПИЗАЦИЯ**
>
> Стиль программирования, в котором вас интересуют только конкретные свойства объекта, а не его имя (номинальная типизация). В некоторых языках это называют утиной типизацией (или «не судите книгу по обложке»).

В TypeScript есть несколько способов использования типов для описания объектов. Первый заключается в объявлении значения в качестве `object`:

```
let a: object = {
    b: 'x'
}
```

Что произойдет, когда вы обратитесь к b?

```
a.b   // Ошибка TS2339: свойство 'b' не существует в типе 'object'.
```

Но какой смысл присваивать чему-либо тип object, если потом ничего нельзя делать?

На самом деле смысл есть. В малой степени object является any. Этот тип сообщает вам о значении, которому он присвоен, только то, что оно является объектом JavaScript (и оно не null).

### ВЫВОД ТИПОВ ПРИ ОБЪЯВЛЕНИИ ОБЪЕКТОВ С CONST

Что произойдет, если для объявления объекта использовать const?

```
const a: {b: number} = {
    b: 12
}                // По-прежнему {b: number}
```

Вас может удивить, что TypeScript вывел тип b как number, а не литерал 12. Ранее мы выяснили, что при объявлении типов number или string выбор const или let влияет на вывод типов.

В отличие от примитивных типов, рассмотренных прежде (boolean, number, bigint, string и symbol), объект, объявленный с const, не подскажет TypeScript вывести более узкий тип. Дело в том, что JavaScript-объекты изменяемы и TypeScript знает, что их поля можно обновить после создания.

В подразделе «Расширение типов» на с. 155 мы более подробно рассмотрим эту тему, а также то, как добиться вывода более узкого типа.

А что, если мы оставим явные аннотации и позволим TypeScript делать свою работу?

```
let a = {
    b: 'x'
}                // {b: string}
a.b              // string
```

```
let b = {
    c: {
        d: 'f'
    }
}                 // {c: {d: string}}
```

Вуаля! Вы только что открыли второй способ типизации объекта: синтаксис объектного литерала. Вы можете либо позволить TypeScript вывести форму объекта, либо описать ее явно, используя фигурные скобки {}:

```
let a: {b: number} = {
    b: 12
}                   // {b: number}
```

Синтаксис объектного литерала сообщает: «Здесь находится элемент такой формы». Этот элемент может быть либо объектным литералом, либо классом:

```
let c: {
    firstName: string
    lastName: string
} = {
    firstName: 'john',
    lastName: 'barrowman'
}

class Person {
    constructor(
        public firstName: string,     // public является сокращением
                                      // this.firstName = firstName
        public lastName: string
    ) {}
}
c = new Person('matt', 'smith')         // OK
```

`{firstName: string, lastName: string}` описывает форму объекта. Объектный литерал и экземпляр класса из последнего примера соответствуют этой форме, поэтому TypeScript позволяет присвоить `Person` литералу `c`.

Вот что произойдет, если добавить дополнительные свойства или упустить необходимые:

```
let a: {b: number}

a = {}      // Ошибка TS2741: свойство 'b' отсутствует в типе '{}',
            // но необходимо в типе '{b: number}'.

a = {
    b: 1,
    c: 2    // Ошибка TS2322: тип '{b: number; c: number}' не может
}           // быть присвоен типу '{b: number}'. Объектный литерал
            // может определять только известные свойства,
            // а 'c' не существует в типе '{b: number}'.
```

**ПРИСВОЕНИЕ**

Это первый пример, когда сначала мы объявляем переменную (`a`), а затем ее инициализируем со значениями (`{}` и `{b:1, c:2}`). Такой паттерн JavaScript поддерживается в TypeScript.

Когда вы объявляете переменную в одном месте, а инициализируете ее позже, TypeScript убеждается, что этой переменной точно присвоено значение на момент ее использования:

```
let i: number
let j = i * 3       // Ошибка TS2454: переменная 'i' используется
                    // до присвоения ей значения.
```

TypeScript проследит за этим, даже если вы не сделаете явные аннотации типов:

```
let i
let j = i * 3          // Ошибка TS2532: объект, вероятно,
                       // 'undefined'.
```

По умолчанию TypeScript достаточно строг относительно свойств объекта. Если вы сообщаете, что объект должен иметь свойство с именем `b` типа `number`, то TypeScript будет ожидать `b` и только `b`. Если `b` будет отсутствовать или появятся дополнительные свойства, то он будет ругаться.

Можно ли сообщить TypeScript, что нечто является опциональным или возможно появление других запланированных свойств? Конечно:

```
let a: {
    b: number ❶
    c?: string ❷
    [key: number]: boolean ❸
}
```

❶ a имеет свойство b, являющееся number.

❷ a может иметь свойство c, являющееся string. Если c задано, то оно может быть undefined.

❸ a может иметь любое количество численных свойств, являющихся boolean.

Рассмотрим, какие типы объектов можно присвоить a:

```
a = {b: 1}
a = {b: 1, c: undefined}
a = {b: 1, c: 'd'}
a = {b: 1, 10: true}
a = {b: 1, 10: true, 20: false}
a = {10: true}              // Ошибка TS2741: свойство 'b' упущено
                            // в типе'{10: true}'.
a = {b: 1, 33: 'red'}       // Ошибка TS2741: тип 'string' не может
                            // быть присвоен типу 'boolean'.
```

При объявлении типа object можно использовать как модификатор опциональности (?), так и модификатор readonly, который не позволит изменять поле после присвоения ему первого значения (своего рода const для свойств объекта):

```
let user: {
     readonly firstName: string
} = {
     firstName: 'abby'
}

user.firstName           // string
user.firstName =
     'abbey with an e'   // Ошибка TS2540: невозможно присвоить
                         // к 'firstName', т.к. это свойство только
                         // для чтения.
```

Каждый тип, за исключением null и undefined, может быть присвоен типу пустого объекта ({}), что усложняет его использование. Старайтесь избегать типов пустых объектов:

```
let danger: {}
danger = {}
danger = {x: 1}
danger = []
danger = 2
```

**СИГНАТУРЫ ИНДЕКСОВ**

Синтаксис `[key: T]: U` называется *сигнатурой индекса*. С ее помощью вы сообщаете компилятору, что данный объект может содержать больше ключей. Читать его следует так: «Для этого объекта все ключи типа `T` должны иметь значения типа `U`». Сигнатуры индекса позволяют безопасно добавлять дополнительные ключи объекту, помимо объявленных ранее.

Но тип (`T`) ключа сигнатуры индекса должен быть совместим либо со `string`, либо с `number`[1].

В качестве имени ключа сигнатуры индекса можно использовать любое слово — не только `key`:

```
let airplaneSeatingAssignments: {
     [seatNumber: string]: string
} = {
     '34D': 'Boris Cherny',
     '34E': 'Bill Gates'
}
```

И стоит упомянуть еще один способ типизации чего-либо в качестве объекта: `Object`. Это практически то же самое, что использование `{}`, и лучше этого избегать[2].

[1] Объекты в JavaScript используют для ключей строки. Массивы же являются особым видом объектов и используют в качестве ключей числа.

[2] Есть одно небольшое техническое отличие: `{}` позволяет определять любые желаемые типы для встроенных методов в прототипе `Object` вроде `.toString` и `.hasOwnProperty` (подробно о прототипах можете узнать на MDN), в то время как `Object` следит, чтобы объявляемые вами типы могли быть присвоены к типам прототипа `Object`. Например, следующий код проходит проверку типов: `let a: {} = {toString() { return 3}}`. Но, если вы измените аннотацию типа на `Object`, то TypeScript укажет ошибку: `let b: Object = {toString() { returns 3 }}` — «Тип `'number'` не может быть присвоен типу `'string'`».

В общей сложности выходит четыре способа объявления объектов в TypeScript:

1. Объявление объектного литерала (вроде `{a: string}`), называемого также формой. Используйте ее, когда знаете, какие поля будет иметь объект, или когда все его значения будут иметь один тип.
2. Объявление пустого объектного литерала (`{}`). Старайтесь его избегать.
3. Тип `object`. Используйте его, когда вам просто нужен объект и неважно, какие у него поля.
4. Тип `Object`. Старайтесь его избегать.

Используйте линтер для предупреждения появления второго и четвертого вариантов, отмечайте их непригодность в обзорах кода, печатайте плакаты, то есть используйте предпочтительный для вашей команды инструмент, чтобы держать их подальше от базы кода.

Таблица 3.1 представляет удобную справку относительно способов 2–4 предыдущего списка.

**Таблица 3.1.** Является ли значение допустимым объектом

| Value | {} | object | Object |
|---|---|---|---|
| `{}` | Да | Да | Да |
| `['a']` | Да | Да | Да |
| `function () {}` | Да | Да | Да |
| `new String('a')` | Да | Да | Да |
| `new String('a')` | Да | Да | Да |
| `'a'` | Да | Нет | Да |
| `1` | Да | Нет | Да |
| `Symbol('a')` | Да | Нет | Да |
| `null` | Нет | Нет | Нет |
| `undefined` | Нет | Нет | Нет |

## Перерыв: псевдонимы, объединения и пересечения типов

Вы быстро осваиваете TypeScript: изучили несколько видов типов и принципов их работы, ознакомились с концепциями системы типов и типобезопасности. Пришло время погружаться глубже.

Как вы уже знаете, если есть значение, то в зависимости от его типа с ним можно производить определенные операции. Например, использовать `+` для сложения двух чисел или `.toUpperCase` для перевода строки в верхний регистр.

С самим типом тоже можно производить операции.

### Псевдонимы типов

Подобно тому как вы используете декларации (`let`, `const` и `var`) для объявления переменной, выступающей псевдонимом значения, вы также можете объявлять псевдоним типа, указывающий на тип. Выглядит это так:

```
type Age = number

type Person = {
    name: string
    age: Age
}
```

`Age` может быть только `number`. Это также помогает облегчить понимание определения формы `Person`. TypeScript не делает вывод псевдонимов, поэтому их нужно объявлять явно:

```
let age: Age = 55

let driver: Person = {
    name: 'James May'
    age: age
}
```

Поскольку `Age` является псевдонимом `number`, значит, он совместим с `number` и можно переписать код так:

```
let age = 55

let driver: Person = {
```

```
    name: 'James May'
    age: age
}
```

Псевдоним типа всегда можно заменить типом, на который он указывает.

Как и в случае с объявлением переменных (`let`, `const` и `var`), объявить тип дважды нельзя:

```
type Color = 'red'
type Color = 'blue'   // Ошибка TS2300: повтор идентификатора 'Color'.
```

Так же как для `let` и `const`, диапазон псевдонимов типов ограничен одним блоком. Каждый блок и каждая функция имеют свой диапазон, и внутренние объявления псевдонимов типов перекрывают внешние:

```
type Color = 'red'

let x = Math.random() < .5

if (x) {
    type Color = 'blue' // Здесь перекрывается Color, объявленный выше.
    let b: Color = 'blue'
} else {
    let c: Color = 'red'
}
```

Псевдонимы типов нужны для соблюдения техники DRY (не повторяйтесь)[1] при работе со сложными повторяющимися типами, а также для прояснения задачи переменной (некоторые программисты предпочитают использовать описательные имена типов вместо описательных имен переменных). Когда вы решаете, применять или нет псевдоним типа, пользуйтесь теми же доводами, что помогают решить, выделять или нет значение в отдельную переменную.

#### Типы объединения и пересечения

Объединением *A* и *B* будет их сумма (все, что есть в *A*, или в *B*, или в обоих), пересечение же — это то, что у них есть общего (все, что есть и в *A*,

[1] Код не должен быть повторяющимся. Концепция DRY (Don't Repeat Yourself) была представлена Эндрю Хантом и Дэвидом Томасом в книге «Программист-прагматик: путь от подмастерья к мастеру» (СПб.: Питер, 2007).

и в *B*). Представим это как наборы (рис. 3.2). Слева изображено объединение (сумма) двух наборов, а справа — их пересечение (произведение).

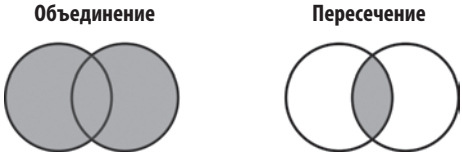


**Рис. 3.2.** Объединение (|) и пересечение (&)

TypeScript предоставляет специальные операторы типов для описания объединений и пересечений: | для объединения и & для пересечения. Поскольку типы во многом схожи с наборами, мы можем воспринимать их как наборы:

```
type Cat = {name: string, purrs: boolean}
type Dog = {name: string, barks: boolean, wags: boolean}
type CatOrDogOrBoth = Cat | Dog
type CatAndDog = Cat & Dog
```

Что известно о `CatOrDogOrBoth`? Мы видим только свойство `name`, являющееся `string`. А что можно присвоить `CatOrDogOrBoth`? Пожалуй, `Cat`, `Dog` или и то и другое.

```
// Cat
let a: CatOrDogOrBoth = {
  name: 'Bonkers',
  purrs: true
}

// Dog
a = {
  name: 'Domino',
  barks: true,
  wags: true
}

// И то и другое
a = {
  name: 'Donkers',
  barks: true,
```

```
  purrs: true,
  wags: true
}
```

Повторный перебор вполне уместен, поскольку значение с типом объединения (`|`) может относиться не к одному из членов объединения, а к обоим членам сразу[1].

С другой стороны, что вы знаете о `CatAndDog`? Ведь гибридный суперпитомец умеет мурлыкать, лаять и вилять хвостом:

```
let b: CatAndDog = {
    name: 'Domino',
    barks: true,
    purrs: true,
    wags: true
}
```

Объединения, как правило, встречаются чаще, чем пересечения. Рассмотрим следующую функцию:

```
function trueOrNull(isTrue: boolean) {
    if (isTrue) {
        return 'true'
    }
    return null
}
```

Каков будет тип возвращаемого ей значения? Это может быть `string` или `null`. То есть можно выразить возвращаемый тип так:

```
type Returns = string | null
```

А как насчет следующего примера?

```
function(a: string, b: number) {
    return a || b
}
```

Если `a` окажется верно, то возвращаемый тип будет `string`, в противном случае он будет `number`. Иначе говоря, он `string | number`.

[1] Ознакомьтесь с подразделом «Типы размеченного объединения» на с. 162, чтобы понять, каким образом можно подсказать TypeScript, что объединение не является связанным и его значение должно иметь один из двух типов, но не оба одновременно.

Последний естественный случай появления объединений — это массивы (разнородный их вид).

## Массивы

Как и в JavaScript, в TypeScript массивы являются особыми объектами, поддерживающими конкатенацию, передачу, поиск и срезы. Вот пример:

```
let a = [1, 2, 3]                  // number[]
var b = ['a', 'b']                 // string[]
let c: string[] = ['a']            // string[]
let d = [1, 'a']                   // (string | number)[]
const e = [2, 'b']                 // (string | number)[]

let f = ['red']
f.push('blue')
f.push(true)                       // Ошибка TS2345: аргумент типа
                                   // 'true' не может быть присвоен
                                   // параметру типа 'string'.

let g = []                         // any[]
g.push(1)                          // number[]
g.push('red')                      // (string | number)[]

let h: number[] = []               // number[]
h.push(1)                          // number[]
h.push('red')                      // Ошибка TS2345: аргумент типа
                                   // '"red"' не может быть присвоен
                                   // параметру типа 'number'.
```

TypeScript поддерживает два варианта синтаксиса для массивов: `T[]` и `Array<T>`. Они идентичны по значению и действию. В этой книге применяется вариант `T[]`, поскольку он более сжат, но вам стоит самостоятельно решить, какой из них подходит больше для вашего кода.

По мере ознакомления с примерами обратите внимание, что все, кроме `c` и `h`, типизировано неявно и есть правила относительно того, что можно, а что нельзя помещать в массив.

Главное правило гласит, что массивы должны сохранять однородность — не стоит смешивать яблоки с апельсинами и числами в одном массиве, чтобы не пришлось лишний раз подтверждать безопасность своих действий.

Понять, почему все гораздо проще, если массив однороден, можно, взглянув на пример `f`. Я инициализировал массив со строкой `'red'` (на момент объявления массив содержал только строки, и TypeScript сделал вывод, что это массив строк). Затем я передал в него `'blue'`, являющееся строкой, и TypeScript позволил это. Затем я попробовал передать в массив `true`, и попытка провалилась.

С другой стороны, когда я инициализировал `d`, то определил в нем `number` и `string`, и TypeScript сделал вывод, что массив имеет тип `number | string`. Поскольку каждый элемент может быть либо числом, либо строкой, перед его использованием нужно проверять, чем он является. Например, вы хотите сделать отображение этого массива, преобразовав каждую букву в верхний регистр и утроив каждое число:

```
let d = [1, 'a']

d.map(_ => {
    if (typeof _ === 'number') {
        return _ * 3
    }
    return _.toUpperCase()
})
```

Прежде чем использовать любой из элементов, нужно запросить его тип посредством `typeof`.

Как и с объектами, создание массива с `const` не побудит TypeScript делать более узкий вывод типа. Вот почему он вывел, что и `d`, и `e` имеют тип `number | string`.

Вариант с `g` особенный. Когда вы инициализируете пустой массив, TypeScript не знает, какой тип будут иметь его элементы, и присваивает ему тип `any`. По мере добавления в массив новых значений TypeScript постепенно определяет его тип в соответствии с ними. Как только массив выйдет за определенный диапазон (например, если вы объявили его в функции, а затем вернули), тогда TypeScript присвоит ему последний тип, который не может быть расширен далее:

```
function buildArray() {
    let a = []                              // any[]
    a.push(1)                               // number[]
    a.push('x')                             // (string | number)[]
    return a
}

let myArray = buildArray()      // (string | number)[]
myArray.push(true)              // Ошибка 2345: аргумент типа 'true'
                                // не может быть присвоен параметру
                                // типа 'string | number'.
```

До тех пор пока использование `any` проходит, можете не переживать.

## Кортежи

Кортежи являются подтипами `array`. Они позволяют типизировать массивы фиксированной длины, в которых значения каждого индекса имеют конкретные известные типы. В отличие от большинства других типов, кортежи в момент их объявления должны типизироваться явно. Это вызвано тем, что в JavaScript для кортежей и массивов используется одинаковый синтаксис (квадратные скобки), а в TypeScript уже есть правила вывода типов из квадратных скобок.

```
let a: [number] = [1]

// Кортеж [имя, фамилия, год рождения]
let b: [string, string, number] = ['malcolm', 'gladwell', 1963]

b = ['queen', 'elizabeth', 'ii', 1926] // Ошибка TS2322: тип
// 'string' не может быть присвоен типу 'number'.
```

Кортежи также поддерживают опциональные элементы. Как и для типов `object`, опциональность обозначается знаком `?`:

```
// Массив железнодорожных тарифов, который может меняться
// в зависимости от направления
let trainFares: [number, number?][] = [
    [3.75],
    [8.25, 7.70],
    [10.50]
]
```

```
// Эквивалент:
let moreTrainFares: ([number] | [number, number])[] = [
    // ...
]
```

Кортежи также поддерживают оставшиеся элементы, которые вы можете использовать для типизации кортежей минимальной длины:

```
// Список строк с как минимум одним элементом
let friends: [string, ...string[]] = ['Sara', 'Tali', 'Chloe', 'Claire']

// Разнородный список
let list: [number, boolean, ...string[]] = [1, false, 'a', 'b', 'c']
```

Кортежные типы не только гарантируют типобезопасность для разнородных списков, но и захватывают длину типизируемого ими списка. Эти особенности позволяют картежам превосходить обычные массивы по уровню безопасности типов. Используйте кортежи чаще.

#### Массивы и кортежи только для чтения

Обычные массивы являются изменяемыми (можно добавлять в них (`.push`), удалять из них или вставлять в них (`.splice`) и обновлять их на месте), что, вероятно, и понадобится. Однако иногда нужен именно неизменяемый массив — тот, при обновлении которого формируется новый массив.

В TypeScript по умолчанию имеется тип массива `readonly`, который можно использовать для создания неизменяемых массивов. Массивы только для чтения похожи на обычные, но их нельзя обновить на месте. Чтобы создать такой массив, используйте неизменяющие методы вроде `.concat` и `.slice` вместо изменяющих — `.push` или `.splice`:

```
let as: readonly number[] = [1, 2, 3]      // readonly number[]
let bs: readonly number[] = as.concat(4)   // readonly number[]
let three = bs[2]                          // number
as[4] = 5                                  // Ошибка TS2542:
                                           // сигнатура индекса
                                           // в типе 'readonly
                                           // number[]' допускает
                                           // только чтение.
as.push(6)                                 // Ошибка TS2339: свойство
                                           // 'push' не существует
                                           // в типе 'readonly
                                           // number[]'.
```

Как и в случае с `Array`, в TypeScript есть пара более длинных форм для объявления массивов и кортежей только для чтения:

```
type A = readonly string[]              // readonly string[]
type B = ReadonlyArray<string>          // readonly string[]
type C = Readonly<string[]>             // readonly string[]

type D = readonly [number, string]      // readonly [number, string]
type E = Readonly<[number, string]>     // readonly [number, string]
```

Какой именно синтаксис использовать — более краткий модификатор readonly или более длинную форму Readonly или ReadonlyArray, — это дело вкуса.

Обратите внимание, что массивы только для чтения могут облегчить восприятие кода, но они подкреплены обычными массивами JavaScript, поэтому даже для небольшого изменения массива необходимо сначала копировать его оригинал, что может при неосторожности навредить производительности приложения. Подобные издержки незаметны в случаях с небольшими массивами, но становятся весьма ощутимы при работе с крупными.

Если вы собираетесь активно использовать неизменяемые массивы, рассмотрите применение более эффективных реализаций наподобие immutable Ли Байрона.

## null, undefined, void и never

В JavaScript есть два значения для выражения отсутствия: `null` и `undefined`. TypeScript поддерживает их оба и имеет для них типы. Есть предположения, как они называются? Да, `null` и `undefined`.

В TypeScript тип `undefined` может иметь только одно значение — `undefined`, то же касается и типа `null`, который описывает значение `null`.

JavaScript-программисты обычно используют их как взаимозаменяемые, хотя между ними и есть тонкое семантическое различие, достойное упоминания: `undefined` означает, что нечто еще не было определено, а `null` показывает отсутствие значения (как если бы вы пытались вычислить значение, но сталкивались при этом с ошибкой). Это всего лишь услов-

ности, и TypeScript вас не принуждает к их соблюдению, однако будет полезным понимать разницу.

Типы `void` и `never` проводят различие между несуществующими вещами: `void` — это возвращаемый тип функции, которая не возвращает ничего явно (например, console.log), а `never` — это тип функции, которая никогда ничего не возвращает (выбрасывает исключение или выполняется бесконечно):

```
// (a) Функция, возвращающая число или null
function a(x: number) {
    if (x < 10) {
        return x
    }
    return null
}

// (b) Функция, возвращающая undefined
function b() {
    return undefined
}

// (c) Функция, возвращающая void
function c() {
    let a = 2 + 2
    let b = a * a
}

// (d) Функция, возвращающая never
function d() {
    throw TypeError('I always error')
}

// (e) Другая функция, возвращающая never
function e() {
    while (true)
        { doSomething()
    }
}
```

(a) и (b) явно возвращают `null` и `undefined` соответственно. (c) возвращает `undefined`, но не делает этого при явной инструкции `return`, поэтому мы

говорим, что она возвращает `void`. (`d`) выбрасывает исключение, а (`e`) выполняется вечно — обе они никогда ничего не вернут, поэтому их возвращаемый тип — `never`.

### СТРОГАЯ ПРОВЕРКА НА NULL

В более старых версиях TypeScript (или при опции `strictNullChecks`, установленной как `false`) `null` ведет себя необычно: он является подтипом всех типов, кроме `never`. Это значит, что каждый тип может быть `null` и вы не можете быть ни в чем уверены, пока не произведете проверку на `null`. Например, если кто-нибудь передает вашей функции переменную `pizza` и вы хотите вызвать для нее метод `.addAnchovies` (добавить анчоус), то сначала вы должны проверить, не является ли `pizza null`, и только затем добавлять в нее рыбку. На практике утомительно делать это для каждой переменной, поэтому о предварительной проверке часто забывают. В таких случаях, когда что-либо действительно оказывается `null`, при выполнении вы получаете зловещее исключение нулевого указателя:

```
function addDeliciousFish(pizza: Pizza) {
    return pizza.addAnchovies()  // Не перехваченная ошибка типа:
}                                // невозможно прочитать свойство
                                // 'addAnchovies', являющееся null

// TypeScript допускает подобное при strictNullChecks = false
addDeliciousFish(null)
```

`null` был назван «ошибкой на триллион долларов» (https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/) тем, кто впервые обозначил его в 1960 году. Проблема в том, что его невозможно выразить и проверить в системах типов большинства языков. Поэтому, когда программисты пытаются сделать что-либо с переменной, которую они считают определенной, но которая при выполнении оказывается `null`, код выбрасывает исключение среды выполнения.

Почему? Откуда я знаю. Но языки находят способы кодирования `null` в их системах типов, и TypeScript является отличным примером того, как это делать правильно. Если цель в обнаружении как можно большего количества багов в процессе компиляции, прежде чем с ними столкнутся пользователи, тогда возможность проверки на `null` просто необходима в системе типов.

Если `unknown` — это супертип любого другого типа, то `never` — это подтип любых других типов, или *низший тип* (bottom type), который может быть присвоен любому другому типу, и значение типа `never` может быть использовано везде безопасно. Это все теория[1], но пригодится в диалогах о TypeScript с другими знатоками языков.

Таблица 3.2 подытоживает принципы использования типов отсутствия.

**Таблица 3.2.** Типы, означающие отсутствие чего-либо

| Type | Meaning |
|---|---|
| `null` | Отсутствие значения |
| `undefined` | Переменная, которой не присвоено значение |
| `void` | Функция, не имеющая оператора `return` |
| `never` | Функция, никогда ничего не возвращающая |

## Enum

`Enum` является способом перечисления возможных значений типа. Он представляет собой неупорядоченную структуру данных, которая сопоставляет ключи и значения. Воспринимайте эти значения как объекты, имеющие во время компиляции фиксированные ключи, что позволяет TypeScript убедиться, что данный ключ будет существовать при обращении к значению.

Есть два типа `enum`: отображающий строки в строки и отображающий строки в числа:

```
enum Language {
    English,
    Spanish,
    Russian
}
```

TypeScript будет автоматически выводить число в качестве значения для каждого члена перечисления, но вы также можете установить значения явно. Давайте сделаем явным вывод из предыдущего примера:

[1] Можно думать о низшем типе как о типе, который не имеет значений. Низший тип соответствует математическому предложению, которое всегда ложно.

```
enum Language {
    English = 0,
    Spanish = 1,
    Russian = 2
}
```

Общепринято, что имена `enum` начинаются с верхнего регистра и имеют форму единственного числа. Его ключи также начинаются с верхнего регистра.

Чтобы извлечь значение из перечисления, обратитесь к нему через запись с точкой или скобкой — так же, как если бы вы получали значение из обычного объекта:

```
let myFirstLanguage = Language.Russian        // Language
let mySecondLanguage = Language['English']    // Language
```

Можно разделить перечисление на несколько деклараций, и TypeScript автоматически произведет их слияние (см. подраздел «Слияние деклараций» на с. 122). Будьте осторожны при разделении перечисления, TypeScript может вывести значения только для одной из деклараций, поэтому лучше всего в таком случае явно присвоить значение каждому члену перечисления:

```
enum Language {
    English = 0,
    Spanish = 1
}

enum Language {
    Russian = 2
}
```

Можете использовать посчитанные значения, и вам не обязательно самстоятельно их определять:

```
enum Language {
    English = 100,
    Spanish = 200 + 300,
    Russian // TypeScript выводит 501 (число, следующее за 500)
}
```

Также вы можете использовать строчные значения для нумерации или даже смешивать строчные и числовые значения:

```
enum Color {
  Red = '#c10000',
  Blue = '#007ac1',
  Pink = 0xc10050,    // Шестнадцатеричный литерал
  White = 255         // Десятичный литерал
}

let red = Color.Red    // Color
let pink = Color.Pink  // Color
```

TypeScript для удобства позволяет обращаться к перечислениям как по значению, так и по ключу, но это легко может нарушить безопасность:

```
let a = Color.Red      // Color
let b = Color.Green    // Ошибка TS2339: свойство 'Green'
                       // не существует в типе 'typeof Color'.
let c = Color[0]       // string
let d = Color[6]       // string (!!!)
```

У вас не должно быть возможности получить `Color[6]`, но TypeScript вас не останавливает. Мы можем попросить его предотвращать подобное небезопасное обращение с помощью `const enum`. Давайте перепишем последнее перечисление `Language`:

```
const enum Language {
  English,
  Spanish,
  Russian
}

// Обращение к верному ключу перечисления
let a = Language.English  // Language

// Обращение к неверному ключу перечисления
let b = Language.Tagalog  // Ошибка TS2339: свойство 'Tagalog'
                          // не существует в типе 'typeof Language'.

// Обращение к верному значению перечисления
let c = Language[0]    // Ошибка TS2476: обратиться к константному
                       // члену перечисления можно только с помощью
                       // строчного литерала.
```

```
// Обращение к неверному значению перечисления
let d = Language[6]     // Ошибка TS2476: обратиться к константному
                        // члену перечисления можно только
                        // с помощью строчного литерала.
```

`const enum` не позволяет производить обратный просмотр и поэтому во многом ведет себя как обычный JavaScript-объект. Он также по умолчанию не генерирует никакой JavaScript-код, а вместо этого подставляет значение члена перечисления везде, где он используется (например, TypeScript заменит `Language.Spanish` на его значение `1` везде, где он встречается).

### TSC-ФЛАГ: PRESERVECONSTENUMS

Подстановка, обусловленная `const enum`, может вызывать проблемы безопасности при импорте `const enum` из чужого TypeScript-кода: если автор перечисления обновит свой `const enum` после того, как вы скомпилируете свой код, то ваши с ним версии могут указывать на различные значения при выполнении, о чем TypeScript знать не будет.

Избегайте подстановок и используйте `const enums` только в контролируемых программах, которые вы не собираетесь публиковать в NPM или делать доступными в качестве библиотек.

Чтобы активировать для `const enums` генерацию кода при выполнении, смените TSC-настройку `preserveConstEnums` на `true` в tsconfig.json:

```
{
    "compilerOptions": {
        "preserveConstEnums": true
    }
}
```

Рассмотрите использование `const enums`:

```
const enum Flippable {
    Burger,
    Chair,
    Cup,
    Skateboard,
    Table
}

function flip(f: Flippable) {
```

```
    return 'flipped it'
}

flip(Flippable.Chair)    // 'flipped it'
flip(Flippable.Cup)      // 'flipped it'
flip(12)                 // 'flipped it' (!!!)
```

Все выглядит отлично — с `Chairs` и `Cups` происходит именно то, что вы и ожидаете... пока вы не осознаете, что все числа также совместимы с перечислениями. Такое поведение является неудачным последствием правил совместимости в TypeScript. Чтобы это исправить, вам нужно быть очень осторожными и использовать только перечисления со строчными значениями:

```
const enum Flippable {
    Burger = 'Burger',
    Chair = 'Chair',
    Cup = 'Cup',
    Skateboard = 'Skateboard',
    Table = 'Table'
}

function flip(f: Flippable) {
    return 'flipped it'
}

flip(Flippable.Chair)  // 'flipped it'
flip(Flippable.Cup)    // 'flipped it'
flip(12)               // Ошибка TS2345: аргумент типа '12'
                       // не может быть присвоен параметру типа
                       // 'Flippable'.
flip('Hat')            // Ошибка TS2345: аргумент типа '"Hat"'
                       // не может быть присвоен параметру типа
                       // 'Flippable'.
```

Достаточно присутствия одного числового значения, чтобы сделать все перечисление небезопасным.

Из-за всех ловушек, которые затрудняют безопасное использование перечислений, рекомендую держаться от них подальше.

Если же на их использовании настаивают ваши коллеги и их никак нельзя переубедить, то обязательно внедрите специальные правила TSLint, предупреждающие об использовании числовых значений и неконстантных перечислений.

## Итоги

TypeScript содержит множество встроенных типов. Вы можете разрешить ему выводить типы за вас на основе значений или же явно типизировать значения. `const` позволит выводить более конкретные типы, `let` и `var` — более общие. Большинство типов имеют основные и более конкретные варианты, причем последние являются подтипами первых (табл. 3.3).

**Таблица 3.3.** Типы и их более конкретные подтипы

| Тип | Подтип |
|---|---|
| `boolean` | `Boolean literal` |
| `bigint` | `BigInt literal` |
| `number` | `Number literal` |
| `string` | `String literal` |
| `symbol` | `unique symbol` |
| `object` | `Object literal` |
| `Array` | `Tuple` |
| `enum` | `const enum` |

# Упражнения к главе 3

1. Какой тип выведет TypeScript для каждого из этих значений?

   а) `let a = 1042`

   б) `let b = 'apples and oranges'`

   в) `const c = 'pineapples'`

   г) `let d = [true, true, false]`

   д) `let e = {type: 'ficus'}`

   е) `let f = [1, false]`

   ж) `const g = [3]`

   з) `let h = null` (выполните это в редакторе, если результат вас удивит, то перейдите к подразделу «Расширение типов» на с. 155).

2. Почему каждый из этих примеров выдает ошибку?

   а)

```
let i: 3 = 3
i = 4                       // Ошибка TS2322: тип '4' не может
                            // быть присвоен типу '3'.
```

   б)

```
let j = [1, 2, 3]
j.push(4)
j.push('5')                 // Ошибка TS2345: аргумент типа '5'
                            // не может быть присвоен параметру
                            // типа 'number'.
```

   в)

```
let k: never = 4            // Ошибка TSTS2322: тип '4' не может
                            // быть присвоен типу 'never'.
```

   г)

```
let l: unknown = 4
let m = l * 2               // Ошибка TS2571: объект имеет тип
                            // 'unknown'.
```

ГЛАВА 4

# Функции

В предыдущей главе мы рассмотрели основы системы типов TypeScript: примитивные типы, объекты, массивы, кортежи и перечисления. Мы также изучили основы вывода типов и принципы их совместимости. Теперь можно переходить к главному номеру программы (а для функциональных программистов — к смыслу жизни) — к функциям. В этой главе мы раскроем следующие темы.

- Разные способы объявления и вызова функций в TypeScript.
- Перегрузка сигнатуры.
- Полиморфные функции.
- Полиморфные псевдонимы типов.

## Объявление и вызов функций

В JavaScript функции являются объектами первого уровня. Это означает, что их можно использовать как объекты: присваивать переменным, передавать другим функциям, возвращать из функций, присваивать объектам и прототипам, записывать в них свойства, считывать эти свойства и т. д. TypeScript моделирует все эти возможности с помощью своей богатой системы типов.

Так выглядит функция в TypeScript (пример вам знаком из предыдущей главы):

```
function add(a: number, b: number) {
    return a + b
}
```

Обычно параметры функции аннотируются явно (`a` и `b` в этом примере). TypeScript выводит типы в теле функции, но не для параметров, за исклю-

чением нескольких ситуаций, когда он ориентируется на контекст (подраздел «Контекстная типизация» на с. 82). Возвращаемый тип подлежит выводу, но при желании его можно аннотировать явно:

```
function add(a: number, b: number): number {
    return a + b
}
```

На протяжении книги я буду явно аннотировать возвращаемые типы там, где это поможет объяснить действие функции. В других случаях я позволю TypeScript их вывести.

В последнем примере был использован синтаксис именованной функции, но в JavaScript и TypeScript предлагается как минимум пять способов объявления функции:

```
// Именованная функция
function greet(name: string) {
    return 'hello ' + name
}

// Функциональное выражение
let greet2 = function(name: string) {
    return 'hello ' + name
}

// Выражение стрелочной функции
let greet3 = (name: string) => {
    return 'hello ' + name
}

// Сокращенное выражение стрелочной функции
let greet4 = (name: string) =>
    'hello ' + name

// Конструктор функции
let greet5 = new Function('name', 'return "hello " + name')
```

За исключением конструктора функции (который не рекомендуется использовать[1]), эти виды синтаксиса поддерживаются TypeScript в типобезопасном режиме и следуют правилам, касающимся обязательных аннотаций типов для параметров и опциональных аннотаций для возвращаемых типов.

Сверим часы.

- Параметр — это часть данных, необходимых функции для запуска, объявленная как часть декларации этой функции. Также может называться *формальным параметром*.
- Аргумент — это часть данных, предаваемая функции при ее вызове. Также может называться *актуальным параметром*.

При вызове функции в TypeScript не нужно предоставлять дополнительную информацию о типе — достаточно передать ей некий аргумент, и TypeScript проверит совместимости этого аргумента с типами параметров функции:

```
add(1, 2)                    // вычисляется как 3
greet('Crystal')             // выводится как 'hello Crystal'
```

Если вы забыли аргумент или передали аргумент неверного типа, TypeScript поспешит на это указать:

```
add(1)       // Ошибка TS2554: ожидается 2 аргумента, но получен 1.
add(1, 'a')  // Ошибка TS2345: аргумент типа '"a"' не может быть
             // присвоен параметру типа 'number'.
```

## Предустановленные и опциональные параметры

Как и для типов `object` и кортежей, можно использовать `?` для обозначения параметра в качестве опционального. При объявлении параметров функции необходимые параметры должны идти перед опциональными:

[1] Если вы введете последний пример в редактор, то увидите тип `Function`. Это вызываемый объект (если поместить за ним `()`), который имеет все методы прототипа из `Function.prototype`. Но его параметры и возвращаемый тип нетипизированы, и вы можете вызвать такую функцию с любыми аргументами, причем TypeScript даже не отреагирует на такие недопустимые действия.

```
function log(message: string, userId?: string) {
    let time = new Date().toLocaleTimeString()
    console.log(time, message, userId || 'Not signed in')
}

log('Page loaded')                  // Логируем "12:38:31 PM
                                    // Page loaded Not signed in"
log('User signed in', 'da763be')    // Логируем "12:38:31 PM
                                    // User signed in da763be"
```

Как и JavaScript, TypeScript позволяет снабдить опциональные параметры значениями по умолчанию. Семантически это схоже с назначением параметра опциональным, поскольку вызывающий элемент не обязан передавать такой параметр функции (разница в том, что параметр по умолчанию можно не ставить в конец списка в отличие от опционального).

Например, можно переписать `log` так:

```
function log(message: string, userId = 'Not signed in') {
    let time = new Date().toISOString()
    console.log(time, message, userId)
}

log('User clicked on a button', 'da763be')
log('User signed out')
```

Заметьте, что когда мы присваиваем `userId` значение по умолчанию, то удаляем его опциональную аннотацию `?`. Больше нет необходимости его типизировать. TypeScript вполне способен вывести тип параметра на основе его значения по умолчанию, сохраняя при этом краткость и читабельность кода.

Конечно, вы также можете добавлять явные аннотации типов для параметров по умолчанию тем же способом, что и для обычных параметров:

```
type Context = {
    appId?: string
    userId?: string
}

function log(message: string, context: Context = {}) {
    let time = new Date().toISOString()
    console.log(time, message, context.userId)
}
```

На практике вы будете использовать предустановленные параметры чаще, чем опциональные.

### Оставшиеся параметры

Если функция получает список аргументов, можно передать этот список в виде массива:

```
function sum(numbers: number[]): number {
    return numbers.reduce((total, n) => total + n, 0)
}

sum([1, 2, 3]) // вычисляется как 6
```

Иногда вместо API функции с фиксированной арностью вы выберете API с переменным числом аргументов. Традиционно для этого используется магический JavaScript-объект `arguments`.

Среда выполнения JavaScript автоматически определяет `arguments` в функциях и присваивает ему список переданных функции аргументов. Поскольку `arguments` является не настоящим массивом, а всего лишь его подобием, то сначала необходимо преобразовать его в массив и лишь затем вызвать для него встроенный метод `.reduce`:

```
function sumVariadic(): number {
    return Array
        .from(arguments)
        .reduce((total, n) => total + n, 0)
}

sumVariadic(1, 2, 3) // вычисляется как 6
```

Однако при использовании `arguments` возникает большая проблема: небезопасность. Если вы наведете курсор на `total` или `n` в редакторе, то увидите результат, показанный на рис. 4.1.

Это означает, что TypeScript вывел типы `n` и `total` как `any`, молча позволив их передать. На ошибку же он укажет, только когда вы попытаетесь использовать `sumVariadic`:

```
sumVariadic(1, 2, 3) // Ошибка TS2554: ожидается ноль аргументов,
                     // но получено 3.
```


**Рис. 4.1.** arguments небезопасен

Поскольку мы не объявили, что `sumVariadic` получает аргументы, TypeScript решает, что она их вообще не получает, отсюда и ошибка `TypeError`.

Как же безопасно типизировать переменные функции?

На помощь приходят оставшиеся параметры. Вместо использования небезопасного `arguments` воспользуемся оставшимися параметрами, чтобы функция `sum` принимала любое число аргументов:

```
function sumVariadicSafe(...numbers: number[]): number {
    return numbers.reduce((total, n) => total + n, 0)
}

sumVariadicSafe(1, 2, 3) // вычисляется как 6.
```

Вот так! Обратите внимание, что единственое различие между этой переменной функцией `sum` и оригинальной функцией с одним параметром — это наличие... в списке параметров, больше ничего не требуется менять. Причем такой вариант полностью безопасен.

Функция может иметь не более одного оставшегося параметра, и этот параметр должен быть последним в списке. Например, взгляните на встроенную декларацию для console.log в TypeScript (`interface` мы рассмотрим в главе 5). console.log получает опциональный аргумент `message` и любое число дополнительных аргументов для логирования:

```
interface Console {
    log(message?: any, ...optionalParams: any[]): void
}
```

## Методы call, apply и bind

В дополнение к вызову функции со скобками `()` JavaScript поддерживает по меньшей мере два других способа вызова. Возьмем `add`, использованную ранее в этой главе:

```
function add(a: number, b: number): number {
    return a + b
}

add(10, 20)                    // вычисляется как 30
add.apply(null, [10, 20])      // вычисляется как 30
add.call(null, 10, 20)         // вычисляется как 30
add.bind(null, 10, 20)()       // вычисляется как 30
```

Метод `apply` привязывает значение к `this` внутри функции (в этом примере мы привязываем к `this` `null`) и вторым аргументом объединяет параметры функции. Метод `call` делает то же самое, но применяет все аргументы по порядку вместо объединения.

Метод `bind` схож с ними в том, что привязывает к функции аргумент `this` и список аргументов. Разница в том, что вместо вызова старой функции `bind` возвращает новую, которую затем вы можете вызвать с `()`, `.call` или `.apply`, передавая ей при желании больше аргументов для привязки к свободным параметрам.

**TSC-ФЛАГ: STRICTBINDCALLAPPLY**

Для безопасного использования `.call`, `.apply` и `.bind`, активируйте опцию `strictBindCallApply` в tsconfig.json (автоматически активируется при включении режима `strict`).

## Типизация this

Если вы ранее не работали с JavaScript, то вас может удивить, что здесь переменная `this` определяется для каждой функции, а не только для тех, которые существуют в качестве методов в классах. Значения `this` отличаются в зависимости от того, как вы вызываете функцию, что может сделать ее заведомо нестабильной и трудной для понимания.

По этой причине многие команды разработчиков запрещают `this` везде, кроме методов классов. Чтобы сделать то же самое в вашей базе кода, добавьте в правила *TSLint* `no-invalid-this`.

Причина нестабильности `this` связана со способом ее присваивания. `this` при вызове метода принимает значение слева от точки. Например:

```
let x = {
    a() {
        return this
    }
}
x.a()            // this является объектом x в теле a()
```

Но если в какой-то момент вы переназначите `a`, прежде чем вызвать ее, результат изменится:

```
let a = x.a
a()              // теперь this является undefined в теле a()
```

Представьте сервисную функцию для форматирования дат:

```
function fancyDate() {
    return `${this.getDate()}/${this.getMonth}/${this.getFullYear()}`
}
```

Вероятно, этот API разработан начинающим программистом (до того как он узнал о параметрах функции). Чтобы использовать `fancyDate`, нужно вызвать ее с `Date`, привязанной к `this`:

```
fancyDate.call(new Date)  // выводится как "4/14/2005"
```

Если вы забудете привязать `Date` к `this`, то получите исключение при выполнении:

```
fancyDate()              // Неперехваченная ошибка типа:
                         // this.getDate не является функцией
```

Хотя изучение всех семантических особенностей `this` выходит за рамки данной книги[1], такое поведение, при котором `this` зависит от того, как вы вызвали, а не объявили функцию, примечательно.

К счастью, TypeScript решает эту проблему. Если ваша функция использует `this`, обязательно объявите ожидаемый тип `this` в качестве первого

[1] Чтобы глубоко вникнуть в теорию `this`, прочтите серию книг «Вы не знаете JS» Кайла Симпсона (СПб.: Питер, 2019).

параметра функции (до любых дополнительных параметров), и TypeScript проследит, чтобы `this` в каждом месте вызова оказалась тем, чем вы ее назначили. `this` не рассматривается как остальные параметры — это резервное слово, используемое как часть сигнатуры функции:

```
function fancyDate(this: Date) {
    return `${this.getDate()}/${this.getMonth()}/${this.getFullYear()}`
}
```

Вот что происходит теперь при вызове `fancyDate`:

```
fancyDate.call(new Date)      // выводится как "6/13/2008"

fancyDate()                   // Ошибка TS2684: контекст 'this'
                              // типа 'void' не может быть присвоен
                              // this метода, имеющему тип 'Date'.
```

Мы взяли ошибку среды выполнения и предоставили TypeScript достаточно информации, чтобы он впредь выводил предупреждение во время компиляции.

**TSC-ФЛАГ: NOIMPLICITTHIS**

Чтобы типы `this` всегда оказывались явно аннотированными в функциях, активируйте настройку `noImolicitThis` в tsconfig.json. Если вы уже включили режим `strict`, эта функция также будет активна.

Заметьте, что `noImplicitThis` не действует в аннотациях `this` для классов или функций в объектах.

## Функции-генераторы

Функции-генераторы (или просто генераторы) являются удобным способом генерировать набор значений. Они дают пользователю точный контроль над темпом создания значений. Ленивые генераторы вычисляют следующее значение, только когда пользователь об этом просит. Генераторы способны делать то, что иным способом сделать весьма трудно, например генерируют бесконечные списки.

Работают они так:

```
function* createFibonacciGenerator() { ❶
    let a = 0
    let b = 1
    while (true) { ❷
        yield a; ❸
        [a, b] = [b, a + b] ❹
    }
}

let fibonacciGenerator = createFibonacciGenerator()
// IterableIterator<number>
fibonacciGenerator.next()          // вычисляется как {значение: 0,
                                   // выполнено: false}
fibonacciGenerator.next()          // вычисляется как {значение: 1,
                                   // выполнено: false}
fibonacciGenerator.next()          // вычисляется как {значение: 1,
                                   // выполнено: false}
fibonacciGenerator.next()          // вычисляется как {значение: 2,
                                   // выполнено: false}
fibonacciGenerator.next()          // вычисляется как {значение: 3,
                                   // выполнено: false}
fibonacciGenerator.next()          // вычисляется как {значение: 5,
                                   // выполнено: false}
```

❶ Значок * перед именем функции делает функцию генератором. Вызов генератора производит `IterableIterator` (итерируемый итератор).

❷ Этот генератор может генерировать значения бесконечно.

❸ Генераторы используют ключевое слово `yield` для выдачи значений. Когда пользователь запросит у генератора следующее значение (например, вызвав `next`), `yield` отправит результат обратно пользователю и приостановит выполнение до момента запроса следующего значения. В этом случае цикл `while(true)` не приводит программу к бесконечному выполнению и сбою.

❹ Для вычисления следующего числа Фибоначчи перепишем `a` на `b` и `b` на `a + b` в один шаг.

Мы вызвали `createFibonacciGenerator`, и он превратился в `IterableIterator`. При каждом вызове `next` итератор вычисляет следующее число Фибо-

наччи и `yield` возвращает его нам. Обратите внимание, что TypeScript способен вывести тип итератора исходя из типа значения, которое мы запросили.

Также можно явно аннотировать генератор, обернув запрашиваемый им в `IterableIterator` тип:

```
function* createNumbers(): IterableIterator<number> {
    let n = 0
    while (1) {
        yield n++
    }
}

let numbers = createNumbers()
numbers.next()     // вычисляется как {значение: 0, выполнено: false}
numbers.next()     // вычисляется как {значение: 1, выполнено: false}
numbers.next()     // вычисляется как {значение: 2, выполнено: false}
```

В этой книге мы не станем углубляться в обширную тему генераторов, связанных с JavaScript. Кратко они представляют суперкрутую возможность, которая также поддерживается в TypeScript. Более подробно о генераторах можете прочитать в соответствующем разделе на сайте MDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*).

## Итераторы

Итераторы являются обратной стороной генераторов. Если генераторы — это способ производить поток значений, то итераторы отвечают за потребление этих значений. Начнем с пары определений.

**ИТЕРИРУЕМЫЙ**

Любой объект, содержащий свойство `Symbol.iterable`, чье значение является функцией, возвращающей итератор.

**ИТЕРАТОР**

Любой объект, который определяет метод `next`, возвращающий объект со свойствами `value` и `done`.

Когда вы создаете генератор (например, вызывая `createFibonacciGenerator`), вы получаете назад значение, являющееся итерируемым итератором, поскольку оно определяет и свойство `Symbol.iterator`, и метод `next`.

Можно вручную определить итератор или итерируемый объект, создав объект или класс, который реализует `Symbol.iterator` или `next` соответственно. В качестве примера определим итератор, возвращающий числа от `1` до `10`:

```
let numbers = {
    *[Symbol.iterator]() {
        for (let n = 1; n <= 10; n++) {
            yield n
        }
    }
}
```

Если вы введете этот итератор в редакторе и наведете на него курсор, то увидите, какой тип для него выводит TypeScript (рис. 4.2).

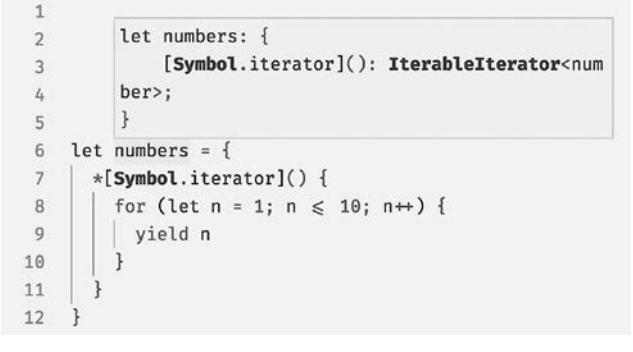


**Рис. 4.2.** Ручное определение итератора

То есть `numbers` является итерируемым объектом и вызов генератора `numbers[Symbol.iterator]()` возвращает итерируемый итератор.

Вы можете не только самостоятельно определять итераторы, но и использовать встроенные в JavaScript итераторы для обычных типов коллекций — `Array`, `Map`, `Set`, `String`[1] и т. д., чтобы, например:

[1] Примечательно, что `Object` и `Number` не являются итераторами.

```
// Производить итерирование по итератору с помощью for-of
for (let a of numbers) {

// 1, 2, 3 и т.д.
}

// Распространить итератор
let allNumbers = [...numbers] // number[]

// Деструктурировать итератор
let [one, two, ...rest] = numbers // [number, number, number[]]
```

И здесь мы также не станем уходить вглубь темы. Вы можете узнать больше об итераторах и асинхронных итераторах на сайте MDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_Generators).

**TSC-ФЛАГ: DOWNLEVELITERATION**

Если вы компилируете TypeScript-код в версию JavaScript старше чем ES2015, то можете активировать пользовательские итераторы флагом `downLevelIteration` в tsconfig.json.

Или оставить выключенной эту функцию, если ваше приложение особо чувствительно к размеру пакета, а для функционирования пользовательских итераторов в старых версиях требуется много кода. Например, предыдущий пример с `numbers` генерирует примерно 1 Кб кода (сжатого *gzip*).

## Сигнатуры вызовов

Итак, вы научились типизировать параметры и возвращаемые типы функций. Теперь разберемся, как можно выразить полные типы самих функций.

Вернемся еще раз к `sum` из начала главы. Напомню, что выглядит она так:

```
function sum(a: number, b: number): number {
    return a + b
}
```

**КОДЫ УРОВНЯ ТИПОВ И УРОВНЯ ЗНАЧЕНИЙ**

Выражения «уровень типов» и «уровень значений» часто можно услышать в контексте программирования со статическими типами.

В этой книге под выражением «код уровня типов» я подразумеваю код, состоящий только из типов и операторов типов. В противоположность ему выступает код уровня значений. Общий принцип здесь таков: рабочий JavaScript-код относится к уровню значений, а рабочий код TypeScript, но при этом нерабочий JavaScript — к уровню типов[1].

В следующем примере жирным шрифтом выделены выражения, которые относятся к уровню типов, а все остальное относятся к уровню значений:

```
function area(radius: number): number | null {
    if (radius < 0) {
        return null
    }
    return Math.PI * (radius ** 2)
}

let r: number = 3
let a = area(r)
if (a !== null) {
    console.info('result:', a)
}
```

Выражения уровня типов являются аннотациями типов с оператором типа объединения |. Все остальные выражения относятся к уровню значений.

Каков тип `sum`? Что ж, `sum` — это функция, следовательно, ее тип:

```
Function
```

Подобно тому как `object` описывает все объекты, `Function` является типом, обобщающим все функции, который не сообщает о конкретной функции ничего.

[1] Исключениями из этого принципа являются перечисления, классы и пространства имен. Перечисления и классы генерируют и типы, и значения, а пространства имен существуют только на уровне значений. Полная справка содержится в приложении B.

Как еще можно типизировать `sum`? `sum` — это функция, получающая два значения и возвращающая `number`. Выразим ее тип так:

```
(a: number, b: number) => number
```

В TypeScript такой синтаксис используется для обозначения типа функции. Иначе он называется сигнатурой вызова (или типа). Он похож на стрелочную функцию — так и задумано. Вы будете использовать этот синтаксис для типизации функций при их передаче в виде аргументов или при их возвращении из других функций.

Имена параметров `a` и `b` просто служат для документации и не влияют на совместимость функции с типом.

Сигнатуры вызовов функций содержат только код уровня типов, то есть только типы, без значений. Так они могут выражать типы параметров (в том числе опциональных), типы `this` (см. подраздел «Типизация this» на с. 73), возвращаемые типы, оставшиеся типы, но не могут выражать значения по умолчанию (поскольку это не типы). И так как у них нет тела, по которому TypeScript мог бы сделать вывод, они требуют явного аннотирования возвращаемых типов.

Рассмотрим несколько функций из этой главы и извлечем их типы в виде отдельных сигнатур вызовов, которые привяжем к псевдонимам типов:

```
// функция greet(name: string)
type Greet = (name: string) => string

// функция log(message: string, userId?: string)
type Log = (message: string, userId?: string) => void

// функция sumVariadicSafe(...numbers: number[]): number
type SumVariadicSafe = (...numbers: number[]) => number
```

Приловчились? Сигнатуры вызовов функций похожи на свои реализации. Такой дизайн был задуман для облегчения их восприятия.

Теперь конкретизируем связи между сигнатурами вызовов и их реализациями. Если у вас есть сигнатура вызова, как вы можете объявить реализующую ее функцию? Просто объединив эту сигнатуру с выражением

функции, которое ее реализует. В качестве примера перепишем Log, чтобы использовать ее свежеиспеченную сигнатуру:

```
type Log = (message: string, userId?: string) => void

let log: Log = (❶
    message, ❷
    userId = 'Not signed in' ❸
) => { ❹
    let time = new Date().toISOString()
    console.log(time, message, userId)
}
```

❶ Объявляем выражение функции log и явно присваиваем ему тип Log,

❷ Не нужно аннотировать параметры дважды: message уже аннотирован как string в определении Log, позволяем TypeScript сделать вывод его типа на основе Log.

❸ Добавляем значение по умолчанию для userId, поскольку тип userId мы изъяли из сигнатуры Log, который является типом и не может содержать значения.

❹ Не нужно повторно аннотировать возвращаемый тип — мы уже объявили его как void в Log.

## Контекстная типизация

Обратите внимание, что последний пример оказался первым примером из пройденных, где нам не пришлось явно аннотировать типы параметров функции. Ранее мы объявили, что log имеет тип Log, и TypeScript смог исходя из контекста вывести, что message — string. Эта мощная особенность системы вывода типов в TypeScript называется контекстной типизацией.

В этой главе мы отметили еще одну ситуацию для контекстной типизации: функции обратных вызовов[1].

Давайте объявим функцию times, вызывающую n раз свой обратный вызов f, передавая каждый раз в f текущий индекс:

[1] Обратный вызов — это функция, которая передана другой функции в виде аргумента.

```
function times(
    f: (index: number) => void,
    n: number
)   {
    for (let i = 0; i < n; i++) {
        f(i)
    }
}
```

При вызове `times` не нужно явно аннотировать функцию, передаваемую `times`, если она объявлена встроенной:

```
times(n => console.log(n), 4)
```

TypeScript на основе контекста выведет, что `n` является `number`, потому что мы объявили в сигнатуре `times`, что аргумент `index`, принадлежащий `f`, является `number`. TypeScript же понимает, что `n` является тем аргументом, следовательно, он должен быть `number`.

Заметьте, что, если бы мы не объявили `f` встроенной, TypeScript не смог бы вывести ее тип:

```
function f(n) { // Ошибка TS7006: параметр 'n' неявно имеет тип 'any'.
    console.log(n)
}

times(f, 4)
```

## Типы перегруженных функций

Использованный в предыдущем разделе синтаксис типизации функции — `type Fn = (…) => …` — является *сокращением сигнатуры вызова*. Можно написать его более явно. Еще раз возьмем пример с `Log`:

```
// Сокращенная сигнатура вызова
type Log = (message: string, userId?: string) => void

// Полная сигнатура вызова
type Log = {
    (message: string, userId?: string): void
}
```

Оба варианта во всех случаях будут взаимными эквивалентами, разница останется только в синтаксисе.

Вы хотели бы использовать вместо сокращенного варианта полный? Для простых случаев вроде нашей функции `Log` стоит предпочесть сокращение, но для более сложных функций есть несколько существенных причин использовать полную сигнатуру.

Первая из них — это *перегрузка* типа функции. Но сначала разберемся, что значит перегрузка функции.

> **ПЕРЕГРУЖЕННАЯ ФУНКЦИЯ**
>
> Функция с несколькими сигнатурами вызовов.

В большинстве языков программирования, как только вы объявляете функцию, получающую некий набор параметров и дающую некий возвращаемый тип, вы можете вызвать эту функцию с точно таким же набором параметров и получать точно такой же возвращаемый тип. Но не в JavaScript. Поскольку JavaScript — динамический язык, в нем есть несколько способов вызвать данную функцию. Причем иногда тип вывода будет зависеть от типа вводного аргумента.

TypeScript моделирует эту динамику — декларации перегруженных функций и зависимость типа вывода функции от типа ввода — при помощи своей статической системы типов. Кто-то воспринимает такую особенность как должное, но это настоящий прорыв.

Вы можете использовать перегруженные сигнатуры функций для проектирования самых выразительных API. Например, спроектируем API для бронирования отпуска — назовем его `Reserve`. Начнем с зарисовки его типов (с полной сигнатурой):

```
type Reserve = {
    (from: Date, to: Date, destination: string): Reservation
}
```

Теперь подытожим реализацию `Reserve`:

```
let reserve: Reserve = (from, to, destination) => {
    // ...
}
```

Итак, пользователь, желающий забронировать путешествие на Бали, должен вызвать API `reserve` с датами `from` и `to`, а также `"Bali"` в виде назначения.

Мы можем видоизменить API для поддержки рейсов в один конец:

```
type Reserve = {
    (from: Date, to: Date, destination: string): Reservation
    (from: Date, destination: string): Reservation
}
```

Вы заметите, что при попытке запустить код TypeScript выдает ошибку в месте, где вы реализуете `Reserve` (рис. 4.3).

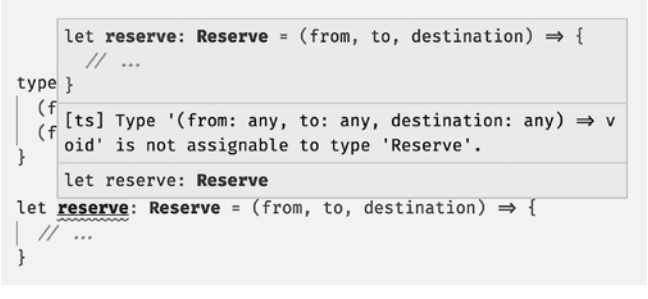


**Рис. 4.3.** Ошибка типа при упущении комбинированной сигнатуры перегрузки

Это происходит из-за особенности работы перегрузки сигнатуры в TypeScript. Если вы объявите набор сигнатур перегрузки для функции `f`, то с точки зрения вызывающего компонента тип `f` — это объединение сигнатур перегрузок. Но с позиции реализации `f` должен быть один скомбинированный тип, который может быть реализован. Вам придется самостоятельно объявить эту комбинированную сигнатуру вызова при реализации `f`. В примере с `Reserve` можете обновить функцию `reserve` так:

```
type Reserve = {
    (from: Date, to: Date, destination: string): Reservation
    (from: Date, destination: string): Reservation
} ❶

let reserve: Reserve = (
    from: Date,
    toOrDestination: Date | string,
```

```
    destination?: string
) => { ❷
    // ...
}
```

❶ Объявляем две перегруженные сигнатуры функций.

❷ Сигнатура реализации является результатом комбинирования двух сигнатур перегрузки (`Signature1 | Signature2` рассчитаны вручную). Обратите внимание, что комбинированная сигнатура невидима для функций, вызывающих `reserve`. С позиции потребителя сигнатура `Reserve` следующая:

```
type Reserve = {
    (from: Date, to: Date, destination: string): Reservation
    (from: Date, destination: string): Reservation
}
```

Примечательно, что она не содержит созданную нами сигнатуру:

```
// Неверно!
type Reserve = {
    (from: Date, to: Date, destination: string): Reservation
    (from: Date, destination: string): Reservation
    (from: Date, toOrDestination: Date | string,
        destination?: string): Reservation
}
```

Поскольку `reserve` может быть вызвана одним из двух способов, то при ее реализации вы можете доказать TypeScript, что проверили то, как она была вызвана (см. подраздел «Уточнение» на с. 160):

```
let reserve: Reserve = (
    from: Date,
    toOrDestination: Date | string,
    destination?: string
) => {
    if (toOrDestination instanceof Date && destination !== undefined) {
        // Book a one-way trip
    } else if (typeof toOrDestination === 'string') {
        // Book a round trip
    }
}
```

### СОХРАНЕНИЕ СПЕЦИФИЧНОСТИ СИГНАТУР ПЕРЕГРУЗОК

Как правило, каждая сигнатура перегрузки (`Reserve`) при объявлении типа перегруженной функции должна быть совместима с сигнатурой реализации (`reserve`). Это означает, что вы можете обобщать, объявляя сигнатуру реализации, лишь бы перегрузки были совместимы с ней. Например, этот код работает:

```
let reserve: Reserve = (
    from: any,
    toOrDestination: any,
    destination?: any
) => {
    // ...
}
```

При использовании перегрузок сохраняйте сигнатуру реализации максимально специфичной, чтобы облегчить реализацию функции. В нашем примере это значит вместо `any` предпочесть `Date` и `Date | string`.

Почему? Если вы типизируете параметр как `any` и захотите безопасно использовать его и получить преимущества автозавершения, вам придется доказать TypeScript, что это `Date`:

```
function getMonth(date: any): number | undefined {
    if (date instanceof Date) {
        return date.getMonth()
    }
}
```

Изначальная типизация параметра как `Date` избавит вас от лишней работы в реализации:

```
function getMonth(date: Date): number {
    return date.getMonth()
}
```

Перегрузки, естественно, встречаются в API DOM браузера. Например, `createElement` в API DOM используется для создания нового HTML-элемента. Он получает строку, соответствующую HTML-тегу, и возвращает новый HTML-элемент с типом этого тега. В TypeScript есть встроенные типы для HTML-элементов, которые включают:

- `HTMLAnchorElement` для элементов `<a>`;
- `HTMLCanvasElement` для элементов `<canvas>`;
- `HTMLTableElement` для элементов `<table>`.

Перегруженные сигнатуры вызовов — это естественный способ моделирования работы `createElement`. Прежде чем читать дальше, подумайте, как можно типизировать `createElement`?

Ответ:

```
type CreateElement = {
    (tag: 'a'): HTMLAnchorElement ❶
    (tag: 'canvas'): HTMLCanvasElement
    (tag: 'table'): HTMLTableElement
    (tag: string): HTMLElement ❷
}

let createElement: CreateElement = (tag: string): HTMLElement => { ❸
    // ...
}
```

❶ Перегружаем тип параметра, сопоставляя его с типами строчных литералов.

❷ Добавляем типичный случай: если пользователь передал стандартное имя тега или новейшее экспериментальное имя, которое еще не было внедрено в число встроенных деклараций типов TypeScript, то возвращаем общий `HTMLElement`. TypeScript обрабатывает перегрузки в порядке их объявления[1], поэтому когда вы вызываете `createElement` со строкой, у которой не определена конкретная перегрузка (например, `createElement ('foo')`), TypeScript использует `HTMLElement`.

❸ Чтобы типизировать параметр реализации, комбинируем все типы, которые этот параметр может иметь в сигнатурах перегрузки `createElement`. И получаем `'a', | 'canvas' | 'table' | 'string'`. Поскольку три типа строчных литералов являются подтипами `string`, тип упрощается до `string`.

[1] Чаще всего TypeScript поднимает литеральные перегрузки над нелитеральными, прежде чем начать обрабатывать их по порядку. Это может усложнить понимание ваших перегрузок инженерами, незнакомыми с таким поведением.

Во всех примерах этого раздела мы перегружали выражения функций. А как перегрузить декларацию функции? Конечно, в TypeScript есть решение — синтаксис для деклараций функций. Давайте перепишем наши перегрузки `createElement`:

```
function createElement(tag: 'a'): HTMLAnchorElement
function createElement(tag: 'canvas'): HTMLCanvasElement
function createElement(tag: 'table'): HTMLTableElement
function createElement(tag: string): HTMLElement {
    // ...
}
```

Вам решать, какой синтаксис использовать, в зависимости от вида перегружаемой функции (выражения функции или ее декларации).

Полные сигнатуры типов не ограничиваются перегрузкой вызова функции. Также можно использовать их для моделирования свойств функций. Поскольку функции в JavaScript являются просто вызываемыми объектами, добавьте им свойства, чтобы делать, например, следующее:

```
function warnUser(warning) {
    if (warnUser.wasCalled) {
        return
    }
    warnUser.wasCalled = true
    alert(warning)
}
warnUser.wasCalled = false
```

Так мы однократно показываем пользователю предупреждение. Давайте используем TypeScript для типизации полной сигнатуры `warnUser`:

```
type WarnUser = {
    (warning: string): void
    wasCalled: boolean
}
```

`warnUser` — это не только функция, которую можно вызвать, но и обладатель свойства `wasCalled`, имеющего тип `boolean`.

# Полиморфизм

До сих пор мы рассматривали конкретные типы и функции, их использующие. Что же такое конкретный тип?

- `boolean`;
- `string`;
- `Date[]`;
- `{a: number} | {b: string}`;
- `(numbers: number[]) => number`.

Конкретные типы полезны, когда вы точно знаете, какой тип ожидаете, и хотите сверить его с переданным типом. Но иногда нецелесообразно ограничивать поведение функции конкретным типом.

Представьте функцию `filter` для итерации по массиву и его очистки. В JavaScript она может выглядеть так:

```
function filter(array, f) {
    let result = []
    for (let i = 0; i < array.length; i++) {
        let item = array[i]
        if (f(item)) {
            result.push(item)
        }
    }
    return result
}

filter([1, 2, 3, 4], _ => _ < 3) // вычисляется как [1, 2]
```

Начнем с извлечения сигнатуры типа `filter` и добавления временных заместителей `unknown` для типов:

```
type Filter = {
    (array: unknown, f: unknown) => unknown[]
}
```

Затем попробуем заполнить типы, например `number`:

```
type Filter = {
    (array: number[], f: (item: number) => boolean): number[]
}
```

Эта сигнатура будет работает для массива чисел, но не для массивов строк, объектов, массивов и т. д. Попробуем использовать перегрузку для ее расширения, чтобы она работала для массива строк:

```
type Filter = {
    (array: number[], f: (item: number) => boolean): number[]
    (array: string[], f: (item: string) => boolean): string[]
}
```

Пока все хорошо, но прописывать перегрузку для каждого типа хлопотно. Как насчет массива объектов?

```
type Filter = {
    (array: number[], f: (item: number) => boolean): number[]
    (array: string[], f: (item: string) => boolean): string[]
    (array: object[], f: (item: object) => boolean): object[]
}
```

Начало выглядит неплохо, но если реализовать функцию `filter` с сигнатурой `filter: Filter` и ее использовать, то получится следующее:

```
let names = [
    {firstName: 'beth'},
    {firstName: 'caitlyn'},
    {firstName: 'xin'}
]

let result = filter(
    names,
    _ => _.firstName.startsWith('b')
) // Ошибка TS2339: свойство 'firstName' не существует в типе 'object'.

result[0].firstName // Ошибка TS2339: свойство 'firstName'
                    // не существует в типе 'object'.
```

В этом месте должно стать понятно, почему TypeScript выдает ошибку. Мы сообщили ему, что можем передать массив чисел, строк или объектов в `filter`, и передали массив объектов. Но, как вы помните, `object` ничего не сообщает TypeScript о конкретной форме самого объекта.

Что же делать?

Если раньше вы писали на языке, поддерживающем обобщенные типы, то наверняка выкрикнули: «ХОЧУ ОБОБЩЕННЫЕ ТИПЫ!» Хорошая новость в том, что вы правы, а плохая — вы только что разбудили своим криком соседского ребенка.

Для несведущих начну с определения обобщенных типов, а затем приведу пример с нашей функцией.

> **ПАРАМЕТР ОБОБЩЕННОГО ТИПА**
>
> Замещающий тип, используемый для применения ограничений на уровне типов в нескольких местах. Также известен как *параметр полиморфного типа*.

Вот как будет выглядеть тип `filter`, когда мы перепишем его с параметром обобщенного типа `T`:

```
type Filter = {
    <T>(array: T[], f: (item: T) => boolean): T[]
}
```

Мы сообщили: «Функция `filter` использует параметр обобщенного типа `T`. Мы не знаем, каким будет этот тип в дальнейшем, поэтому, TypeScript, если ты сможешь делать его вывод при каждом вызове `filter`, то будет очень хорошо». TypeScript выводит тип `T` на основе типа, который мы передаем для `array`. Как только TypeScript делает вывод, чем является `T` для вызова `filter`, он подставляет этот тип для каждого видимого `T`. `T` выступает в роли замещающего типа, который заполняется модулем проверки на основе контекста. Он параметризует тип `Filter`, поэтому мы и зовем его параметром обобщенного типа.

Фраза «параметр обобщенного типа» часто заменяется на «обобщенный тип» или «обобщение». В книге преимущественно будет использовано полное выражение.

Для объявления обобщенных типов используются угловые скобки (`<>`) (воспринимайте их как ключевое слово `type`). Место размещения скобок

определяет диапазон охватываемых типов (есть всего несколько мест, где вы можете их поставить). TypeScript, в свою очередь, убеждается, что внутри этого диапазона все экземпляры параметров обобщенных типов привязаны к одному реальному типу. В текущем примере при вызове `filter` TypeScript привяжет конкретные типы к обобщенному типу `T` в зависимости от обозначенного скобками диапазона. Какой именно тип привязывать к `T`, он решит исходя из того, с каким типом мы вызовем `filter`. Между угловых скобок вы можете объявить столько обобщений, сколько пожелаете, разделив их точкой с запятой.

`T` — это просто имя типа, такое же, как `A`, `Zebra` или `133t`. Принято использовать имена, состоящие из одной заглавной буквы `T` или `U`, `V`, `W` и т. д., в зависимости от того, сколько обобщенных типов требуется.

Если вы объявляете множество таких типов подряд или используете их сложным образом, рассмотрите вариант более описательных имен вроде `Value` или `WidgetType`.

Некоторые программисты предпочитают начинать с `A` вместо `T`. Сообщества пользователей разных языков программирования делают свой выбор согласно устоявшейся традиции. Например, функциональные программисты предпочитают `A`, `B`, `C` и т. д. из-за своего пристрастия к греческим буквам α, β и γ. Разработчики объектно-ориентированных языков склонны использовать `T` а-ля «Type». TypeScript же хоть и поддерживает оба этих стиля, но чаще использует последний.

Подобно тому как параметр функции повторно привязывается при каждом вызове функции, так же и каждый вызов `filter` получает свою привязку для `T`:

```
type Filter = {
    <T>(array: T[], f: (item: T) => boolean): T[]
}

let filter: Filter = (array, f) => // ...

// (a) T привязан к number
filter([1, 2, 3], _ => _ > 2)
```

```
// (b) T привязан к строке
filter(['a', 'b'], _ => _ !== 'b')

// (c) T привязан к {firstName: string}
let names = [
    {firstName: 'beth'},
    {firstName: 'caitlyn'},
    {firstName: 'xin'}
]
filter(names, _ => _.firstName.startsWith('b'))
```

TypeScript делает вывод привязок обобщенных типов на основе типов переданных аргументов. Рассмотрим, как он привязывает `T` для (a):

1. Исходя из сигнатуры `filter` TypeScript знает, что `array` — это массив элементов некоего типа `T`.
2. TypeScript замечает, что мы передали в массив `[1, 2, 3]`, а значит, `T` должен быть `number`.
3. Везде, где TypeScript видит `T`, он заменяет его на `number`. Следовательно, параметр `f: (item: T) => boolean` становится `f: (item: number) => boolean`, а возвращаемый тип `T[]` становится `number[]`.
4. TypeScript проверяет типы на совместимость и убеждается, что функция, которую мы передали как `f`, совместима со своей только что выведенной сигнатурой.

Обобщенные типы — это эффективный способ выразить более обширное действие функции, чем это позволяют конкретные типы. Воспринимать же их стоит в виде ограничений. Как аннотирование параметра функции в виде `n: number` ограничивает значение параметра `n` типом `number`, так и использование обобщенного типа `T` ограничивает тип любого привязываемого к `T` условием типа быть одинаковым в каждом `T`.

Обобщенные типы также могут применяться в псевдонимах типов, классах и интерфейсах — мы будем активно их использовать на протяжении книги. Я представлю их в контексте остальных тем.

Используйте обобщенные типы везде, где можете. Они помогут сохранить код обобщенным, переиспользуемым и кратким.

### Когда привязывать конкретные типы к обобщенным

Место объявления обобщенного типа не только определяет его диапазон, но также указывает TypeScript, когда нужно привязать к нему конкретный тип. Из предыдущего примера:

```
type Filter = {
    <T>(array: T[], f: (item: T) => boolean): T[]
}

let filter: Filter = (array, f) =>
    // ...
```

Мы объявили `<T>` как часть сигнатуры вызова (перед открывающимися скобками), и TypeScript привяжет конкретный тип к `T`, когда мы вызовем функцию типа `Filter`.

Если бы мы вместо этого ограничили диапазон `T` псевдонимом типа `Filter`, TypeScript потребовал бы от нас при использовании `Filter` привязать тип явно:

```
type Filter<T> = {
    (array: T[], f: (item: T) => boolean): T[]
}
let filter: Filter = (array, f) => // Ошибка TS2314: обобщенный тип
    // ...                          // 'Filter' требует 1 аргумент типа.
type OtherFilter = Filter          // Ошибка TS2314: условный тип
                                   // 'Filter' требует 1 аргумент типа.

let filter: Filter<number> = (array, f) =>
    // ...

type StringFilter = Filter<string>
let stringFilter: StringFilter = (array, f) =>
    // ...
```

Чаще всего TypeScript будет привязывать конкретные типы к обобщенным, когда вы используете их для функций при их вызове, для классов при их инстанцировании (см. раздел «Полиморфизм» на с. 131) или для псевдонимов типов и интерфейсов (см. раздел «Интерфейсы» на с. 119) при их использовании или реализации.

### Где можно объявлять обобщенные типы

Для каждого способа объявления сигнатуры вызова есть способ добавить к ней обобщенный тип:

```
type Filter = { ❶
   <T>(array: T[], f: (item: T) => boolean): T[]
}
let filter: Filter = // ...

type Filter<T> = { ❷
   (array: T[], f: (item: T) => boolean): T[]
}
let filter: Filter<number> = // ...

type Filter = <T>(array: T[], f: (item: T) => boolean) => T[] ❸
let filter: Filter = // ...

type Filter<T> = (array: T[], f: (item: T) => boolean) => T[] ❹
let filter: Filter<string> = // ...

function filter<T>(array: T[], f: (item: T) => boolean): T[] { ❺
    // ...
}
```

❶ Полная сигнатура вызова с диапазоном `T`, включающим только одну сигнатуру. Поскольку диапазон `T` ограничен одной сигнатурой, TypeScript привяжет `T` к реальному типу в этой сигнатуре при вызове функции типа `filter`. Каждый вызов `filter` будет получать свою собственную привязку для `T`.

❷ Полная сигнатура вызова с диапазоном `T`, включающим все сигнатуры. `T` объявлен как часть типа `Filter` (а не часть конкретной сигнатуры типа), и TypeScript привяжет `T`, когда вы объявите функцию типа `Filter`.

❸ Как ❶, но при помощи сокращенной сигнатуры вызова.

❹ Как ❷, но при помощи сокращенной сигнатуры вызова.

❺ Именованная сигнатура вызова функции с диапазоном `T`, ограниченным сигнатурой. TypeScript привяжет конкретный тип к `T` при вызове `filter`, и каждый вызов `filter` получит свою собственную привязку для `T`.

### FILTER И MAP В СТАНДАРТНОЙ БИБЛИОТЕКЕ

Наши определения для `filter` и `map` чрезвычайно похожи на те, что изначально присутствуют в TypeScript:

```
interface Array<T> {
    filter(
        callbackfn: (value: T, index: number, array: T[]) => any,
        thisArg?: any
    ): T[]
    map<U>(
        callbackfn: (value: T, index: number, array: T[]) => U,
        thisArg?: any
    ): U[]
}
```

Мы еще не рассматривали интерфейсы, но это определение говорит, что `filter` и `map` являются функциями в массиве типа `T`. Они обе получают функцию `callbackfn` и тип для `this` внутри функции.

`filter` использует параметр обобщенного типа `T`, чей диапазон охватывает весь интерфейс `Array`. `map` также использует `T` и добавляет второй параметр обобщенного типа `U`, чей диапазон включает только функцию `map`. Это значит, что TypeScript привяжет конкретный тип к `T` при создании массива и каждый вызов `filter` и `map` для этого массива также получит этот конкретный тип. Каждый вызов `map` будет получать свою собственную привязку для `U` помимо обращения к уже привязанному `T`.

В качестве второго примера напишем функцию `map`, похожую на `filter`, но вместо удаления элементов из массива преобразуем каждый из них посредством отображающей функции. Начнем с набросков реализации:

```
function map(array: unknown[], f: (item: unknown) =>
                                  unknown): unknown[] {
    let result = []
    for (let i = 0; i < array.length; i++) {
        result[i] = f(array[i])
    }
    return result
}
```

Прежде чем продолжить, продумайте, как сделать `map` обобщенной и заменяющей каждый тип `unknown` неким конкретным типом? Сколько обобщенных типов понадобится? Как их надо объявить и задать их диапазон в функции `map`? Какими должны быть типы `array` и `f` и возвращаемое значение?

Готовы? Попробуйте, вы справитесь!

Проверьте ответ:

```
function map<T, U>(array: T[], f: (item: T) => U): U[] {
    let result = []
    for (let i = 0; i < array.length; i++) {
        result[i] = f(array[i])
    }
    return result
}
```

Нужны именно два обобщенных типа: `T` для типа элементов, поступающих в массив, и `U` — для покидающих его. Мы передаем внутрь массив `T` и отображающую функцию, которая преобразует `T` в `U`. И в завершение возвращаем массив `U`.

Многие функции в стандартных библиотеках JavaScript являются обобщенными, особенно находящиеся в прототипе `Array`. Массивы могут содержать значения любого типа, который мы называем `T` и говорим, например, так: «`.push` получает аргумент типа `T`» или «`.map` отображает массив `T` в массив `U`».

## Вывод обобщенных типов

В большинстве случаев TypeScript проводит серьезную работу по выводу обобщенных типов. Когда вы вызываете функцию `map`, TypeScript делает вывод, что `T` — это `string`, а `U` — это `boolean`:

```
function map<T, U>(array: T[], f: (item: T) => U): U[] {
    // ...
}

map(
    ['a', 'b', 'c'],    // Массив T
    _ => _ === 'a'      // Функция, возвращающая U
)
```

Но их можно аннотировать явно по принципу «все или ничего»: либо каждый обобщенный тип, либо ни один из них:

```
map <string, boolean>(
    ['a', 'b', 'c'],
    _ => _ === 'a'
)

map <string>( // Ошибка TS2558: ожидается 2 аргумента типов, но получен 1.
    ['a', 'b', 'c'],
    _ => _ === 'a'
)
```

TypeScript проверит, что каждый выведенный обобщенный тип совместим с соответствующим ему явно привязанным обобщенным типом. В противном случае вы получите ошибку:

```
// OK, т.к. boolean совместим с boolean | string
map<string, boolean | string>(
    ['a', 'b', 'c'],
    _ => _ === 'a'
)

map<string, number>(
    ['a', 'b', 'c'],
    _ => _ === 'a'    // Ошибка TS2322: тип 'boolean' несовместим
)                     // с типом 'number'.
```

Поскольку TypeScript выводит конкретные типы для обобщенных на основе аргументов, передаваемых в обобщенную функцию, иногда может возникнуть такая ситуация:

```
let promise = new Promise(resolve =>
    resolve(45)
)
promise.then(result =>    // Выведен как {}
    result * 4    // Ошибка TS2362: левая часть арифметической
)                 // операции должна иметь тип 'any', 'number',
                  // 'bigint' или enum.
```

Что происходит? Почему TypeScript вывел `result` как `{}`? Потому что мы не предоставили ему достаточно информации. TypeScript для вывода

обобщенного типа использует только типы аргументов обобщенной функции, поэтому он по умолчанию вывел `T` как `{}`.

Чтобы это исправить, нужно явно аннотировать параметр обобщенного типа промисов:

```
let promise = new Promise<number>(resolve =>
    resolve(45)
)
promise.then(result => // number
    result * 4
)
```

### Псевдонимы обобщенных типов

Мы уже затрагивали псевдонимы обобщенных типов в примере с `Filter`. Как вы помните из предыдущей главы (см. подраздел «Массивы и кортежи только для чтения» на с. 56), типы `Array` и `ReadonlyArray` также относятся к обобщенным типам. Рассмотрим их применение в псевдонимах типов подробнее с помощью короткого примера.

Определим тип `MyEvent`, описывающий событие DOM вроде `click` или `mousedown`:

```
type MyEvent<T> = {
    target: T
    type: string
}
```

Обратите внимание, что единственное допустимое место для объявления обобщенного типа в псевдониме типа находится сразу после имени псевдонима типа и перед его присваиванием (`=`).

Свойство `MyEventTarget` указывает на элемент, в котором произошло событие: `<button />`, `<div />` и т. д. Например, событие кнопки можно описать так:

```
type ButtonEvent = MyEvent<HTMLButtonElement>
```

Применяя обобщенный тип вроде `MyEvent`, вы должны явно привязать его параметры при использовании, поскольку он не будет выведен:

```
let myEvent: MyEvent<HTMLButtonElement | null> = {
    target: document.querySelector('#myButton'),
    type: 'click'
}
```

Можете использовать `MyEvent` для построения другого типа, например `TimedEvent`. Когда обобщенный тип `T` в `TimedEvent` будет привязан, TypeScript привяжет его и к `MyEvent`:

```
type TimedEvent<T> = {
    event: MyEvent<T>
    from: Date
    to: Date
}
```

Также можете использовать псевдоним обобщенного типа в сигнатуре функции. Когда TypeScript привяжет тип к `T`, он также привяжет его и к `MyEvent`:

```
function triggerEvent<T>(event: MyEvent<T>): void {
    // ...
}

triggerEvent({ // T является Element | null
    target: document.querySelector('#myButton'),
    type: 'mouseover'
})
```

Рассмотрим этот процесс пошагово:

1. Вызываем `triggerEvent` с объектом.
2. TypeScript видит, что согласно сигнатуре функции переданный нами аргумент должен иметь тип `MyEvent<T>`. Он также замечает, что мы определили `MyEvent<T>` как `{target: T, type: string}`.
3. TypeScript замечает, что поле `target` переданного нами объекта является `document.querySelector('#myButton')`. Это подразумевает, что `T` должен иметь тот же тип, что и `document.querySelector('#myButton')`, то есть `Element | Null`. И теперь `T` привязан к `Element | Null`.
4. TypeScript проходит по коду и заменяет каждый `T` на `Element | Null`.
5. TypeScript проверяет, чтобы все типы соответствовали правилам совместимости. Код проходит проверку типов.

## Ограниченный полиморфизм

В этом разделе я буду использовать в качестве примера двоичное дерево. Ничего страшного, если вы ранее не работали с двоичными деревьями. Для наших целей достаточно понимания следующих основ:

- Двоичное дерево — это вид структуры данных.
- Оно состоит из узлов.
- Узел содержит значение и может указывать не более чем на два дочерних узла.
- Узел может быть одного из двух типов: листовой узел (у которого нет дочерних) или внутренний узел (имеющий не менее одного дочернего).

Иногда недостаточно просто сказать: «Этот элемент имеет некий обобщенный тип `T`, и другой элемент тоже должен его иметь». В некоторых случаях вы хотите сказать: «Тип `U` должен быть как минимум `T`». Мы зовем это установкой верхней границы `U`.

Зачем? Представим реализацию двоичного дерева с тремя типами узлов:

1. Обычные `TreeNodes`.
2. `LeafNodes`, являющиеся `TreeNodes`, не имеющими дочерних узлов.
3. `InnerNodes`, являющиеся `TreeNodes`, имеющими дочерние узлы.

Начнем с объявления типов для узлов:

```
type TreeNode = {
    value: string
}
type LeafNode = TreeNode & {
    isLeaf: true
}
type InnerNode = TreeNode & {
    children: [TreeNode] | [TreeNode, TreeNode]
}
```

Здесь мы говорим, что TreeNode является объектом с одним свойством — value. Тип LeafNode имеет те же свойства, что и TreeNode, плюс свойство isLeaf, которое всегда true. InnerNode также имеет все свойства TreeNode плюс свойство children, указывающее на один или два дочерних узла.

Теперь напишем функцию mapNode, которая получает TreeNode, делает отображение на его значение и возвращает новый TreeNode. Нам нужно получить функцию mapNode, которую можно будет использовать так:

```
let a: TreeNode = {value: 'a'}
let b: LeafNode = {value: 'b', isLeaf: true}
let c: InnerNode = {value: 'c', children: [b]}

let a1 = mapNode(a, _ => _.toUpperCase()) // TreeNode
let b1 = mapNode(b, _ => _.toUpperCase()) // LeafNode
let c1 = mapNode(c, _ => _.toUpperCase()) // InnerNode
```

Стоп, а как можно написать функцию mapNode, получающую подтип TreeNode и возвращающую тот же самый подтип? При передаче в нее LeafNode должен возвращаться LeafNode, при передаче InnerNode — InnerNode, а при передаче TreeNode — TreeNode. Прежде чем продолжать, подумайте, как бы вы это сделали? Возможно ли это вообще?

Вот ответ:

```
function mapNode<T extends TreeNode>( ❶
    node: T, ❷
    f: (value: string) => string
): T { ❸
    return {
        ...node,
        value: f(node.value)
    }
}
```

❶ mapNode — это функция, определяющая один параметр обобщенного типа — T. T имеет верхнюю границу в виде TreeNode. Это значит, что T — это либо TreeNode, либо подтип TreeNode.

❷ mapNode получает два параметра. Первый — это node типа T, потому что в ❶ мы указали, что node extends TreeNode. Если мы передаем нечто не являющееся TreeNode — например, пустой объект {}, null или массив

из нескольких `TreeNode`, — это сразу вызовет красное подчеркивание. `node` должен быть либо `TreeNode`, либо подтипом `TreeNode`.

❸ `mapNode` возвращает значение типа `T`. Вспомните, что `T` может быть либо `TreeNode`, либо подтипом `TreeNode`.

Почему надо объявить `T` таким образом?

- Если бы мы типизировали `T` просто как `T` (не добавляя `extends TreeNode`), тогда `mapNode` выдавала бы ошибку при компиляции, потому что нельзя безопасно прочитать `node.value` в неограниченном `node` типа `T` (пользователь может передать число).
- Если бы мы вовсе не стали использовать `T` и объявили `mapNode` как `(node: TreeNode, f: (value: string) => string) => TreeNode`, тогда мы утеряли бы информацию после отображения узла: `a1`, `b1`, и `c1` были бы просто `TreeNode`.

Указывая, что `T extends TreeNode`, мы сохраняем конкретный тип вводного узла (`TreeNode`, `LeafNode` или `InnerNode`) даже после его отображения.

#### Ограниченный полиморфизм с несколькими ограничениями

В последнем примере мы накладывали на `T` одно ограничение: `T` должен быть как минимум `TreeNode`. Но что, если вам нужно сделать несколько ограничений?

Просто расширьте пересечение (`&`) ограничений:

```
type HasSides = {numberOfSides: number}
type SidesHaveLength = {sideLength: number}

function logPerimeter< ❶
    Shape extends HasSides & SidesHaveLength ❷
>(s: Shape): Shape { ❸
    console.log(s.numberOfSides * s.sideLength)
    return s
}

type Square = HasSides & SidesHaveLength
let square: Square = {numberOfSides: 4, sideLength: 3}
logPerimeter(square) // Square, logs "12"
```

❶ `logPerimeter` — это функция, получающая одни аргумент `s` типа `Shape`.

❷ `isShape` — это обобщенный тип, расширяющий типы `HasSides` (имеет стороны) и `SidesHaveLength` (стороны имеют длину). Иначе говоря, `Shape` должна как минимум иметь стороны с длинами.

❸ `logPerimeter` возвращает значение точно такого же типа, какой вы ей передали.

#### Использование ограниченного полиморфизма для моделирования арности

Еще одна ситуация, где уместен ограниченный полиморфизм, — это моделирование переменных функций (функций, получающих любое количество аргументов). В качестве примера реализуем нашу версию встроенных в JavaScript функций `call` (получающих функцию и переменное число аргументов, а затем применяющих эти аргументы к этой функции)[1]. Мы определим и используем ее, указав `unknown` для типов, которые заполним позже:

```
function call(
    f: (...args: unknown[]) => unknown,
    ...args: unknown[]
): unknown {
    return f(...args)
}

function fill(length: number, value: string): string[] {
    return Array.from({length}, () => value)
}

call(fill, 10, 'a') // вычисляется как массив 10и 'a'
```

Теперь заместим `unknown`. Нам нужны следующие ограничения:

- `f` должна быть функцией, получающей набор аргументов `T` и возвращающей тип `R`. Заранее мы не знаем, сколько аргументов у нее будет;
- `call` должна получать `f` наряду с тем же набором аргументов `T`, который получает сама `f`. И снова мы не знаем, сколько аргументов ожидать;
- `call` должна возвращать тот же тип `R`, что и `f`.

---

[1] Для упрощения реализации построим функцию `call` так, чтобы она не принимала во внимание `this`.

Понадобятся два типа параметров: `T`, который является массивом аргументов, и `R`, являющийся непостоянным возвращаемым значением. Заполним типы:

```
function call<T extends unknown[], R>( ❶
    f: (...args: T) => R, ❷
    ...args: T ❸
): R { ❹
    return f(...args)
}
```

Разберем пошагово, как это работает:

❶ `call` является переменной функцией (принимает любое число аргументов) с двумя типами параметров: `T` и `R`. `T` является подтипом `unknown[]`, выступая как массив или кортеж любого типа.

❷ Первый параметр `call` — это переменная функция `f`, аргументы которой получают тип как у `args`.

❸ Еще `call` имеет переменное число дополнительных параметров `...args`, описывающих переменное число аргументов. `args` имеет тип `T`, а `T` должен иметь тип массива (если мы забудем сообщить, что `T` расширяет тип массива, TypeScript подчеркнет ошибку красным), поэтому TypeScript выведет тип `T` как кортеж исходя из конкретных аргументов, переданных нами для `args`.

❹ `call` возвращает значение типа `R` (`R` привязан к тому типу, который возвращает `f`).

Теперь при вызове `call` TypeScript будет точно знать возвращаемый тип и станет ругаться, если мы передадим неверное число аргументов:

```
let a = call(fill, 10, 'a')         // string[]
let b = call(fill, 10)              // Ошибка TS2554: ожидается
                                    // 3 аргумента, но получено 2.
let c = call(fill, 10, 'a', 'z')    // Ошибка TS2554: ожидается
                                    // 3 аргумента, но получено 4.
```

В подразделе «Улучшение вывода типов для кортежей» на с. 177 мы задействуем схожую технику, чтобы воспользоваться преимуществом TypeScript при выводе типов кортежей для оставшихся параметров.

### Предустановки обобщенных типов

Подобно тому как вы задаете параметрам функции значения по умолчанию, вы можете задавать предустановки обобщенным типам. Возьмем в качестве примера тип `MyEvent` из подраздела «Псевдонимы обобщенных типов» на с. 100, где мы использовали его для моделирования событий DOM. Выглядит он так:

```
type MyEvent<T> = {
    target: T
    type: string
}
```

Чтобы создать новое событие, явно привяжем обобщенный тип к `MyEvent`, представляя тип *HTML*-элемента, на который было направлено событие:

```
let buttonEvent: MyEvent<HTMLButtonElement> = {
    target: myButton,
    type: string
}
```

Если тип элемента, к которому будет привязан `MyEvent`, заранее не известен, можно добавить предустановку для обобщенного типа `MyEvent`:

```
type MyEvent<T = HTMLElement> = {
    target: T
    type: string
}
```

Или добавить ограничение для `T`, чтобы убедиться, что `T` является *HTML*-элементом:

```
type MyEvent<T extends HTMLElement = HTMLElement> = {
    target: T
    type: string
}
```

Теперь можно легко создать событие, не являющееся конкретным для отдельного типа *HTML*-элемента, не привязывая вручную `T`, принадлежащий `MyEvent`, к `HTMLElement`:

```
let myEvent: MyEvent = {
    target: myElement,
    type: string
}
```

Обратите внимание, что, подобно опциональным параметрам в функции, обобщенные типы с предустановками должны идти после обобщенных типов без предустановок:

```
// Хорошо
type MyEvent2<
    Type extends string,
    Target extends HTMLElement = HTMLElement,
> = {
    target: Target
    type: Type
}

// Плохо
type MyEvent3<
    Target extends HTMLElement = HTMLElement,
    Type extends string  // Ошибка TS2706: необходимые параметры
> = {                      // типов не могут следовать за опциональными.
    target: Target
    type: Type
}
```

## Разработка на основе типов

В системе типов заложен огромный потенциал. При написании кода на TypeScript вы часто будете замечать, что «впереди идут типы». Это, конечно же, говорит о разработке на основе типов.

> **РАЗРАБОТКА НА ОСНОВЕ ТИПОВ**
>
> Стиль программирования, где сначала прописываются сигнатуры типов, а значения подставляются позже.

Суть статической системы типов в ограничении значений типов, которые может содержать выражение. Чем более выразительна система типов, тем

больше она вам сообщает о значении, содержащемся в этом выражении. Когда вы применяете выразительную систему типов к функции, ее сигнатура типа в итоге может сообщать вам бóльшую часть того, что нужно знать об этой функции.

Рассмотрим сигнатуру типа для функции `map`, использованной в этой главе:

```
function map<T, U>(array: T[], f: (item: T) => U): U[] {
    // ...
}
```

Даже если вы раньше не видели эту функцию, с первого взгляда понятно, что она делает[1]: получает массив `T` и функцию, производящую отображение из `T` в `U`, и возвращает массив `U`.

Начните написание программы в TypeScript с определения сигнатур типов функций. Так вы убедитесь, что все осмысленно на верхнем уровне, прежде чем спускаться на уровень реализации.

Вы заметите, что до сих пор мы делали наоборот. Теперь, когда вы уже навострились в написании и типизации функций в TypeScript, сменим режимы и начнем сначала делать наброски типов.

## Итоги

В этой главе мы рассмотрели объявление и вызов функции, типизацию параметров и выражение в TypeScript таких особенностей JavaScript-функций, как значения по умолчанию, оставшиеся параметры, функции-генераторы и итераторы. Мы изучили разницу между сигнатурами вызовов функций и реализацией, контекстную типизацию, а также различные способы перегрузки функций. В завершение мы рассмотрели полиморфизм для функций и псевдонимов типов, а именно научились работать с обобщенными типами. А также обозначили преимущества разработки на основе типов.

[1] Есть несколько языков программирования (вроде языка Irdis, подобного Haskell), которые имеют встроенные решатели задач для удовлетворения ограничений, автоматически создающие реализации тел функций на основе прописываемых вами сигнатур.

## Упражнения к главе 4

1. Для каких частей сигнатуры типа функции TypeScript делает вывод: параметры, возвращаемый тип или и то и другое?
2. Является ли JavaScript-объект `arguments` типобезопасным? Если нет, то что вместо него можно использовать?
3. Как забронировать отпуск, который начнется сразу? Обновите перегруженную функцию `reserve` из этой главы (см. подраздел «Типы перегруженных функций» на с. 83), добавив третью сигнатуру вызова, которая получает только место назначения без явной даты отправки. Обновите реализацию функции `reserve`, чтобы она поддерживала новую перегруженную сигнатуру.
4. (Сложно.) Обновите реализацию `call` из этой главы (см. подраздел «Использование ограниченного полиморфизма для моделирования арности» на с. 105), чтобы она работала только для функций, чей второй аргумент является `string`. Для остальных функций реализация должна проваливаться при компиляции.
5. Реализуйте небольшую типобезопасную библиотеку для проверки утверждений — `is`. Начните с типов. По завершении вы должны иметь возможность использовать ее следующим образом:

```
// Сравнить string и string
is('string', 'otherstring') // false

// Сравнить boolean и boolean
is(true, false) // false

// Сравнить number и number
is(42, 42) // true

// Сравнение двух различных типов должно выдавать ошибку
// при компиляции
is(10, 'foo') // Ошибка TS2345: аргумент типа '"foo"' не может
              // быть присвоен параметру типа 'number'.

// (Сложно.) Нужно иметь возможность передать любое число аргументов
is([1], [1, 2], [1, 2, 3]) // false
```

ГЛАВА 5

# Классы и интерфейсы

Если вы, как и большинство программистов, раньше работали с объектно-ориентированным языком, то классы — это ваш конек. Они являются способом организации кода и служат главным инструментом инкапсуляции. Вы будете рады узнать, что классы TypeScript заимствуют очень многое из C# и поддерживают модификаторы видимости, инициализаторы свойств, полиморфизм, декораторы и интерфейсы. Благодаря тому что классы в TypeScript компилируются в обычные JavaScript-классы, вы также можете выражать JavaScript-примеси типобезопасным способом.

Некоторые особенности классов в TypeScript, вроде инициализаторов и декораторов, также поддерживаются классами JavaScript[1] и, следовательно, генерируют код среды выполнения. Другие же особенности, вроде модификаторов видимости, интерфейсов и обобщенных типов, относятся только к TypeScript и существуют лишь в процессе компиляции, не генерируя код при компиляции в JavaScript.

В этой главе я продемонстрирую расширенный пример работы с классами в TypeScript, чтобы вы могли не только выработать некоторую интуицию в отношении объектно-ориентированных особенностей языка, но и понять, как и зачем их использовать. Старайтесь не отставать и экспериментируйте с кодом в редакторе по ходу изучения темы.

## Классы и наследование

Разработаем движок, моделирующий игру в шахматы и предоставляющий API для двух игроков.

Начнем с типов:

```
// Представляет игру в шахматы
class Game {}
```

[1] Или вскоре будут поддерживаться классами JavaScript.

```
// Шахматная фигура
class Piece {}

// Набор координат шахматной фигуры
class Position {}
```

Есть шесть типов шахматных фигур:

```
// ...
class King extends Piece {}
class Queen extends Piece {}
class Bishop extends Piece {}
class Knight extends Piece {}
class Rook extends Piece {}
class Pawn extends Piece {}
```

Каждая фигура имеет цвет и текущую позицию. В шахматах позиции моделируются как пара координат (буква, число). Буквы идут слева направо по оси *X*, а числа снизу вверх по оси *Y* (рис. 5.1).

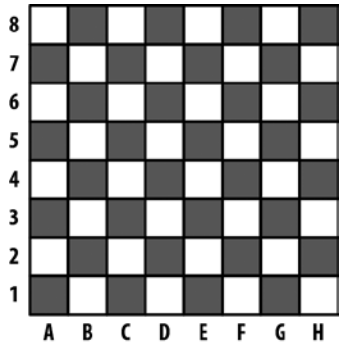


**Рис. 5.1.** Стандартная шахматная нотация: A–H (ось X) называется вертикалью, а 1–8 (ось Y) — горизонталью

Добавим цвет и позицию для класса `Piece`:

```
type Color = 'Black' | 'White'
type File = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H'
type Rank = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 ❶

class Position {
    constructor(
```

```
        private file: File, ❷
        private rank: Rank
    ) {}
}

class Piece {
    protected position: Position ❸
    constructor(
        private readonly color: Color, ❹
        file: File,
        rank: Rank
    ) {
        this.position = new Position(file, rank)
    }
}
```

❶ Имея относительно немного цветов, горизонталей и вертикалей, мы можем вручную пронумеровать их возможные значения в виде литералов типов. Это повысит типобезопасность.

❷ Модификатор доступа `private` в конструкторе автоматически присваивает параметр к `this` (`this.file` и т. д.) и устанавливает его видимость как приватную. Это значит, что код внутри экземпляра `Position` может считывать из приватного параметра и производить в него запись, но код, находящийся вне экземпляра `Position`, не может этого сделать. Различные экземпляры `Position` могут получать доступ к приватным членам друг друга. Экземпляры же любого другого класса — даже подкласса `Position` — не могут.

❸ Объявляем переменную экземпляра `position` как `protected`. Наподобие `private`, `protected` присваивает свойство к `this`, но в отличие от него делает это свойство видимым как для экземпляров `Piece`, так и для экземпляров любых подклассов `Piece`. Мы не присвоили значение для `position` при ее объявлении, и теперь нужно присвоить его в функции конструктора `Piece`, иначе TypeScript сообщит, что переменная не присвоена однозначно (является `T | undefined` вместо `T`), потому что в инициализаторе свойств ее значение тоже не указано. Может понадобиться обновить ее сигнатуру и указать, что она не обязательно является `Position`, но может быть `undefined`.

❹ `new Piece` получает три параметра: `color`, `file` и `rank`. Мы добавили два модификатора для `color`: `private`, означающий его присвоение

`this` и гарантирующий, что он будет доступен только из класса `Piece`, и `readonly`, обеспечивающий после изначального присваивания его доступность только для чтения.

**TSC-ФЛАГИ: STRICTNULLCHECKS И STRICTPROPERTYINITIALIZATION**

Чтобы активировать проверку присваивания переменных в экземплярах классов, включите флаги `strictNullChecks` и `strictPropertyInitialization` в tsconfig.json (не нужно, если активирован режим `strict`).

TypeScript поддерживает три модификатора доступа для свойств и методов класса.

`public`

Доступен отовсюду. Это уровень доступа по умолчанию.

`protected`

Доступен из экземпляров определенного класса и его подклассов.

`private`

Доступен только из экземпляров определенного класса.

Используя модификаторы доступа, вы можете конструировать классы, которые не столько выражают информацию об их реализации, сколько грамотно определяют API для их использования.

Допустим, мы хотим, чтобы пользователи не инстанцировали новый класс `Piece` непосредственно, а расширяли его, создавая классы `Queen`, `Bishop` и т. д., и инстанцировали их. Система типов позволит сделать это с помощью ключевого слова `abstract`:

```
// ...
abstract class Piece {
    constructor(
        // ...
```

Теперь, если вы попробуете инстанцировать `Piece` непосредственно, TypeScript будет ругаться:

```
new Piece('White', 'E', 1)    // Ошибка TS2511: невозможно создать
                              // экземпляр абстрактного класса.
```

Ключевое слово `abstract` означает, что вы не можете напрямую инстанцировать класс, но все еще можете определять для него методы:

```
// ...
abstract class Piece {
    // ...
    moveTo(position: Position) {
        this.position = position
    }
    abstract canMoveTo(position: Position): boolean
}
```

Теперь наш класс `Piece`:

- Сообщает подклассам, что они должны реализовывать метод с именем `canMoveTo`, который совместим с заданной сигнатурой. В противном случае при компиляции возникнет ошибка типа «абстрактный класс реализуется только вместе с его методами».
- Имеет предустановленную реализацию для `moveTo` (которую его подклассы могут при желании переопределять). Мы не добавляли модификатор доступа для `moveTo`, и он остался `public`.

Обновим `King`, добавив реализацию `canMoveTo`, чтобы выполнить новое требование. А также реализуем функцию `distanceFrom`, чтобы легче рассчитывать расстояние между двумя фигурами:

```
// ...
class Position {
    // ...
    distanceFrom(position: Position) {
        return {
            rank: Math.abs(position.rank - this.rank),
            file: Math.abs(position.file.charCodeAt(0) -
                           this.file.charCodeAt(0))
        }
    }
}

class King extends Piece {
    canMoveTo(position: Position) {
        let distance = this.position.distanceFrom(position)
        return distance.rank < 2 && distance.file < 2
    }
}
```

В начале новой игры мы будем автоматически создавать доску и фигуры:

```
// ...
class Game {
    private pieces = Game.makePieces()

    private static makePieces() {
        return [

            // Короли
            new King('White', 'E', 1),
            new King('Black', 'E', 8),

            // Ферзи
            new Queen('White', 'D', 1),
            new Queen('Black', 'D', 8),

            // Слоны
            new Bishop('White', 'C', 1),
            new Bishop('White', 'F', 1),
            new Bishop('Black', 'C', 8),
            new Bishop('Black', 'F', 8),

            // ...
        ]
    }
}
```

Благодаря строгой типизации `Rank` и `File` ввод другой буквы (например, `'J'`) или выходящего за диапазон числа (например, `12`) спровоцирует ошибку при компиляции (рис. 5.2).

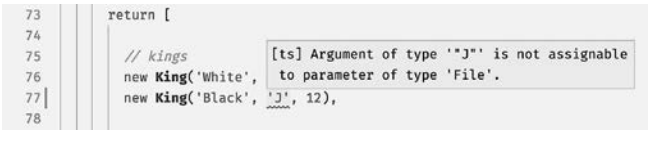


**Рис. 5.2.** TypeScript помогает придерживаться верных горизонталей и вертикалей

Этого достаточно для демонстрации работы классов в TypeScript. Не мне вам рассказывать, как ходят ладьи и пр. Попробуйте сами завершить реализацию игры.

Подытожим:

- Объявляйте и расширяйте классы при помощи ключевых слов `class` и `extends` соответственно.
- Классы могут быть либо конкретными, либо абстрактными. `abstract`-классы могут иметь `abstract`-свойства и `abstract`-методы.
- Методы `private`, `protected` или по умолчанию `public` могут быть статическими или методами экземпляра.
- Классы могут иметь свойства экземпляра: `private`, `protected` или по умолчанию `public`. Можно объявлять их в параметрах конструктора или в качестве инициализаторов свойств.
- При объявлении свойств экземпляра можно отмечать их как `readonly`.

## super

Как и JavaScript, TypeScript поддерживает вызовы `super`. Если дочерний класс переопределяет метод родительского класса (например, `Queen` и `Piece` — оба реализуют метод `take`), то дочерний экземпляр может сделать вызов `super`, чтобы вызвать версию метода родителя (к примеру, `super.take`). Есть два вида вызовов `super`:

- Вызовы метода вроде `super.take`.
- Вызовы конструктора, которые имеют особую форму `super()` и могут быть вызваны только из функции конструктора. Если дочерний класс имеет функцию конструктора, то нужно вызывать `super()` из этого дочернего конструктора, чтобы присоединить класс корректно. (TypeScript предупредит, если вы забудете. В этом смысле он похож на надежного футуристического робота-слона.)

Заметьте, что посредством `super` можно обращаться только к методам родительского класса, но не к его свойствам.

## Использование this в качестве возвращаемого типа

`this` можно использовать в качестве значения или типа (см. подраздел «Типизация this» на с. 73). При работе с классами тип `this` может оказаться полезен для аннотирования возвращаемых типов методов.

Создадим упрощенную версию структуры данных `Set` в ES6, которая поддерживает две операции: добавление числа в набор и проверку, находится ли данное число в наборе. Используется она так:

```
let set = new Set
set.add(1).add(2).add(3)
set.has(2) // верно
set.has(4) // неверно
```

Определим класс `Set` начиная с метода `has`:

```
class Set {
    has(value: number): boolean {
        // ...
    }
}
```

Что насчет `add`? Вызывая `add`, обратно вы получаете экземпляр `Set`. Можно типизировать его так:

```
class Set {
    has(value: number): boolean {
        // ...
    }
    add(value: number): Set {
        // ...
    }
}
```

Пока все хорошо. Что произойдет, когда мы попробуем создать подкласс `Set`?

```
class MutableSet extends Set {
    delete(value: number): boolean {
        // ...
    }
}
```

Конечно, метод `add`, принадлежащий `Set`, по-прежнему возвращает `Set`, который для подкласса придется заменить на `MutableSet`:

```
class MutableSet extends Set {
    delete(value: number): boolean {
        // ...
    }
```

```
    add(value: number): MutableSet {
        // ...
    }
}
```

При работе с расширяющими классами утомительно переписывать сигнатуры для каждого метода, возвращающего `this`. И если только так можно угодить модулю проверки типов, то какой смысл делать наследование от базового класса?

Используем `this` в качестве аннотации возвращаемого типа, чтобы TypeScript проделал работу за нас:

```
class Set {
    has(value: number): boolean {
        // ...
    }
    add(value: number): this {
        // ...
    }
}
```

Теперь вы можете убрать перезапись `add` из `MutableSet`, поскольку `this` в `Set` указывает на экземпляр `Set`, а `this` в `MutableSet` указывает на экземпляр `MutableSet`:

```
class MutableSet extends Set {
    delete(value: number): boolean {
        // ...
    }
}
```

Эта особенность очень удобна при работе с цепочками API (см. подраздел «Паттерн строитель» на с. 141).

# Интерфейсы

Классы часто используются вместе с интерфейсами.

Подобно псевдонимам типов, интерфейсы являются способом назвать тип без необходимости определять его встроенным. Псевдонимы типов и интерфейсы — это в целом два синтаксиса для одной задачи (как выражения

и декларации функций), но есть небольшие отличия. Начнем с изучения их общих характеристик. Рассмотрите следующий псевдоним типа:

```
type Sushi = {
    calories: number
    salty: boolean
    tasty: boolean
}
```

Его легко переписать в виде интерфейса:

```
interface Sushi {
    calories: number
    salty: boolean
    tasty: boolean
}
```

Псевдоним типа `Sushi` можно заменить на интерфейс `Sushi`. Обе декларации определяют формы, и эти формы являются совместимыми (на деле они идентичны).

Становится интереснее, когда вы начинаете комбинировать типы. Смоделируем дополнительную еду:

```
type Cake = {
    calories: number
    sweet: boolean
    tasty: boolean
}
```

Наличие калорий и вкуса свойственно разной еде — не только `Sushi` и `Cake`. Поэтому давайте выделим `Food` в отдельный тип и переопределим виды еды в нем:

```
type Food = {
    calories: number
    tasty: boolean
}
type Sushi = Food & {
    salty: boolean
}
type Cake = Food & {
    sweet: boolean
}
```

Почти идентично вы можете проделать то же самое с интерфейсами:

```
interface Food {
    calories: number
    tasty: boolean
}
interface Sushi extends Food {
    salty: boolean
}
interface Cake extends Food {
    sweet: boolean
}
```

Интерфейсы не обязательно должны расширять другие интерфейсы. На деле интерфейс может расширить любую форму: объект `type`, `class` или другой `interface`.

В чем отличие типов от интерфейсов? Отличия три, и они не сильно выражены.

Во-первых, псевдонимы типов более обобщены: их правая часть может быть любого типа, включая выражение типа (тип и, возможно, операции вроде `&` или `|`). А правая часть интерфейса должна быть формой. Следующие псевдонимы типов никак не переписать в виде интерфейсов:

```
type A = number
type B = A | string
```

Во-вторых, при расширении интерфейса TypeScript будет проверять, совместим ли этот интерфейс с произведенным вами расширением:

```
interface A {
    good(x: number): string
    bad(x: number): string
}

interface B extends A {
    good(x: string | number): string
    bad(x: string): string  // Ошибка TS2430: Interface 'B'
                            // некорректно расширяет
}                           // interface 'A'. Тип 'number'
                            // несовместим с типом 'string'.
```

Это не тот случай, когда вы используете типы пересечений: если вы превратите интерфейсы из последнего примера в псевдонимы типов и расширите (`extends`) их в пересечения (`&`), TypeScript постарается скомбинировать это расширение с расширяемым типом, формируя в итоге перегруженную сигнатуру для `bad` вместо выдачи ошибки при компиляции (проверьте это в редакторе).

Когда вы моделируете наследование для типов объектов, проверка совместимости, производимая TypeScript в отношении интерфейсов, может оказаться очень полезной для перехвата ошибок.

В-третьих, несколько интерфейсов с одинаковым именем в одной области подвергаются автоматическому слиянию. А несколько одноименных псевдонимов типов в одной области будут вызывать ошибку при компиляции. Эта особенность называется слиянием деклараций.

## Слияние деклараций

Слияние деклараций в TypeScript — это способ автоматического комбинирования нескольких деклараций, имеющих одинаковое имя. Мы уже встречались с этим процессом, когда знакомились с перечислениями (см. подраздел «Enum» на с. 60). Слияния также встречаются при работе с такими возможностями, как декларации `namespace` (см. раздел «Пространства имен» на с. 274). В текущем подразделе мы вкратце познакомимся со слиянием деклараций в контексте интерфейсов. Для более глубокого изучения темы обратитесь к разделу «Слияние деклараций» на с. 279.

Если вы объявите два одноименных интерфейса `User`, TypeScript автоматически объединит их в один интерфейс:

```
// User имеет одно поле, name
interface User {
    name: string
}

// Теперь User имеет два поля, name и age
interface User {
    age: number
}
```

```
let a: User = {
    name: 'Ashley',
    age: 30
}
```

Вот что произойдет, если вы повторите этот пример с псевдонимами типов:

```
type User = { // Ошибка TS2300: повторяющийся идентификатор 'User'.
    name: string
}

type User = { // Ошибка TS2300: повторяющийся идентификатор 'User'.
    age: number
}
```

Обратите внимание, что два интерфейса не должны конфликтовать. Если один типизирует `property`(свойство) как `T`, а другой типизирует его как `U` и `T` не совпадет с `U`, тогда вы получите ошибку:

```
interface User {
    age: string
}

interface User {
    age: number // Ошибка TS2717: последующие декларации свойств
                // должны иметь тот же тип.
}               // Свойство 'age' должно иметь тип 'string',
                // но здесь имеет тип 'number'.
```

И если интерфейс объявляет обобщенные типы (см. раздел «Полиморфизм» на с. 131), то для слияния двух интерфейсов они должны быть объявлены одинаково — вплоть до имен.

```
interface User<Age extends number> { // Ошибка TS2428: все декларации
    age: Age                         // 'User' должны иметь
                                     // идентичные параметры типа.
}

interface User<Age extends string> {
    age: Age
}
```

Это редкий случай, когда TypeScript проверяет не только совместимость двух типов, но и их *идентичность*.

## Реализации

При объявлении класса можно использовать ключевое слово `implements`, чтобы указать, что он соответствует определенному интерфейсу. Подобно другим явным аннотациям, это удобный способ добавить ограничение на уровне типов, чтобы класс был реализован максимально близко к реализации и позже не возникло проблем. Это также распространенный способ реализовывать паттерны проектирования вроде адаптеров, фабрик и т. д. (примеры — в конце главы).

Вот как это выглядит:

```
interface Animal {
    eat(food: string): void
    sleep(hours: number): void
}

class Cat implements Animal {
    eat(food: string) {
        console.info('Ate some', food, '. Mmm!')
    }
    sleep(hours: number) {
        console.info('Slept for', hours, 'hours')
    }
}
```

`Cat` обязан реализовать не только каждый метод, объявленный в `Animal`, но и дополнительные методы и свойства.

Интерфейсы могут объявлять свойства экземпляров, но не модификаторы видимости (`private`, `protected` и `public`) и использовать ключевое слово `static`. Также можно отмечать свойства экземпляров как `readonly`, как в примерах из главы 3:

```
interface Animal {
    readonly name: string
    eat(food: string): void
    sleep(hours: number): void
}
```

Вы не ограничены реализацией одного интерфейса:

```
interface Animal {
    readonly name: string
```

```
    eat(food: string): void
    sleep(hours: number): void
}

interface Feline {
    meow(): void
}

class Cat implements Animal, Feline {
    name = 'Whiskers'
    eat(food: string) {
        console.info('Ate some', food, '. Mmm!')
    }
    sleep(hours: number) {
        console.info('Slept for', hours, 'hours')
    }
    meow() {
        console.info('Meow')
    }
}
```

Все эти возможности типобезопасны. Если вы забудете реализовать метод или свойство или реализуете их некорректно, TypeScript придет на помощь (рис. 5.3).


**Рис. 5.3.** TypeScript выбрасывает ошибку, когда вы забываете реализовать метод

### Реализация интерфейсов против расширения абстрактных классов

Реализация интерфейса действительно схожа с расширением абстрактного класса. Разница в том, что интерфейсы более обобщены и легковесны.

Интерфейс — это способ моделирования формы. На уровне значений это объект, массив, функция, класс или экземпляр класса. Он не формирует JavaScript-код и существует только в процессе компиляции.

Абстрактный класс может смоделировать только класс. Он формирует код среды выполнения, являющийся, как вы уже догадались, классом. Он может иметь конструкторы, обеспечивать предустановленные реализации и устанавливать модификаторы доступа для свойств и методов. Интерфейсы не могут делать ничего из перечисленного.

Что из них лучше, зависит от конкретного случая. Когда реализация разделяется сразу несколькими классами, используются абстрактные классы. Когда нужно просто сообщить, что «этот класс является `T`», используйте интерфейс.

## Классы структурно типизированы

Как и любые другие типы, TypeScript сравнивает классы по их структуре, а не по имени. Класс совместим с любым другим типом, разделяющим его форму, включая старый привычный объект, определяющий те же свойства или методы, что и класс. Это важно помнить, если вы пришли из C#, Java, Scala или другого языка, типизирующего классы номинально. Это значит, что если у вас есть функция, получающая `Zebra`, и вы передаете ей `Poodle`, TypeScript не станет возражать:

```
class Zebra {
    trot() {
        // ...
    }
}

class Poodle {
    trot() {
        // ...
    }
}
```

```
function ambleAround(animal: Zebra) {
    animal.trot()
}

let zebra = new Zebra
let poodle = new Poodle

ambleAround(zebra)    // OK
ambleAround(poodle)   // OK
```

Как известно, зебра совсем не пудель, но с точки зрения функции они взаимозаменяемы. Важно, что они реализуют `.trot`. В языках, типизирующих классы номинально, этот код вызовет ошибку. Но TypeScript типизирован структурно.

Исключением из этого правила будут классы с полями `private` или `protected`: если при проверке совместимости формы и класса этот класс имеет такие поля, а форма не является экземпляром этого класса или его подклассом, то она окажется с ним несовместима:

```
class A {
    private x = 1
}
class B extends A {}
function f(a: A) {}
f(new A)    // OK
f(new B)    // OK

f({x: 1})   // Ошибка TS2345: аргумент типа '{x: number}' несовместим
            // с параметром типа 'A'. Свойство 'x' является
            // private в типе 'A', но не в типе '{x: number}'.
```

## Классы объявляют и значения, и типы

Бóльшая часть того, что вы можете выразить в TypeScript, оказывается либо значением, либо типом:

```
// значения
let a = 1999
function b() {}
```

```
// типы
type a = number
interface b {
    (): void
}
```

Типы и значения в TypeScript относятся к разным областям имен. В зависимости от того, как вы используете выражение (`a` или `b` в этом примере), TypeScript решает, обрабатывать его как тип или как значение.

```
// ...
if (a + 1 > 3) //... // TypeScript выводит из контекста,
                     // что вы подразумеваете значение a
let x: a = 3         // TypeScript выводит из контекста,
                     // что вы подразумеваете тип a
```

Контекстное прояснение выражений очень удобно и позволяет делать крутые вещи вроде реализации типов компаньонов (см. подраздел «Паттерн объект-компаньон» на с. 176).

Классы и перечисления являются особенными. Они уникальны благодаря тому, что генерируют как тип в пространстве типов, так и значение в пространстве значений:

```
class C {}
let c: C ❶
    = new C ❷

enum E {F, G}
let e: E ❸
    = E.F ❹
```

❶ В этом контексте `C` относится к типу экземпляра класса `C`.

❷ В этом контексте `C` относится к `C`-значению.

❸ В этом контексте `E` относится к типу перечисления `E`.

❹ В этом контексте `E` относится к `E`-значению.

При работе с классами нужно иметь возможность сказать: «Эта переменная должна быть экземпляром этого класса», и то же самое касается перечислений («Эта переменная должна быть членом этого перечисления»).

Поскольку классы и перечисления генерируют типы на уровне типов, они позволяют легко выразить связь «является тем-то»[1] (*is-a*).

Также нужен способ представить класс в среде выполнения, чтобы инстанцировать его с `new`, вызывать для него статические методы, производить с ними метапрограммирование и оперировать с ними посредством `instanceof`. То есть класс должен генерировать и значение тоже.

В предыдущем примере `C` относится к экземпляру класса `C`. Как описать класс `C`? Мы используем ключевое слово `typeof` (оператор типа в TypeScript, который дублирует `typeof` уровня значений в JavaScript, но для типов).

Создадим класс `StringDatabase` — простейшую базу данных:

```
type State = {
    [key: string]: string
}

class StringDatabase {
    state: State = {}
    get(key: string): string | null {
        return key in this.state ? this.state[key] : null
    }
    set(key: string, value: string): void {
        this.state[key] = value
    }
    static from(state: State) {
        let db = new StringDatabase
        for (let key in state) {
            db.set(key, state[key])
        }
        return db
    }
}
```

Какие типы генерирует эта декларация класса? Тип экземпляра `StringDataBase`:

[1] TypeScript структурно типизирован, и связь для классов больше похожа на «выглядит как» (looks-like) — любой объект, реализующий ту же форму, что и класс, будет совместим с типом этого класса.

```
interface StringDatabase {
    state: State
    get(key: string): string | null
    set(key: string, value: string): void
}
```

И тип конструктора `typeof StringDatabase`:

```
interface StringDatabaseConstructor {
    new(): StringDatabase
    from(state: State): StringDatabase
}
```

То есть `StringDatabaseConstructor` имеет один метод `.from`, а использование в конструкторе `new` дает экземпляр `StringDatabase`. Вместе два интерфейса моделируют обе стороны класса — конструктор и экземпляр класса.

Эта часть `new()` называется сигнатурой конструктора, позволяющей на языке TypeScript сказать, что данный тип может быть инстанцирован с оператором `new`. Вот наилучший вариант описания класса в структурно типизированном языке — «все, что может быть создано с `new`».

В этом случае конструктор не принимает никаких аргументов, но вы можете использовать его для объявления конструкторов, которые также принимают и аргументы. Например, мы обновляем `StringDatabase` для получения опционального изначального состояния:

```
class StringDatabase {
    constructor(public state: State = {}) {}
    // ...
}
```

Затем мы можем типизировать сигнатуру конструктора `StringDatabase` как:

```
interface StringDatabaseConstructor {
    new(state?: State): StringDatabase
    from(state: State): StringDatabase
}
```

Таким образом, декларация класса генерирует не только выражения на уровнях типов и значений, но и генерирует два выражения на уровне типов: одно, представляющее экземпляр класса, и другое, представляющее сам конструктор класса (доступный с помощью оператора типа `typeof`).

# Полиморфизм

Подобно функциям и типам, классы и интерфейсы имеют богатую поддержку параметров обобщенных типов, включая предустановки и ограничения. Область обобщенного типа можно расширить до всего класса или интерфейса или сузить до конкретного метода:

```
class MyMap<K, V> { ❶
    constructor(initialKey: K, initialValue: V) { ❷
        // ...
    }
    get(key: K): V { ❸
        // ...
    }
    set(key: K, value: V): void {
        // ...
    }
    merge<K1, V1>(map: MyMap<K1, V1>): MyMap<K | K1, V | V1> { ❹
        // ...
    }
    static of<K, V>(k: K, v: V): MyMap<K, V> { ❺
        // ...
    }
}
```

❶ При объявлении `class` привязывайте обобщенные типы к диапазону класса. Здесь `K` и `V` доступны для каждого метода и свойства экземпляра в `MyMap`.

❷ Заметьте, что вы не можете объявить обобщенные типы в `constructor`. Вместо этого переместите их декларацию в декларацию `class`.

❸ Используйте обобщенные типы с диапазоном класса везде внутри класса.

❹ Методы экземпляра имеют доступ к обобщенным типам уровня класса и также могут объявлять свои собственные обобщенные типы поверх них. `.merge` использует обобщенные типы уровня класса `K` и `V` и также объявляет два своих собственных обобщенных типа `K1` и `V1`.

❺ Статические методы не имеют доступа к обобщенным типам их класса, так же как на уровне значений они не имеют доступа к переменным экземпляра их класса. У `of` нет доступа к `K` и `V`, объявленным в ❶, поэтому он объявляет свои собственные обобщенные типы `K` и `V`.

Также можно привязывать обобщенные типы к интерфейсам:

```
interface MyMap<K, V> {
    get(key: K): V
    set(key: K, value: V): void
}
```

И явно привязать конкретные типы к обобщенным типам как к функциям или позволить TypeScript вывести их за вас:

```
let a = new MyMap<string, number>('k', 1) // MyMap<string, number>
let b = new MyMap('k', true) // MyMap<string, boolean>

a.get('k')
b.set('k', false)
```

# Примеси

В JavaScript и TypeScript нет ключевых слов `trait` или `mixin`, но их можно реализовать. Они оба симулируют наследование классов и осуществляют ролевое программирование — стиль программирования, где вместо фразы «это является `Shape`» мы говорим «это можно измерить» или «у этого есть четыре стороны».

Создадим реализацию примеси.

Примесь — это паттерн, позволяющий примешивать поведения и свойства в класс. Она может:

- Иметь состояние (например, свойства экземпляра).
- Предоставлять только конкретные методы (не абстрактные).
- Иметь конструкторы, которые вызываются в том же порядке, в каком их классы были примешаны.

В TypeScript нет встроенной концепции примесей, но мы легко реализуем их сами. Спроектируем отладочную библиотеку EZDebug для классов TypeScript, которая позволит регистрировать информацию о том, какие классы ее используют, чтобы инспектировать их при выполнении:

```
class User {
    // ...
}
User.debug() // вычисляется как 'User({"id": 3, "name": "Emma Gluzman"})'
```

Со стандартным интерфейсом `.debug` пользователи смогут отлаживать что угодно. Смоделируем его с примесью `withEZDebug` — функцией, получающей конструктор класса и возвращающей конструктор класса:

```
type ClassConstructor = new(...args: any[]) => {} ❶

function withEZDebug<C extends ClassConstructor>(Class: C) { ❷
    return class extends Class { ❸
        constructor(...args: any[]) { ❹
            super(...args) ❺
        }
    }
}
```

❶ Объявляем тип `ClassConstructor`. Указываем, что конструктор — это нечто созданное с `new` и способное получать любое число аргументов любого типа[1].

❷ Объявляем примесь `withEZDebug` с одним параметром типа `C`. Он должен по меньшей мере быть конструктором класса, который мы приведем в исполнение при помощи `extends`. Путь TypeScript выведет возвращаемый тип `withEZDebug` — пересечение `C` и нового анонимного класса.

❸ Поскольку примесь является функцией, получающей конструктор и возвращающей конструктор, мы возвращаем анонимный конструктор класса.

❹ Конструктор класса должен получить как минимум те же аргументы, что и передаваемый класс. Но мы не знаем, какой класс будет передан, и нужно сохранять конструктор обобщенным.

❺ Наконец, поскольку анонимный класс расширяет другой класс, для корректной взаимосвязи потребуется вызвать и конструктор `Class`.

Как и в обычных классах JavaScript, если у вас в `constructor` больше нет логики, то вы можете опустить строки ❹ и ❺. В примере с `withEZDebug` мы не собираемся добавлять логику.

Теперь, когда с рутинным кодом покончено, настало время отладочной магии. При вызове `.debug` мы будем регистрировать имя конструктора класса и значение экземпляра:

---

[1] Заметьте, что TypeScript здесь придирчив: тип аргументов типа конструктора должен быть `any[]` (не `void`, `unknown[]` и пр.), чтобы мы могли его расширить.

```
type ClassConstructor = new(...args: any[]) => {}

function withEZDebug<C extends ClassConstructor>(Class: C) {
    return class extends Class {
        debug() {
            let Name = Class.constructor.name
            let value = this.getDebugValue()
            return Name + '(' + JSON.stringify(value) + ')'
        }
    }
}
```

Но стоп! Подумайте, как убедиться, что класс реализует метод `.getDebugValue`, чтобы можно было его вызвать.

Ответ в том, что вместо принятия любого привычного класса мы используем обобщенный тип для уверенности, что переданный в `withEZDebug` класс определяет метод `.getDebugValue`:

```
type ClassConstructor<T> = new(...args: any[]) => T ❶

function withEZDebug<C extends ClassConstructor<{
    getDebugValue(): object ❷
}>>(Class: C) {
    // ...
}
```

❶ Добавляем обобщенный тип в `ClassConstructor`.

❷ Привязываем к `ClassConstructor` форму `C`, гарантируя, что переданный нами в `withEZDebug` конструктор как минимум определяет метод `.getDebugValue`.

Вот и все! Итак, как же использовать эту невероятную отладочную утилиту? Вот так:

```
class HardToDebugUser {
    constructor(
        private id: number,
        private firstName: string,
        private lastName: string
    ) {}
    getDebugValue() {
        return {
```

```
            id: this.id,
            name: this.firstName + ' ' + this.lastName
        }
    }
}

let User = withEZDebug(HardToDebugUser)
let user = new User(3, 'Emma', 'Gluzman')
user.debug() // вычислется как 'User({"id": 3, "name": "Emma Gluzman"})'
```

Круто, правда? Вы можете применить к классу столько примесей, сколько пожелаете, чтобы создать класс с более разнообразным поведением, причем типобезопасным способом. Примеси помогают инкапсулировать поведение класса, а также служат выразительным способом создать его переиспользуемый вариант.

# Декораторы

Декораторы — это экспериментальная возможность TypeScript, предоставляющая синтаксис для метапрограммирования с классами, методами классов, свойствами и параметрами методов. Это синтаксис, который работает посредством вызова функции для декорируемого элемента.

**TSC-ФЛАГ: EXPERIMENTALDECORATORS**

Поскольку декораторы все еще экспериментальная возможность, позднее они могут стать несовместимыми с прежними версиями или и вовсе быть удалены. Они также регулируются флагом TSC. Если вы хотите их попробовать, то установите `experimentalDecorators: true` в tsconfig.json и продолжайте изучение раздела.

Начнем разбор принципа работы декораторов со следующего примера:

```
@serializable
class APIPayload {
    getValue(): Payload {
        // ...
    }
}
```

Декоратор класса `@serializable` оборачивает класс `APIPayload` и опционально возвращает новый класс на замену. Без декораторов можно сделать то же самое так:

```
let APIPayload = serializable(class APIPayload {
    getValue(): Payload {
        // ...
    }
})
```

Для каждого типа декоратора TypeScript требует наличия в области действия функции с заданным именем и необходимой сигнатурой (табл. 5.1).

**Таблица 5.1.** Ожидаемые сигнатуры типов для различных видов функций декораторов

| Что декорируем | Ожидаемая сигнатура типа |
|---|---|
| *Класс* | `(Constructor: {new(...any[]) => any}) => any` |
| *Метод* | `(classPrototype: {}, methodName: string, descriptor: PropertyDescriptor) => any` |
| *Статический метод* | `(Constructor: {new(...any[]) => any}, methodName: string, descrip tor: PropertyDescriptor) => any` |
| *Параметр метода* | `(classPrototype: {}, paramName: string, index: number) => void` |
| *Параметр статического метода* | `(Constructor: {new(...any[]) => any}, paramName: string, index: number) => void` |
| *Свойство* | `(classPrototype: {}, propertyName: string) => any` |
| *Свойство статического метода* | `(Constructor: {new(...any[]) => any}, propertyName: string) => any` |
| *Свойство геттера/сеттера* | `(classPrototype: {}, propertyName: string, descriptor: PropertyDescriptor) => any` |
| *Статическое свойство геттера/сеттера* | `(Constructor: {new(...any[]) => any}, propertyName: string, descriptor: PropertyDescriptor) => any` |

В TypeScript нет встроенных декораторов: их можно реализовать самостоятельно или установить из *NPM*. Реализация для каждого вида декоратора — для классов, методов, свойств и параметров функций — это регулярная функция, соответствующая определенной сигнатуре в зави-

симости от того, что она декорирует. Например, декоратор `@serializable` может выглядеть так:

```
type ClassConstructor<T> = new(...args: any[]) => T ❶

function serializable<

    T extends ClassConstructor<{
      getValue(): Payload ❷
    }>
>(Constructor: T) { ❸
    return class extends Constructor { ❹
        serialize() {
            return this.getValue().toString()
        }
    }
}
```

❶ `new()` — это способ структурной типизации конструктора класса в TypeScript. А для конструктора класса, который может быть расширен (с помощью `extends`), TypeScript требует, чтобы его аргументы были типизированы с распространением `any: new(…any[])`.

❷ `@serializable` может декорировать любой класс, чьи экземпляры реализуют метод `.getValue`, возвращающий `PayLoad`.

❸ Декораторы классов — это функции, получающие один аргумент — класс. Если функция декоратора вернет класс (как в примере), то он заменит декорируемый класс в среде выполнения. В противном случае он вернет оригинальный класс.

❹ Для декорирования класса мы возвращаем класс, который его расширяет, и попутно добавляем метод `.serialize`.

Что происходит, когда мы пытаемся вызвать `.serialize`?

```
let payload = new APIPayload
let serialized = payload.serialize()    // Ошибка TS2339: свойство
                                        // 'serialize' не существует
                                        // в типе 'APIPayload'.
```

TypeScript предполагает, что декоратор не изменяет форму того, что декорирует (не добавляются и не удаляются методы и свойства). В процессе

компиляции он проверяет, чтобы возвращаемый вами класс был совместим с переданным классом, но на момент написания книги TypeScript не отслеживает расширения, производимые вами в декораторах.

Пока декораторы не стали более проработанными, я рекомендую избегать их применения и придерживаться регулярных функций:

```
let DecoratedAPIPayload = serializable(APIPayload)
let payload = new DecoratedAPIPayload
payload.serialize()                          // string
```

Дополнительную информацию о декораторах можно найти в официальной документации (https://www.typescriptlang.org/docs/handbook/decorators.html).

## Имитация финальных классов

Хотя TypeScript и не поддерживает ключевое слово `final` для классов и методов, его можно легко имитировать. Если у вас нет опыта работы с объектно-ориентированными языками, то `final` — это ключевое слово, которое в некоторых из них сообщает, что класс не может быть расширен, а метод — переопределен.

Для имитации `final` классов в TypeScript можно использовать приватные конструкторы:

```
class MessageQueue {
    private constructor(private messages: string[]) {}
}
```

Когда `constructor` помечен как `private`, вы не можете использовать в классе `new` или расширять его:

```
class BadQueue extends MessageQueue {}    // Ошибка TS2675: невозможно
                                          // расширить класс
                                          // 'MessageQueue'.
                                          // Конструктор класса
                                          // отмечен как приватный.

new MessageQueue([])                 // Ошибка TS2673: конструктор
                                     // класса 'MessageQueue'
                                     // является приватным и доступен
                                     // только внутри декларации класса.
```

Помимо препятствия расширению класса (что нам и нужно) приватные конструкторы также запрещают непосредственное инстанцирование этого класса (а этого не надо). Как сохранить первое ограничение и избавиться от второго? Легко:

```
class MessageQueue {
    private constructor(private messages: string[]) {}
    static create(messages: string[]) {
        return new MessageQueue(messages)
    }
}
```

API `MessageQueue` несколько изменился, но зато успешно предотвращено его расширение в процессе компиляции:

```
class BadQueue extends MessageQueue {} // Ошибка TS2675: невозможно
                                       // расширить класс
                                       // 'MessageQueue'. Конструктор
                                       // класса отмечен как приватный.

MessageQueue.create([])                // MessageQueue
```

# Паттерны проектирования

Глава об объектно-ориентированном программировании была бы неполной без обсуждения реализации хотя бы пары паттернов проектирования, не так ли?

## Паттерн фабрика

*Паттерн фабрика* — это способ создать объект, на базе которого можно формировать объекты схожего типа.

Построим обувную фабрику. Начнем с определения типа `Shoe` и нескольких видов ботинок:

```
type Shoe = {
    purpose: string
}

class BalletFlat implements Shoe {
```

```
    purpose = 'dancing'
}

class Boot implements Shoe {
    purpose = 'woodcutting'
}

class Sneaker implements Shoe {
    purpose = 'walking'
}
```

Заметьте, что в этом примере используется `type`, но мы могли бы с тем же успехом использовать `interface`.

Приступим к созданию обувной фабрики:

```
let Shoe = {
    create(type: 'balletFlat' | 'boot' | 'sneaker'): Shoe { ❶
        switch (type) { ❷
            case 'balletFlat': return new BalletFlat
            case 'boot': return new Boot
            case 'sneaker': return new Sneaker
        }
    }
}
```

❶ Использование типа объединения для `type` помогает сделать `.create` максимально безопасным, предотвращая возможную передачу потребителями неверного `type` при компиляции.

❷ Переключение на `type` позволяет TypeScript проследить, что каждый тип `Shoe` обработан.

В этом примере мы использовали паттерн объект-компаньон (см. подраздел «Паттерн объект-компаньон» на с. 176) для определения типа `Shoe` и значения `Shoe` с одинаковым именем (в TypeScript пространства имен значений и имен типов не пересекаются), указав тем самым, что значение предоставляет методы для оперирования в типе. Использование фабрики начинается с вызова `.create`:

```
Shoe.create('boot') // Shoe
```

Вуаля! У нас есть паттерн фабрика. Заметьте, что можно пойти дальше и указать в сигнатуре типа `Shoe.create`, что передача `'boot'` будет давать `Boot`, `'sneaker'` — давать `Sneaker` и т. д., но это бы разрушило абстракцию, предоставляемую паттерном (потребители не должны знать, какой конкретный класс они получат назад, но должны видеть, что класс соответствует интерфейсу).

## Паттерн строитель

Строитель является способом отделить конструкцию объекта от его фактической реализации. Если вы пользовались JQuery или структурами данных ES6 вроде `Map` или `Set`, то этот стиль API должен быть вам знаком. Он выглядит так:

```
new RequestBuilder()
    .setURL('/users')
    .setMethod('get')
    .setData({firstName: 'Anna'})
    .send()
```

Как мы реализуем `RequestBuilder`? Легко — мы начнем с чистого класса:

```
class RequestBuilder {}
```

Сначала добавим метод `.setURL`:

```
class RequestBuilder {

    private url: string | null = null ❶

    setURL(url: string): this { ❷
       this.url = url
       return this
    }
}
```

❶ Отслеживаем *URL*, установленный пользователем в переменной приватного экземпляра `url`, которую мы инициализируем с `null`.

❷ Возвращаемым типом `setURL` будет `this` (см. раздел «Использование this в качестве возвращаемого типа» на с. 117), являющийся конкретным экземпляром `RequestBuilder`, в котором пользователь вызвал `setURL`.

Теперь добавим другие методы из примера:

```
class RequestBuilder {

    private data: object | null = null
    private method: 'get' | 'post' | null = null
    private url: string | null = null

    setMethod(method: 'get' | 'post'): this {
        this.method = method
        return this
    }
    setData(data: object): this {
        this.data = data
        return this
    }
    setURL(url: string): this {
        this.url = url
        return this
    }

    send() {
        // ...
    }
}
```

На этом все.

Традиционный паттерн строитель небезопасен: мы можем вызвать `.send` до того, как установим метод, URL или данные, что приведет к исключению среды выполнения (вспомните, что это плохой вид исключения). Обратитесь к упражнению 4 за дополнительными идеями по улучшению этого представления.

## Итоги

Вы изучили классы TypeScript: их объявление, наследование и интерфейсы. Узнали, как помечать классы `abstract`, чтобы они не могли быть инстанцированы, добавлять поле или метод в класс со `static` и в экзем-

пляр без `static`, контролировать доступ к полю или методу при помощи модификаторов видимости `private`, `protected` и `public`, а также исключать перезаписываемость поля с помощью модификатора `readonly`. Рассмотрели безопасное использование `this` и `super`, изучили, что значит для классов быть одновременно и значениями, и типами, а также поняли разницу между псевдонимами типов и интерфейсами и основы слияния деклараций и использования обобщенных типов в классах. Вы увидели несколько продвинутых паттернов для работы с классами: примеси, декораторы и имитирование `final`, а также два стандартных паттерна проектирования и их применение в классах.

## Упражнения к главе 5

1. В чем отличие класса от интерфейса?
2. Когда вы помечаете конструктор класса как `private`, это означает, что вы не можете инстанцировать или расширять класс. Что происходит, когда вместо этого вы помечаете его как `protected`? Поработайте с этим в редакторе и постарайтесь разобраться.
3. Расширьте разработанную на с. 139 реализацию паттерна фабрика, чтобы сделать ее безопаснее ценой некоторой утраты абстрактности. Обновите реализацию, чтобы потребитель во время компиляции знал, что вызов `Shoe.create('boot')` возвращает `Boot`, а вызов `Shoe.create('balletFlat')` — `BalletFlat` (вместо ситуации, при которой оба возвращают `Shoe`). Подсказка: обратитесь к подразделу «Типы перегруженных функций» на с. 83.
4. (Сложно.) Подумайте о том, как спроектировать безопасный паттерн строитель. Расширьте паттерн, описанный на с. 141:

   а) чтобы он при компиляции гарантировал, что никто не может вызвать `.send` до установки *URL* и метода. Станет ли легче гарантировать это, если вы принудите пользователя вызывать методы в определенном порядке? (Подсказка: что вы можете вернуть вместо `this`?);

   б) (Очень сложно.) Как бы вы изменили структуру, если бы хотели сохранить старые гарантии, но при этом позволить пользователям вызывать методы в любом порядке? (Подсказка: какая из возможностей TypeScript добавляет (`''add''`) возвращаемый тип метода в тип `this` после каждого вызова метода?)

ГЛАВА 6

# Продвинутые типы

Система типов TypeScript, признанная во всем мире, своими возможностями удивляет даже Haskell-программистов. Как вы уже знаете, она не только выразительна, но и легка в использовании: ограничения типов и связи в ней кратки, понятны и в большинстве случаев выводятся автоматически.

Моделирование таких элементов динамического JavaScript, как прототипы, привязанные `this`, перегрузки функций и вечно меняющиеся объекты, требует настолько богатой системы типов и их операторов, которую даже Бэтмен взял бы на вооружение.

Я начну эту главу с глубокого погружения в темы подтипов, совместимости, вариантности, случайной величины и расширения. Затем более детально раскрою особенности проверки типов на основе потока команд, включая уточнение и тотальность. Далее продемонстрирую некоторые продвинутые особенности программирования на уровне типов: подключение и отображение типов объектов, использование условных типов, определение защит типов и запасные решения вроде утверждений типов и утверждений явного присваивания. В заключение я познакомлю вас с продвинутыми паттернами для повышения безопасности типов: паттерном объект-компаньон, улучшением интерфейса для кортежей, имитированием номинальных типов и безопасным расширением прототипа.

## Связи между типами

Рассмотрим связи в TypeScript подробнее.

### Подтипы и супертипы

Мы уже затрагивали совместимость в разделе «О типах» на с. 34, поэтому сразу углубимся в эту тему, начиная с определения подтипа.

**ПОДТИП**

Если у вас есть два типа, *A* и *B*, и при этом *B* является подтипом *A*, то вы можете безопасно использовать *B* везде, где требуется *A* (рис. 6.1).

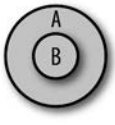


**Рис. 6.1.** B является подтипом *A*

Вернитесь к рис. 3.1 и увидите встроенные в TypeScript связи подтипов.

- Массив является подтипом объекта.
- Кортеж является подтипом массива.
- Все является подтипом `any`.
- `never` является подтипом всего.
- Класс `Bird`, расширяющий класс `Animal`, — это подтип класса `Animal`.

Согласно определению, которое я только что дал для подтипа, это значит, что:

- Везде, где нужен объект, можно использовать массив.
- Везде, где нужен массив, можно использовать кортеж.
- Везде, где нужен `any`, можно использовать объект.
- Везде можно использовать `never`.
- Везде, где нужен `Animal`, можно использовать `Bird`.

Супертип — это противоположность подтипа.

**СУПЕРТИП**

Если у вас есть два типа, *A* и *B*, и при этом *B* является супертипом *A*, то вы можете безопасно использовать *A* везде, где требуется *B* (рис. 6.2).

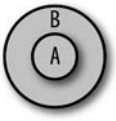


**Рис. 6.2.** B является супертипом A

И снова исходя из схемы на рис. 3.1:

- Массив является супертипом кортежа.
- Объект является супертипом массива.
- `Any` является супертипом всего.
- `Never` не является чьим-либо супертипом.
- `Animal` — это супертип `Bird`.

Это просто противоположный подтипам принцип и ничего более.

## Вариантность

Для большинства типов достаточно легко понять, является ли некий тип `A` подтипом `B`. Для простых типов вроде `number`, `string` и др. можно обратиться к схеме на рис. 3.1 или самостоятельно определить, что `number`, содержащийся в объединении `number | string`, является подтипом этого объединения.

Но есть более сложные типы, например обобщенные. Подумайте над такими вопросами:

- Когда `Array<A>` является подтипом `Array<B>`?
- Когда форма `A` является подтипом формы `B`?
- Когда функция `(a: A) => B` является подтипом функции `(c: C) => D`?

Правила подтипизации для типов, содержащих другие типы (то есть имеющих параметры типа вроде `Array<A>`, формы с полями вроде `{a: number}` или функции вроде `(a: A) => B`), осмысливать уже сложнее, потому что они не согласованы в разных языках программирования.

Чтобы облегчить чтение последующих правил, я представлю несколько элементов синтаксиса, который не работает в TypeScript (не беспокойтесь, он не математический):

- `A <: B` означает, что «`A` является подтипом того же, что и тип `B`»;
- `A >: B` означает, что «`A` является супертипом того же, что и тип `B`».

#### Вариантность формы и массива

Понять, почему языки не согласуются в правилах подтипизации сложных типов, поможет пример с формой, которая описывает пользователя в приложении. Представим ее посредством пары типов:

```
// Существующий пользователь, переданный с сервера.
type ExistingUser = {
    id: number
    name: string
}

// Новый пользователь, еще не сохраненный на сервере.
type NewUser = {
    name: string
}
```

Предположим, что стажер в вашей компании получил задание написать код для удаления пользователя. Начинает он со следующего:

```
function deleteUser(user: {id?: number, name: string}) {
    delete user.id
}

let existingUser: ExistingUser = {
    id: 123456,
    name: 'Ima User'
}

deleteUser(existingUser)
```

`deleteUser` получает объект типа `{id?: number, name: string}` и передает в него `existingUser` типа `{id: number, name: string}`. Обратите внимание, что тип свойства `id (number)` — это подтип ожидаемого типа `(number | undefined)`. Следовательно, весь объект `{id: number, name: string}` — это подтип `{id?: number, name: string}`, поэтому TypeScript это допускает.

Видите ли вы какие-либо проблемы с безопасностью? Есть одна: после передачи `ExistingUser` в `deleteUser` TypeScript не знает, что `id` пользователя был удален, поэтому если вы прочитаете `existingUser.id` после его удаления `deleteUser(existingUser)`, то TypeScript по-прежнему будет считать, что `existingUser.id` имеет тип `number`.

Очевидно, что использование типа объекта там, где ожидается его супертип, небезопасно. Так почему же TypeScript это допускает? Суть в том, что он не задумывался как абсолютно безопасный. Его система типов стремится перехватывать реальные ошибки и делать их наглядными для программистов любого уровня. Поскольку деструктивные обновления (вроде удаления свойства) относительно редки на практике, TypeScript расслаблен и позволяет вам присвоить объект там, где ожидается его супертип.

А что насчет противоположного случая: можно ли присвоить объект там, где ожидается его подтип?

Добавим новый тип для старого пользователя, а затем удалим пользователя с этим типом (представьте, что вы добавляете типы в код, который написал ваш коллега):

```
type LegacyUser = {
    id?: number | string
    name: string
}

let legacyUser: LegacyUser = {
    id: '793331',
    name: 'Xin Yang'
}

deleteUser(legacyUser)  // Ошибка TS2345: аргумент типа 'LegacyUser'
                        // несовместим с параметром типа
                        // '{id?: number |undefined, name: string}'.
                        // Тип 'string' несовместим с типом 'number |
                        // undefined'.
```

Когда вы передаете форму со свойством, чей тип является супертипом ожидаемого типа, TypeScript ругается. Все потому, что `id` — это `string | number | undefined`, а `deleteUser` обрабатывает только тот случай, где `id` является `number | undefined`.

Ожидая форму, вы можете передать тип с типами свойств, которые `<:` ожидаемых типов, но не можете передать форму без типов свойств, являющихся супертипами их ожидаемых типов. Когда речь идет о типах, мы говорим: «TypeScript-формы (объекты и классы) являются ковариантными в типах их свойств». То есть, чтобы объект `A` мог быть присвоен объекту `B`, каждое его свойство должно быть `<:` соответствующего ему свойства в `B`.

Ковариантность — это один из четырех видов вариантности:

*Инвариантность*

Нужен конкретно `T`.

*Ковариантность*

Нужен `<:T`.

*Контравариантность*

Нужен `>:T`.

*Бивариантность*

Устроит либо `<:T`, либо `>:T`.

В TypeScript каждый сложный тип является ковариантным в своих членах — объектах, классах, массивах и возвращаемых типах функций, за одним исключением: типы параметров функций контравариантны.

Не все языки применяют такое же конструктивное решение. В одних объекты являются инвариантными в типах свойств, потому что ковариантные типы свойств могут вести к небезопасному поведению. Другие языки имеют разные правила для изменяемых и неизменяемых объектов (попробуйте порассуждать об этом самостоятельно). Третьи, вроде Scala, Kotlin и Flow, даже имеют явный синтаксис, позволяющий программистам определять вариантность типов данных.

Создатели TypeScript предпочли баланс: инвариантность объектов в типах свойств повышает безопасность, но усложняет использование системы типов, поскольку заставляет вас запрещать то, что на деле безопасно (например, если не удалять `id` в `deleteUser`, передача объекта, являющегося супертипом ожидаемого типа, все равно будет безопасной).

#### Вариантность функции

Рассмотрим несколько примеров.

Функция `A` является подтипом функции `B`, если `A` имеет такую же или меньшую арность (число параметров), чем `B`, и:

1. Тип `this`, принадлежащий `A`, либо не определен, либо `>:` типа `this`, принадлежащего `B`.
2. Каждый из параметров `A >:` соответствующего параметра в `B`.
3. Возвращаемый тип `A <:` возвращаемого типа `B`.

Обратите внимание, что для того, чтобы функция `A` могла быть подтипом функции `B`, ее тип `this` и параметры должны быть `>:` встречных частей в `B`, в то время как ее возвращаемый тип должен быть `<:`. Почему происходит разворот условия? Почему не работает просто условие `<:` для каждого компонента (типа `this`, типов параметров и возвращаемого типа), как в случае с объектами, массивами, объединениями и т. д.?

Начнем с определения трех типов (вместо `class` можно использовать другие типы, где `A :< B <: C`):

```
class Animal {}
class Bird extends Animal {
    chirp() {}
}
class Crow extends Bird {
    caw() {}
}
```

Итак, `Crow <: Bird <: Animal`.

Определим функцию, получающую `Bird` и заставляющую ее чирикать:

```
function chirp(bird: Bird): Bird {
    bird.chirp()
    return bird
}
```

Пока все хорошо. Что TypeScript позволяет вам передать в `chirp`?

```
chirp(new Animal)    // Ошибка TS2345: аргумент типа 'Animal'
chirp(new Bird)      // несовместим с параметром типа 'Bird'.
chirp(new Crow)
```

Экземпляр `Bird` (как параметр `chirp` типа `bird`) или экземпляр `Crow` (как подтип `Bird`). Передача подтипа работает, как и ожидалось.

Создадим новую функцию. На этот раз ее параметр будет функцией:

```
function clone(f: (b: Bird) => Bird): void {
    // ...
}
```

`clone` требуется функция `f`, получающая `Bird` и возвращающая `Bird`. Какие типы функций можно передать для `f` безопасно? Очевидно, функцию, получающую и возвращающую `Bird`:

```
function birdToBird(b: Bird): Bird {
    // ...
}
clone(birdToBird) // OK
```

Что насчет функции, получающей `Bird`, но возвращающей `Crow` или `Animal`?

```
function birdToCrow(d: Bird): Crow {
    // ...
}
clone(birdToCrow) // OK

function birdToAnimal(d: Bird): Animal {
    // ...
}
clone(birdToAnimal)    // Ошибка TS2345: аргумент типа '(d: Bird) =>
                       // Animal' несовместим с параметром типа
                       // '(b: Bird) => Bird'.Тип 'Animal'
                       // несовместим с типом 'Bird'.
```

`birdToCrow` работает, как и ожидалось, но `birdToAnimal` выдает ошибку. Почему? Представьте, что реализация `clone` выглядит так:

```
function clone(f: (b: Bird) => Bird): void {
   let parent = new Bird
   let babyBird = f(parent)
   babyBird.chirp()
}
```

Передав функции `clone` функцию `f`, возвращающую `Animal`, мы не сможем вызвать в ней `.chirp`. Поэтому TypeScript должен убедиться, что переданная нами функция возвращает как минимум `Bird`.

Когда мы говорим, что функции ковариантны в их возвращаемых типах, это значит, что функция может быть подтипом другой функции, только если ее возвращаемый тип `<:` возвращаемого типа той функции.

Хорошо, а что насчет типов параметров?

```
function animalToBird(a: Animal): Bird {
    // ...
}
clone(animalToBird) // OK

function crowToBird(c: Crow): Bird {
    // ...
}
clone(crowToBird)    // Ошибка TS2345: аргумент типа '(c: Crow) =>
                     // Bird' несовместим с параметром типа '
                     // (b: Bird) => Bird'.
```

Чтобы функция была совместима с другой функцией, все ее типы параметров (включая `this`) должны быть `>:` соответствующих им параметров в другой функции. Чтобы понять почему, подумайте о том, как пользователь мог бы реализовать `crowToBird`, прежде чем передавать ее в `clone`?

```
function crowToBird(c: Crow): Bird {
    c.caw()
    return new Bird
}
```

**TSC-ФЛАГ: STRICTFUNCTIONTYPES**

Из-за наследования функции в TypeScript по умолчанию ковариантны в своих параметрах и типах `this`. Чтобы использовать более безопасное контрвариантное поведение, которое мы только что изучили, нужно активировать флаг `{"strictFunctionTypes": true}` в tsconfig.json.

Если вы уже используете `{"strict": true}`, то ничего дополнительно делать не нужно.

Теперь, если `clone` вызовет `crowToBird` с `new Bird`, мы получим исключение, поскольку `.caw` определен во всех `Crow`, но не во всех `Bird`.

Это означает, что функции контрвариантны в их параметрах и типах `this`. То есть функция может быть подтипом другой функции, только если каждый из ее параметров и тип `this` будут `>:` соответствующих им параметров в другой функции.

К счастью, эти правила не нужно заучивать. Просто вспомните о них, когда редактор выдаст красное подчеркивание при передаче вами некорректно типизированной функции куда-либо.

## Совместимость

Взаимосвязи подтипов и супертипов являются ключевой концепцией любого статически типизированного языка. Они также важны для понимания того, как работает совместимость (напомню, что совместимость относится к правилам TypeScript, определяющим возможность использования типа `A` там, где требуется тип `B`).

Когда TypeScript требуется ответить на вопрос: «Совместим ли тип `A` с типом `B`?», он следует простым правилам. Для не `enum`-типов — вроде массивов, логических типов, чисел, объектов, функций, классов, экземпляров классов и строк, включая типы литералов, — `A` совместим с `B`, если верно одно из условий.

1. `A <: B`.
2. `A` является `any`.

Правило 1 — это просто определение подтипа: если `A` является подтипом `B`, тогда везде, где нужен `B`, можно использовать `A`.

Правило 2 — это исключение из правила 1 для удобства взаимодействия с кодом JavaScript.

Для типов перечислений, созданных ключевыми словами `enum` или `const enum`, тип `A` совместим с перечислением `B`, если верно одно из условий.

1. `A` является членом перечисления `B`.
2. `B` имеет хотя бы один член типа `number`, а `A` является `number`.

Правило 1 в точности такое же, как и для простых типов (если `A` является членом перечисления `B`, тогда `A` имеет тип `B` и мы говорим, что `B <: B`).

Правило 2 необходимо для удобства работы с перечислениями, которые серьезно подрывают безопасность в TypeScript (см. подраздел «Enum» на с. 60), и я рекомендую их избегать.

### Расширение типов

Расширение типов — это ключ к пониманию работы системы вывода типов. TypeScript снисходителен при выполнении и скорее допустит ошибку при выводе более общего типа, чем при выводе максимально конкретного. Это упростит вам жизнь и сократит временны́е затраты на борьбу с замечаниями модуля проверки типов.

В главе 3 вы уже видели несколько примеров расширения типов. Рассмотрим другие.

Когда вы объявляете переменную как измененяемую (с `let` или `var`), ее тип расширяется от типа значения ее литерала до базового типа, к которому литерал принадлежит:

```
let a = 'x'                  // string
let b = 3                    // number
var c = true                 // boolean
const d = {x: 3}             // {x: number}

enum E {X, Y, Z}
let e = E.X                  // E
```

Это не касается неизменяемых деклараций:

```
const a = 'x'               // 'x'
const b = 3                 // 3
const c = true              // true

enum E {X, Y, Z}
const e = E.X               // E.X
```

Можно использовать явную аннотацию типа, чтобы не допустить его расширения:

```
let a: 'x' = 'x'            // 'x'
let b: 3 = 3                // 3
var c: true = true          // true
const d: {x: 3} = {x: 3}    // {x: 3}
```

Когда вы повторно присваиваете нерасширенный тип с помощью `let` или `var`, TypeScript расширяет его за вас. Чтобы это предотвратить, добавьте явную аннотацию типа в оригинальную декларацию:

```
const a = 'x'       // 'x'
let b = a           // string

const c: 'x' = 'x'  // 'x'
let d = c           // 'x'
```

Переменные, инициализированные как `null` или `undefined`, расширяются до `any`:

```
let a = null        // any
a = 3               // any
a = 'b'             // any
```

Но, когда переменная, инициализированная как `null` или `undefined`, покидает область, в которой была объявлена, TypeScript присваивает ей определенный тип:

```
function x() {
    let a = null    // any
    a = 3           // any
    a = 'b'         // any
    return a
}

x()                 // string
```

#### Тип const

Тип `const` помогает отказаться от расширения декларации типа. Используйте его как утверждение типа (см. подраздел «Утверждения типов» на с. 185):

```
let a = {x: 3}                  // {x: number}
let b: {x: 3}                   // {x: 3}
let c = {x: 3} as const         // {readonly x: 3}
```

`const` исключает расширение типа и рекурсивно отмечает его члены как `readonly` даже в глубоко вложенных структурах данных:

```
let d = [1, {x: 2}]             // (number | {x: number})[]
let e = [1, {x: 2}] as const    // readonly [1, {readonly x: 2}]
```

Используйте `as const`, когда хотите, чтобы TypeScript вывел максимально узкий тип.

#### Проверка лишних свойств

Расширение типов также фигурирует, когда TypeScript проверяет, является ли один тип объекта совместимым с другим типом объекта.

Типы объектов ковариантны в их членах (см. подраздел «Вариантность формы и массива» на с. 148). Но, если TypeScript будет следовать этому правилу без дополнительных проверок, могут возникнуть проблемы.

Например, рассмотрите объект `Options`, который можно передать в класс для его настройки:

```
type Options = {
    baseURL: string
    cacheSize?: number
    tier?: 'prod' | 'dev'
}

class API {
    constructor(private options: Options) {}
}

new API({
    baseURL: 'https://api.mysite.com',
    tier: 'prod'
})
```

Что произойдет теперь, если вы допустите ошибку в опции?

```
new API({
   baseURL: 'https://api.mysite.com',
   tierr: 'prod'   // Ошибка TS2345: аргумент типа '{tierr: string}'
})                 // несовместим с параметром типа 'Options'.
                   // Объектный литерал может определять только
                   // известные свойства, но 'tierr' не существует
                   // в типе 'Options'. Вы хотели написать 'tier'?
```

Это распространенный баг при работе в JavaScript, и хорошо, что TypeScript помогает его перехватить. Но если типы объектов ковариантны в их членах, как TypeScript его перехватывает?

Иначе говоря:

- Мы ожидали тип `{baseURL: string, cacheSize?: number, tier?: 'prod' | 'dev'}`.
- Мы передали тип `{baseURL: string, tierr: string}`.
- Переданный тип — это подтип ожидаемого типа, но TypeScript знал, что надо сообщить об ошибке.

Благодаря *проверке лишних свойств*, когда вы пытаетесь присвоить новый тип объектного литерала `T` другому типу, `U`, а в `T` есть свойства, которых нет в `U`, TypeScript сообщает об ошибке.

*Новый тип объектного литерала* — это тип, который TypeScript вывел из объектного литерала. Если этот объектный литерал использует утверждение типа (см. подраздел «Утверждения типов» на с. 185) или присвоен переменной, тогда новый тип расширяется до регулярного типа объекта и его новизна пропадает.

Попробуем сделать это определение более емким:

```
type Options = {
    baseURL: string
    cacheSize?: number
    tier?: 'prod' | 'dev'
}

class API {
    constructor(private options: Options) {}
}

new API({ ❶
    baseURL: 'https://api.mysite.com',
    tier: 'prod'
})

new API({ ❷
    baseURL: 'https://api.mysite.com',
    badTier: 'prod'    // Ошибка TS2345: аргумент типа '{baseURL:
})                     // string; badTier: string}' несовместим
                       // с параметром типа 'Options'.

new API({ ❸
    baseURL: 'https://api.mysite.com',
```

```
    badTier: 'prod'
} as Options)

let badOptions = { ❹
    baseURL: 'https://api.mysite.com',
    badTier: 'prod'
}
new API(badOptions)

let options: Options = { ❺
    baseURL: 'https://api.mysite.com',
    badTier: 'prod'    // Ошибка TS2322: тип '{baseURL: string;
}                      // badTier: string}'несовместим с типом
                       // 'Options'.
new API(options)
```

❶ Инстанцинируем API с `baseURL` и одно из двух опциональных свойств: `tier`. Все работает.

❷ Ошибочно прописываем `tier` как `badTier`. Объект опций, который мы передаем в `new API`, — новый (его тип выведен, он несовместим с переменной, и мы не делаем для него утверждения типа), поэтому при проверке лишних свойств TypeScript обнаруживает лишнее свойство `badTier` (которое определено в объекте опций, но не в типе `Options`).

❸ Делаем утверждение, что неверный объект опций имеет тип `Options`. TypeScript больше не рассматривает его как новый и делает заключение из проверки лишних свойств, что ошибок нет. Синтаксис `as T` описан в подразделе «Утверждения типов» на с. 185.

❹ Присваиваем объект опций к переменной `badOptions`. TypeScript больше не воспринимает его как новый и, произведя проверку лишних свойств, делает заключение, что ошибок нет.

❺ Когда мы явно типизируем `options` как `Options`, объект, присваиваемый нами `options`, является новым, поэтому TypeScript выполняет проверку лишних свойств и находит баг. Заметьте, что в этом случае проверка лишних свойств не производится, когда мы передаем `options` в `new API`, но она происходит, когда мы пытаемся присвоить объект опций к переменной `options`.

Эти правила не нужно заучивать. Это лишь внутренняя эвристика TypeScript для перехвата как можно большего числа багов. Просто

помните о них, если вдруг станет интересно, откуда TypeScript узнал про баг, который даже Иван — старожил вашей компании и по совместительству профессиональный цензор кода — не заметил.

## Уточнение

TypeScript производит символическое выполнение вывода типов. Модуль проверки типов использует инструкции потока команд (вроде `if`, `?`, `||` и `switch`) наряду с запросами типов (вроде `typeof`, `instanceof` и `in`), тем самым уточняя типы по ходу чтения кода, как это делал бы программист[1]. Однако эта удобная особенность поддерживается весьма небольшим количеством языков[2].

Представьте, что вы разработали в TypeScript API для определения правил *CSS* и ваш коллега хочет использовать его, чтобы установить *HTML*-элемент `width`. Он передает ширину, которую вы хотите позднее разобрать и сверить.

Сначала реализуем функцию для разбора строки *CSS* в значение и единицу измерения:

```
// Мы используем объединение строчных литералов для описания
// возможных значений, которые может иметь единица измерения CSS
type Unit = 'cm' | 'px' | '%'

// Перечисление единиц измерения
let units: Unit[] = ['cm', 'px', '%']
```

[1] Символическое выполнения — это форма программного анализа, в котором используется особая программа — символический исполнитель для запуска программы в том же режиме, что и в среде выполнения, но без присваивания определенных значений переменным. Каждая переменная моделируется как символ, чье значение ограничивается на время выполнения программы. Символическое выполнение позволяет говорить, например: «Эта переменная никогда не используется», «Эта функция никогда не возвращает» или «В положительной ветви инструкции `if` на строке 102 переменная `x` гарантированно не является `null`».

[2] TypeScript, Flow, Kotlin и Ceylon способны уточнять типы внутри блока кода. Это похоже на явную аннотацию в C или Java и сопоставление шаблонов в Haskell, Okami или Scala. Идея заключается во внедрении движка символического выполнения прямо в модуль проверки типов, чтобы давать ему обратную связь и анализировать программу приближенно к тому, как это делал бы программист.

```
// Проверить каждую ед. изм. и вернуть null, если не будет совпадений
function parseUnit(value: string): Unit | null {
    for (let i = 0; i < units.length; i++) {
        if (value.endsWith(units[i])) {
            return units[i]
        }
    }
    return null
}
```

Затем используем `parseUnit`, чтобы разобрать значение ширины, переданное пользователем. `width` может быт числом (возможно, в пикселах) или строкой с прикрепленными единицами измерения, или `null`, или `undefined`.

В этом примере мы несколько раз прибегаем к уточнению типа:

```
type Width = {
    unit: Unit,
    value: number
}

function parseWidth(width: number | string | null |
                    undefined): Width | null {
// Если width — null или undefined, вернуть заранее.
if (width == null) { ❶
    return null
}

// Если width — number, предустановить пикселы.
if (typeof width === 'number') { ❷
    return {unit: 'px', value: width}
}

// Попытка получить единицы измерения из width.
let unit = parseUnit(width)
if (unit) { ❸
    return {unit, value: parseFloat(width)}
}

// В противном случае вернуть null.
return null ❹
}
```

❶ TypeScript способен понять, что свободная проверка равенства на соответствие `null` в JavaScript вернет `true` и для `null`, и для `undefined`. Он также знает, что если проверка пройдет, то мы сделаем возврат, а если мы не делаем возврат, значит, проверка не прошла и с этого момента тип `width` — это `number | string` (он больше не может быть `null` или `undefined`). Мы говорим, что тип был уточнен из `number | string | null | undefined` в `number | string`.

❷ Проверка `typeof` запрашивает значение при выполнении, чтобы увидеть его тип. TypeScript также пользуется преимуществом `typeof` во время компиляции: в ветви `if`, где проверка проходит, TypeScript знает, что `width` — это `number`. В противном случае (если эта ветка делает `return`) `width` должна быть `string` — единственным оставшимся типом.

❸ Поскольку `parseUnit` может вернуть `null`, мы проверяем это[1]. TypeScript знает, что, если `unit` верна, тогда она должна иметь тип `Unit` в ветке `if`. В противном случае `unit` неверна, что значит — ее тип `null` (уточненный из `Unit | null`).

❹ В завершение мы возвращаем `null`. Это может случиться, только если пользователь передаст `string` для `width`, но эта строка будет содержать неподдерживаемые единицы измерения.

Я проговорил ход мыслей TypeScript в отношении каждого произведенного уточнения типа. TypeScript проделывает огромную работу, учитывая ваши рассуждения в процессе чтения и написания кода и кристаллизуя их в проверку типов и порядок их вывода.

#### Типы размеченного объединения

Как мы только что выяснили, TypeScript хорошо понимает принципы работы JavaScript и способен отслеживать наше уточнение типов, словно читая мысли.

Допустим, мы создаем систему пользовательских событий для приложения. Начинаем с определения типов событий наряду с функциями, обрабатывающими поступление этих событий. Представьте, что `UserTextEvent` моделирует событие клавиатуры (например, пользователь напечатал текст `<input />`), а `UserMouseEvent` моделирует событие мыши (пользователь сдвинул мышь в координаты `[100, 200]`):

---

[1] В JavaScript есть семь обозначений понятия «неверно»: `null`, `undefined`, `NaN`, `0`, `-0`, `''` и `false`. Все остальные относятся к понятию «верно».

```
type UserTextEvent = {value: string}
type UserMouseEvent = {value: [number, number]}

type UserEvent = UserTextEvent | UserMouseEvent

function handle(event: UserEvent) {
    if (typeof event.value === 'string') {
        event.value    // string
        // ...
        return
    }
    event.value        // [number, number]
}
```

TypeScript знает, что внутри блока `if event.value` должен быть `string` (благодаря проверке `typeof`), то есть `event.value` после блока `if` должно быть кортежем `[number, number]` (из-за `return` в блоке `if`).

К чему приведет усложнение? Добавим уточнения к типам событий:

```
type UserTextEvent = {value: string, target: HTMLInputElement}
type UserMouseEvent = {value: [number, number], target: HTMLElement}

type UserEvent = UserTextEvent | UserMouseEvent

function handle(event: UserEvent) {
    if (typeof event.value === 'string') {
        event.value    // string
        event.target   // HTMLInputElement | HTMLElement (!!!)
        // ...
        return
    }
    event.value        // [number, number]
    event.target       // HTMLInputElement | HTMLElement (!!!)
}
```

Хотя уточнение и сработало для `event.value`, этого не случилось для `event.target`. Почему? Когда `handle` получает параметр типа `UserEvent`, это не значит, что нужно передать ему либо `UserTextEvent`, либо `UserMouseEvent`, — на деле можно передать аргумент типа `UserMouseEvent | UserTextEvent`. И поскольку члены объединения могут перекрываться, TypeScript требуется более надежный способ узнать, когда и какой случай объединения актуален.

Сделать это можно с помощью типов литералов и определения тега для каждого случая типа объединения. Хороший тег:

- В каждом случае располагается на одном и том же месте типа объединения. Подразумевает то же поле объекта, если речь идет об объединении типов объектов, или тот же индекс, если дело касается объединения кортежей. На практике размеченные объединения чаще являются объектами.
- Типизирован как тип литерала (строчный литерал, численный, логический и т. д.). Можно смешивать и сопоставлять различные типы литералов, но лучше придерживаться единственного типа. Как правило, это тип строчного литерала.
- Не универсален. Теги не должны получать аргументы обобщенных типов.
- Взаимоисключающий (уникален внутри типа объединения).

Обновим типы событий с учетом вышесказанного:

```
type UserTextEvent = {type: 'TextEvent', value: string,
                      target: HTMLInputElement}
type UserMouseEvent = {type: 'MouseEvent', value: [number, number],
                       target: HTMLElement}

type UserEvent = UserTextEvent | UserMouseEvent

function handle(event: UserEvent) {
    if (event.type === 'TextEvent') {
        event.value    // string
        event.target   // HTMLInputElement
        // ...
        return
    }
    event.value        // [number, number]
    event.target       // HTMLElement
}
```

Теперь, когда мы уточняем `event` на основе значения его размеченного поля (`event.type`), TypeScript знает, что в ветке `if event` должен быть `UserTextEvent`, а после ветки `if` он должен быть `UserMouseEvent`, Поскольку теги уникальны в каждом типе объединения, TypeScript знает, что они являются взаимоисключающими.

Используйте размеченные объединения при написании функции, обрабатывающей различные случаи типа объединения. К примеру, при работе с действиями `Flux`, восстановлениями в `Redux` или с `useReducer` в `React`.

## Тотальность

> Программист перед сном ставит на тумбочку два стакана: полный — на случай, если захочет пить, и пустой — если не захочет.
>
> *Аноним*

Тотальность, или проверка полноты охвата, позволяет модулю проверки убедиться, что вы проработали все случаи. Она была позаимствована из языков, основанных на сопоставлении паттернов, таких как Haskell, OCaml и др.

TypeScript производит проверку тотальности в различных случаях, делая полезные предупреждения, если вы что-либо упустили. Эта функция отлично помогает предотвращать реальные баги. Например:

```
type Weekday = 'Mon' | 'Tue'| 'Wed' | 'Thu' | 'Fri'
type Day = Weekday | 'Sat' | 'Sun'

function getNextDay(w: Weekday): Day {
    switch (w) {
        case 'Mon': return 'Tue'
    }
}
```

Очевидно, что мы упустили пару дней (неделя была утомительной), но TypeScript приходит на помощь:

```
Ошибка TS2366: в функции отсутствует конечный оператор возврата,
а возвращаемый тип не включает 'undefined'.
```

Это сообщение указывает, что нужно либо дополнить упущенные случаи в конце инструкцией `return`, возвращающей что-то вроде `'Sat'`, либо изменить возвращаемый тип `getNextDay` в `Day | Undefined`. После добавления `case` для каждого `Day` ошибка исчезнет (попробуйте). Поскольку вы аннотировали возвращаемый тип `getNextDay`, но нет гарантии, что все ветви вернут значение этого типа, TypeScript вас предупреждает.

### TSC-ФЛАГ: NOIMPLICITRETURNS

Чтобы TypeScript проверял, все ли ветви функций возвращают значения (и выдавал предупреждение, когда вы что-то упустили), активируйте флаг `noImplicitReturns` в tsconfig.json. Вам решать, стоит ли это делать: некоторые разработчики используют меньше явных инструкций `return`, а другие большим числом таких инструкций повышают типобезопасность и улучшают перехват ошибок.

Детали реализации в этом примере непринципиальны: неважно, какую именно структуру управления вы используете — `switch`, `if`, `throw` или др., — TypeScript проследит, чтобы вы проработали все случаи.

Вот еще один пример:

```
function isBig(n: number) {
    if (n >= 100) {
        return true
    }
}
```

Возможно, продолжительные сообщения клиента о сорванном дедлайне заставили вас нервничать и вы забыли обработать числа ниже `100` в важной для бесперебойной деятельности функции `isBig`. Не стоит переживать, ведь TypeScript вас прикроет:

```
Ошибка TS7030: не все ветви кода возвращают значение.
```

А может, в выходные вас осенило, что нужно переписать этот пример с `getNextDay`, сделав его более эффективным. Почему бы вместо `switch` не использовать в объекте постоянно действующий просмотр?

```
let nextDay = {
    Mon: 'Tue'
}

nextDay.Mon // 'Tue'
```

Возможно, вас отвлекло тявканье соседской собаки, раз вы упустили заполнение остальных дней в новом объекте `nextDay`, машинально сделав коммит и приступив к другим делам.

Хоть TypeScript и сообщит вам об ошибке при следующей попытке обратиться к `nextDay.Tue`, вы могли бы быть изначально более предусмотрительны при объявлении `nextDay`. Позднее, в подразделе «Тип Record» на с. 172, вы узнаете, что для этого есть два способа. Но прежде, чем мы там окажемся, взгляните на операторы для типов объектов.

# Продвинутые типы объектов

Объекты — это центральная часть JavaScript, и TypeScript поддерживает множество способов безопасного управления ими.

## Операторы типов объектов

Помните операторы объединения ( `|` ) и пересечения (`&`) из подраздела «Типы объединения и пересечения» на с. 50? Рассмотрим несколько других операторов, подходящих для работы с формами.

### Оператор подключения (key in)

Представьте комплексный вложенный тип для моделирования ответа API GraphQL, получаемого из API выбранной социальной сети:

```
type APIResponse = {
    user: {
        userId: string
        friendList: {
            count: number
            friends: {
                firstName: string
                lastName: string
            }[]
        }
    }
}
```

Можно получить ответ от API, а затем отобразить его:

```
function getAPIResponse(): Promise<APIResponse> {
    // ...
}
```

```
function renderFriendList(friendList: unknown) {
    // ...
}

let response = await getAPIResponse()
renderFriendList(response.user.friendList)
```

Каким должен быть тип `friendList`? (Сейчас он зафиксирован как `unknown`.) Можно прописать его и повторно реализовать тип верхнего уровня `APIResponse` с учетом изменений:

```
type FriendList = {
    count: number
    friends: {
        firstName: string
        lastName: string
    }[]
}

type APIResponse = {
    user: {
        userId: string
        friendList: FriendList
    }
}

function renderFriendList(friendList: FriendList) {
    // ...
}
```

Но тогда придется придумать имена для каждого из типов верхнего уровня, что не всегда желательно (например, если вы использовали систему сборки для генерации типов TypeScript из схемы GraphQL). Вместо этого можно подключиться к типу:

```
type APIResponse = {
    user: {
        userId: string
        friendList: {
            count: number
            friends: {
                firstName: string
```

```
                lastName: string
            }[]
        }
    }
}
type FriendList = APIResponse['user']['friendList']

function renderFriendList(friendList: FriendList) {
    // ...
}
```

Это можно делать для любой формы (объекта, конструктора класса или его экземпляра), а также для любого массива. Например, чтобы получить тип отдельного друга:

```
type Friend = FriendList['friends'][number]
```

`number` — это способ подключения к типу массива. В кортежах используйте `0`, `1` или другой тип численного литерала для указания индекса, к которому нужно сделать подключение.

Синтаксис для этого намеренно аналогичен такому, какой используется для просмотра полей в регулярных объектах JavaScript, — так же как значение содержится в объекте, тип расположен в форме. Заметьте, что для поиска типов свойств при подключении нужно прописывать скобки, а не точки.

### Оператор keyof

Используйте `keyof` для получения всех ключей объекта в виде объединения типов строчных литералов. Возьмем предыдущий пример с APIResponse:

```
type ResponseKeys = keyof APIResponse    // 'user'
type UserKeys = keyof APIResponse['user']  // 'userId' | 'friendList'
type FriendListKeys =
    keyof APIResponse['user']['friendList']    // 'count' | 'friends'
```

Объединяя операторы подключения и `keyof`, вы можете реализовать типобезопасную функцию получения, которая ищет значение в заданном ключе объекта:

```
function get< ❶
    O extends object,
    K extends keyof O ❷
```

```
>(
    o: O,
    k: K
): O[K] { ❸
    return o[k]
}
```

❶ `get` — это функция, получающая объект `o` и ключ `k`.

❷ `keyof O` — это объединение типов строчных литералов, представляющее все ключи `o`. Обобщенный тип `K` является подтипом этого объединения и одновременно расширяет его. Например, если `o` имеет тип `{a: number, b: string, c: boolean}`, тогда `keyof o` — это тип `'a' | 'b' |'c'`, а `K` (расширяющий `keyof o`) может быть типом `'a', 'b', 'a' | 'c'` или любым другим подтипом `keof o`.

❸ `O[K]` — это тип, который вы получаете, когда ищете `K` в `0`. Продолжая пример из ❷, если `K` — это `'a'`. тогда при компиляции мы знаем, что `get` возвращает `number`. Если же `K` — это `'b' | 'c'`, тогда мы знаем, что `get` возвращает `string | boolean`.

Что хорошо в этих операторах типов, так это то, насколько точно и безопасно они позволяют описать формы типов:

```
type ActivityLog = {
    lastEvent: Date
    events: {
        id: string
        timestamp: Date
        type: 'Read' | 'Write'
    }[]
}

let activityLog: ActivityLog = // ...
let lastEvent = get(activityLog, 'lastEvent') // Дата (Date)
```

TypeScript сам проверяет при компиляции, что тип `lastEvent` является `Date`. Конечно, вы могли бы использовать расширение, чтобы произвести подключение также или более глубоко в объекте. Перегрузим `get`, чтобы она принимала до трех ключей:

```
type Get = { ❶
    <
        O extends object,
```

```
        K1 extends keyof O
    >(o: O, k1: K1): O[K1] ❷
    <
        O extends object,
        K1 extends keyof O,
        K2 extends keyof O[K1] ❸
    >(o: O, k1: K1, k2: K2): O[K1][K2] ❹
    <
        O extends object,
        K1 extends keyof O,
        K2 extends keyof O[K1],
        K3 extends keyof O[K1][K2]
    >(o: O, k1: K1, k2: K2, k3: K3): O[K1][K2][K3] ❺
}

let get: Get = (object: any, ...keys: string[]) => {
    let result = object
    keys.forEach(k => result = result[k])
    return result
}

get(activityLog, 'events', 0, 'type') // 'Чтение' | 'Запись'

get(activityLog, 'bad')    // Ошибка TS2345: аргумент типа '"bad"'
                           // несовместим с параметром типа
                           // '"lastEvent" | "events"'.
```

❶ Объявляем перегруженную сигнатуру функции для `get` с тремя случаями, где мы вызываем `get` с одним, двумя и тремя ключами.

❷ Случай с одним ключом совпадает с последним примером: `O` — это подтип `object`, `K1` — это подтип ключей этого объекта, а возвращаемый тип — это любой конкретный тип, который мы получаем при подключении к `O` через `K1`.

❸ Случай с двумя ключами похож на случай с одним, но мы объявляем еще один обобщенный тип, `K2`, для моделирования возможных ключей во вложенном объекте, получающемся после подключения к `O` через `K1`.

❹ Продолжаем построение ❷, дважды произведя подключение. Сначала мы получили тип `O[K]`, а затем от этого результата получили тип `[K2]`.

❺ В этом примере мы обрабатывали до трех вложенных ключей. Если вы пишете реальную библиотеку, то, вероятно, решите обработать большее число случаев.

Круто? Найдите время показать этот пример своим Java-друзьям и позлорадствуйте над ними.

**TSC-ФЛАГ: KEYOFSTRINGSONLY**

В JavaScript объекты и массивы могут иметь как строчные, так и символьные (`symbol`) ключи. Обычно для массивов применяются численные ключи, которые при выполнении переводятся в строки.

В связи с этим `keyof` в TypeScript по умолчанию возвращает значение типа `number | string | symbol` (если вы вызовете его для более конкретной формы, то TypeScript может вывести более конкретный подтип).

Это правильное поведение, но оно может сделать многословным работу с `typeof` в случаях, когда требуется доказать TypeScript, что конкретный ключ является `string`, а не `number` или `symbol`.

Чтобы переключиться на старую модель поведения TypeScript, когда ключи должны быть строками, активируйте флаг `keyofStringsOnly` в tsconfig.json.

## Тип Record

Встроенный в TypeScript тип `Record` — это способ описать объект как отображение на другой объект.

Вспомните из примера с `Weekday` в разделе «Тотальность» на с. 165, что есть два способа сделать так, чтобы объект определял конкретный набор ключей. Типы `Record` — это первый из них.

Давайте используем `Record` для построения отображения каждого дня недели на следующий день недели. С помощью `Record` можно накладывать некоторые ограничения на ключи и значения в `nextDay`:

```
type Weekday = 'Mon' | 'Tue'| 'Wed' | 'Thu' | 'Fri'
type Day = Weekday | 'Sat' | 'Sun'
```

```
let nextDay: Record<Weekday, Day> = {
    Mon: 'Tue'
}
```

И тут вы получите полезное сообщение об ошибке:

```
Ошибка TS2739: в типе '{Mon: "Tue"}' упущены следующие свойства
из типа 'Record<Weekday, Day>': Tue, Wed, Thu, Fri.
```

Добавление упущенных `Weekdays` в объект, конечно же, устранит проблему.

`Record` дает вам бóльшую свободу, по сравнению с обычными сигнатурами индексов объектов, позволяющими ограничивать типы значений объектов, но допускает только стандартные ключи: `string`, `number` или `symbol`. С помощью `Record` можно дополнительно ограничивать типы ключей объекта до подтипов `string` и `number`.

## Отображенные типы

TypeScript дает второй отличный способ объявить более безопасный тип `nextDay` — отображенные типы. Используем их для указания, что `nextDay` — это объект с ключом для каждого `Weekday`, чье значение — это `Day`:

```
let nextDay: {[K in Weekday]: Day} = {
    Mon: 'Tue'
}
```

Это еще один подход для исправления упущенного:

```
Ошибка TS2739: в типе '{Mon: "Tue"}' упущены следующие свойства типа
type '{Mon: Weekday; Tue: Weekday; Wed: Weekday; Thu: Weekday; Fri:
Weekday}': Tue, Wed, Thu, Fri.
```

Отображенные типы — это уникальная особенность TypeScript. Подобно типам литералов, они являются полезной функцией для статической типизации JavaScript-кода.

Как вы увидели, отображенные типы имеют особый синтаксис. И в объекте должно быть не более одного отображенного типа (как и сигнатур индекса):

```
type MyMappedType = {
    [Key in UnionType]: ValueType
}
```

По определению они помогают отображать ключи объекта и типы значений. На деле TypeScript использует отображенные типы для реализации встроенного типа `Record`:

```
type Record<K extends keyof any, T> = {
    [P in K]: T
}
```

Они дают больше возможностей, чем простой тип `Record`, позволяя не только присваивать типы ключам и значения объектам, но и устанавливать при комбинировании с подключенными типами ограничения, определяющие, какие типы значений соответствуют тем или иным именам ключей.

Вкратце рассмотрим некоторые возможности, доступные при их применении:

```
type Account = {
    id: number
    isEmployee: boolean
    notes: string[]
}

// Сделать все поля опциональными
type OptionalAccount = {
    [K in keyof Account]?: Account[K] ❶
}

// Сделать все поля допускающими null
type NullableAccount = {
    [K in keyof Account]: Account[K] | null ❷
}

// Сделать все поля read-only
type ReadonlyAccount = {
    readonly [K in keyof Account]: Account[K] ❸
}

// Снова сделать все поля записываемыми (эквивалент Account)
type Account2 = {
    -readonly [K in keyof ReadonlyAccount]: Account[K] ❹
}
```

```
// Снова сделать все поля обязательными (эквивалент Account)
type Account3 = {
    [K in keyof OptionalAccount]-?: Account[K] ❺
}
```

❶ Создаем новый тип объекта `OptionalAccount`, отображаем `Account` и попутно помечаем каждое поле как опциональное.

❷ Создаем новый тип объекта `NullableAccount`, отображаем `Account` и попутно добавляем `null` в качестве возможного значения для каждого поля.

❸ Создаем новый тип объекта `ReadonlyAccount` и делаем каждое поле `Account` доступным только для чтения.

❹ Можно помечать поля как опциональные или `readonly` и убирать подобные пометки. С помощью оператора типа минус (`-`), доступного только для отображенных типов, отменяем `?` и `readonly`, что сделает поля снова обязательными и записываемыми соответственно. Далее создаем новый тип объекта `Account2` — эквивалент `Account`, отображаем `ReadonlyAccount` и удаляем модификатор `readonly` с помощью оператора минус (`-`).

❺ Создаем новый тип объекта `Account3` — тоже эквивалент `Account`, отображаем `OptionalAccount` и удаляем оператор опциональности (`?`) с помощью оператора минус (`-`).

Минус (`-`) имеет встречный оператор плюс (`+`), который вам, вероятно, никогда не придется использовать непосредственно, поскольку внутри отображенного типа `readonly` является эквивалентом `+readonly`, а `?` является эквивалентом `+?`. То есть `+` присутствует просто для полноты.

#### Встроенные отображенные типы

Встроенные типы, с которыми мы познакомились в последнем разделе, настолько полезны, что TypeScript содержит множество их вариантов:

`Record<Keys, Values>`

Объект с ключами типа `Keys` и значениями типа `Values`.

`Partial<Object>`

Помечает каждое поле в `Object` как опциональное.

`Required<Object>`

Помечает каждое поле в `Object` как обязательное.

`Readonly<Object>`

Помечает каждое поле в `Object` только для чтения.

`Pick<Object, Keys>`

Возвращает подтип `Object` только с заданными `Keys`.

## Паттерн объект-компаньон

Этот паттерн позаимствован из Scala (https://docs.scala-lang.org/tour/singleton-objects.html#companion-objects), где он дает возможность объединять объекты и классы, имеющие одинаковое имя. В TypeScript аналогичный паттерн служит для объединения типа и объекта.

Вот как он выглядит:

```
type Currency = {
    unit: 'EUR' | 'GBP' | 'JPY' | 'USD'
    value: number
}

let Currency = {
    DEFAULT: 'USD',
    from(value: number, unit = Currency.DEFAULT): Currency {
        return {unit, value}
    }
}
```

Вспомните, что в TypeScript типы и значения имеют раздельные пространства имен (см. раздел «Слияние деклараций» на с. 279). Это значит, что в одной области могут присутствовать тип и значение с одним именем (в данном примере `Currency`). В паттерне объект-компаньон это разделение позволяет объявить имя дважды: сначала для типа, а затем для значения.

У этого паттерна есть ряд положительных свойств. Он позволяет группировать информацию типа и значения, которая семантически является частью одного имени. Он также позволяет потребителям импортировать и то и другое за раз:

```
import {Currency} from './Currency'

let amountDue: Currency = { ❶
    unit: 'JPY',
    value: 83733.10
}

let otherAmountDue = Currency.from(330, 'EUR') ❷
```

❶ Использование `Currency` как типа.

❷ Использование `Currency` как значения.

Используйте этот паттерн, когда тип и объект семантически связаны, а объект предоставляет служебные методы, которые оперируют с типом.

# Продвинутые функциональные типы

Рассмотрим некоторые более продвинутые техники, часто используемые с функциональными типами.

## Улучшение вывода типов для кортежей

Когда вы объявляете кортеж, TypeScript выводит его как можно более общим исходя из предоставленных ему данных независимо от длины кортежа и позиционирования типов:

```
let a = [1, true] // (number | boolean)[]
```

Но иногда нужен более строгий вывод, который расценивал бы `a` как кортеж фиксированной длины, а не как массив. Конечно, можно использовать утверждение типа для приведения кортежа в тип кортежа (см. подраздел «Утверждения типов» на с. 185) либо утверждение `as const` (см. подраздел «Тип const» на с. 156) для максимально узкого вывода типа кортежа, пометив его доступным только для чтения.

Но что, если вы хотите типизировать кортеж как кортеж, но избежать утверждения, узкого вывода и доступности только для чтения? Воспользуйтесь преимуществом вывода типов для оставшихся параметров (см. подраздел «Использование ограниченного полиморфизма для моделирования арности» на с. 105):

```
function tuple< ❶
    T extends unknown[] ❷
>(
    ...ts: T ❸
): T { ❹
    return ts ❺
}

let a = tuple(1, true) // [number, boolean]
```

❶ Объявляем функцию `tuple`, которую также используем для построения типа кортежа (вместо встроенного синтаксиса `[]`).

❷ Объявляем один параметр типа `T`, являющийся подтипом `unknown[]` (это значит, что `T` является массивом любого типа).

❸ `tuple` получает переменное число параметров `ts`. Поскольку `T` описывает оставшийся параметр, TypeScript выводит для него тип кортежа.

❹ `tuple` возвращает значение того же типа кортежа, который был выведен для `ts`.

❺ Функция возвращает тот же аргумент, который был ей передан. Вся магия в типах.

Воспользуйтесь этой техникой, когда в коде присутствует много кортежных типов.

### Пользовательские защиты типов

Для некоторых видов функций, возвращающих `boolean`, может быть недостаточно просто указать: «Эта функция возвращает `boolean`». Напишем функцию, сообщающую, передали вы `string` или нет:

```
function isString(a: unknown): boolean {
    return typeof a === 'string'
}

isString('a')    // вычисляется как true
isString([7])    // вычисляется как false
```

Пока все хорошо. Что произойдет, если использовать `isString` в реальном коде?

```
function parseInput(input: string | number) {
    let formattedInput: string
    if (isString(input)) {
      formattedInput = input.toUpperCase() // Ошибка TS2339: свойство
                                           // 'toUpperCase' не существует
    }                                      // в типе 'number'.
}
```

`typeof` работает для обычного уточнения типа (см. подраздел «Уточнение» на с. 160) но не работает здесь?

Особенность уточнения типов в том, что оно работает только для типа переменной, находящейся в текущей области. В реализации `isString` мы уточнили тип вводного параметра до `string`, использовав `typeof`, но уточнение не перешло в другие области и утратилось, поэтому TypeScript узнал только, что `isString` вернула `boolean`.

В этом случае мы можем сообщить модулю проверки не только что `isString` возвращает `boolean`, но также что если `boolean` будет `true`, то переданный нами в `isString` аргумент является `string`. Чтобы это сделать, мы используем так называемые пользовательские защиты типов:

```
function isString(a: unknown): a is string {
    return typeof a === 'string'
}
```

Защиты типов — это встроенная особенность TypeScript, позволяющая уточнять типы с `typeof` и `instanceof`. Иногда объявить защиты типов самостоятельно позволяет оператор `is`. Когда есть функция, уточняющая типы своих параметров и возвращающая `boolean`, можно использовать пользовательскую защиту типа, гарантирующую, что уточнение будет передано при каждом использовании этой функции.

Пользовательские защиты типов ограничены одним параметром, но не ограничены простыми типами:

```
type LegacyDialog = // ...
type Dialog = // ...

function isLegacyDialog(
    dialog: LegacyDialog | Dialog
):  dialog is LegacyDialog {
    // ...
}
```

Они используются нечасто, но отлично помогают в написании чистого переиспользуемого кода. Без них вместо построения функций вроде `isLegacyDialog` и `isString` пришлось бы встраивать все защиты типов `typeof` и `instanceof`, чтобы выполнить те же проверки более инкапсулированным и читабельным способом.

## Условные типы

Условные типы — уникальная особенность TypeScript. На высоком уровне они позволяют сказать: «Объявляю тип `T`, зависящий от типа `U` и `V`. Если `U <: V`, то присваиваю `T` типу `A`, в противном случае — присваиваю `T` типу `B`».

В коде это может выглядеть так:

```
type IsString<T> = T extends string ❶
    ? true ❷
    : false ❸

type A = IsString<string>    // true
type B = IsString<number>    // false
```

Рассмотрим каждую строку.

❶ Объявляем новый условный тип `isString`, получающий обобщенный тип `T`. «Условная» часть этого типа — `T extends string` — означает «Является ли `T` подтипом `string`?»

❷ Если `T` — это подтип `string`, переходим к типу `true`.

❸ В противном случае переходим к типу `false`.

Обратите внимание, что синтаксис напоминает обычное троичное выражение уровня значений, но на уровне типов. И подобно обычному троичному выражению, он может быть вложенным.

Условные типы не ограничиваются псевдонимами типов. Их можно применить почти везде, где используется тип: в псевдонимах типов, классах, типах параметров и обобщенных предустановках функций и методов.

### Условное распределение

Хоть вы и можете выражать простые условия, подобные рассмотренным в примерах, различными способами — с помощью условных типов, пере-

груженных сигнатур функций и отображенных типов, — условные типы дают больше всего возможностей. Причина в том, что они подчиняются закону распределения (помните уроки алгебры?). Это означает, что выражение в правой части условного типа эквивалентно выражению в его левой части (табл. 6.1).

Знаю-знаю, вы купили эту книгу не для изучения математики, ваша цель — освоение типов. Значит, будем более конкретными. Возьмем функцию, получающую некую переменную типа `T` и повышающую ее до массива типа `T[]`. Что случится, если мы передадим для `T` тип объединения?

```
type ToArray<T> = T[]
type A = ToArray<number>              // number[]
type B = ToArray<number | string>     // (number | string)[]
```

**Таблица 6.1.** Распределение условных типов

| Это | эквивалентно этому |
|---|---|
| `string extends T ? A : B` | `string extends T ? A : B` |
| `(string \| number) extends T ? A : B` | `(string extends T ? A : B) \| (number extends T ? A : B)` |
| `(string \| number \| boolean) extends T ? A : B` | `(string extends T ? A : B) \| (number extends T ? A : B) \| (boolean extends T ? A : B)` |

Достаточно просто. А что произойдет, если мы добавим условный тип? (Заметьте, что условие здесь ни к чему не приводит, поскольку обе ветви приходят к одному типу `T[]`. Оно просто сообщает TypeScript, что нужно *распределить* `T` по типу кортежа.) Взгляните:

```
type ToArray2<T> = T extends unknown ? T[] : T[]
type A = ToArray2<number>             // number[]
type B = ToArray2<number | string>    // number[] | string[]
```

Уловили? Когда вы используете условный тип, TypeScript распределяет типы объединения по ветвям условного выражения. Как будто условный тип отображен (то есть *распределен*) на каждый элемент объединения.

Почему все это важно? Для безопасного выражения привычных операций.

Например, TypeScript имеет оператор `&` для вычисления общих черт двух типов и оператор `|` для объединения двух типов. Создадим выражение `Without<T, U>`, вычисляющее, какие типы есть в `T`, но отсутствуют в `U`.

```
type Without<T, U> = T extends U ? never : T
```

`Without` используется так:

```
type A = Without<
    boolean | number | string,
    boolean
> // number | string
```

Рассмотрим, как TypeScript вычисляет этот тип.

1. Начнем с вводных данных:

   ```
   type A = Without<boolean | number | string, boolean>
   ```

2. Распределим условие по объединению:

   ```
   type A = Without<boolean, boolean>
          | Without<number, boolean>
          | Without<string, boolean>
   ```

3. Сделаем подстановку в определение `Without` и применим `T` и `U`:

   ```
   type A = (boolean extends boolean ? never : boolean)
          | (number extends boolean ? never : number)
          | (string extends boolean ? never : string)
   ```

4. Вычислим условия:

   ```
   type A = never
          | number
          | string
   ```

5. Упростим:

   ```
   type A = number | string
   ```

Если бы не свойство распределения условных типов, мы бы получили на выходе `never` (если вы не поняли почему, попробуйте самостоятельно рассмотреть такой вариант).

### Ключевое слово infer

Заключительной особенностью условных типов является возможность объявлять обобщенные типы как часть условия. Напомню, что до сих пор мы встречали лишь один способ объявления параметров обобщенного типа: с помощью угловых скобок (`<T>`). Условные типы имеют свой собственный синтаксис для объявления встроенных обобщенных типов: ключевое слово `infer`.

Объявим условный тип `ElemetType`, получающий тип элементов массива:

```
type ElementType<T> = T extends unknown[] ? T[number] : T
type A = ElementType<number[]>   // number
```

А теперь перепишем его, используя `infer`:

```
type ElementType2<T> = T extends (infer U)[] ? U : T
type B = ElementType2<number[]>   // number
```

В этом простом примере `ElementType` является эквивалентом `ElementType2`. Заметьте, как с `infer` объявляется переменная нового типа `U` — TypeScript выводит тип `U` из контекста на основе `T`, переданного в `ElementType2`.

Почему мы объявили `U` встроенным вместо объявления его впереди вместе с `T`? Что бы произошло, если бы мы это сделали?

```
type ElementUgly<T, U> = T extends U[] ? U : T
type C = ElementUgly<number[]>  // Ошибка TS2314: обобщенный тип
                            // 'ElementUgly' требует 2 аргумента типа.
```

Поскольку `ElementUgly` определяет два обобщенных типа (`T` и `U`), при инстанцировании `ElementUgly` мы должны передать их оба. Но так мы утратим смысл типа `ElemenUgly`. Вычисление `U` будет возложено на вызывающего, а мы хотели, чтобы `ElementUgly` вычислял тип самостоятельно.

Честно говоря, это был слишком простой пример, поскольку для поиска типов элементов массива мы уже имеем оператор подключения (`[]`). Как насчет примера посложнее?

```
type SecondArg<F> = F extends (a: any, b: infer B) => any ? B : never

// Получаем тип Array.slice
type F = typeof Array['prototype']['slice']

type A = SecondArg<F> // number | undefined
```

Итак, второй аргумент `[].slice` является `number | undefined`, и мы узнаем об этом в процессе компиляции — попробуйте-ка сделать *такое* в *Java*.

## Встроенные условные типы

Условные типы позволяют выражать действительно мощные операции на уровне типов. Именно поэтому в TypeScript по умолчанию имеется несколько их вариантов.

`Exclude<T, U>`

Подобно уже знакомому типу `Without`, он вычисляет, какие типы есть в `T`, но отсутствуют в `U`:

```
type A = number | string
type B = string
type C = Exclude<A, B> // number
```

`Extract<T, U>`

Вычисляет типы в `T`, которые можно присвоить `U`:

```
type A = number | string
type B = string
type C = Extract<A, B> // string
```

`NonNullable<T>`

Вычисляет версию `T`, исключающую `null` и `undefined`:

```
type A = {a?: number | null}
type B = NonNullable<A['a']> // number
```

`ReturnType<F>`

Вычисляет возвращаемый тип функции (вопреки ожиданиям, это не сработает для обобщенных типов и перегруженных функций):

```
type F = (a: number) => string
type R = ReturnType<F> // string
```

`InstanceType<C>`

Вычисляет тип экземпляра конструктора класса:

```
type A = {new(): B}
type B = {b: number}
type I = InstanceType<A> // {b: number}
```

# Запасные решения

Иногда нет времени на идеальную типизацию и просто хочется, чтобы TypeScript поверил безопасность. Возможно, неверна декларация для стороннего модуля и вы хотите протестировать код до отправки исправлений обратно на DefinitelyTyped[1]. А может, вы получаете данные от API, но еще не восстановили декларации типов с помощью `Apollo`.

К счастью, TypeScript понимает, что мы всего лишь люди, и дает нам несколько запасных решений на случаи, когда мы хотим сделать что-то быстро.

Рекомендуется использовать приведенные решения как можно реже и только в случаях реальной необходимости. Если вы начнете на них полагаться, то вполне можете допустить ошибку.

## Утверждения типов

Если у вас есть тип `B`, а `A <: B <: C`, тогда вы можете сделать утверждение для модуля проверки, что `B` фактически является `A` или `C`. Стоит отметить, что утверждение можно использовать, только если тип является супертипом или подтипом самого себя. К примеру, вы не можете утверждать, что `number` — это `string`, потому что эти типы не родственны.

В TypeScript есть два вида синтаксиса для утверждения типов:

```
function formatInput(input: string) {
    // ...
}
```

[1] DefinitelyTyped — это открытый репозиторий деклараций типов для стороннего JavaScript-кода (https://github.com/DefinitelyTyped/DefinitelyTyped, см. подраздел «JavaScript, имеющий декларации типов на DefinitelyTyped» на с. 302).

```
function getUserInput(): string | number {
    // ...
}

let input = getUserInput()

// Утверждение, что input это string
formatInput(input as string) ❶

// Его эквивалент
formatInput(<string>input) ❷
```

❶ Утверждаем (`as`), что `input` является `string`, а не `string | number`, как якобы предполагают типы. Это поможет быстро протестировать функцию `formatInput`, но вы должны быть уверены, что `getUserInput` возвращает для вашего теста `string`.

❷ Устоявшийся синтаксис для утверждений типов использует угловые скобки. При этом оба синтаксиса равнозначны.

Если два типа не родственны, утвердите `any` (вспомните из подраздела «Совместимость» на с. 154, что `any` совместим с чем угодно), а после встаньте в угол и поразмышляйте, что вы наделали:

```
function addToList(list: string[], item: string) {
    // ...
}
addToList('this is really,' as any, 'really unsafe')
```

Очевидно, что утверждения типов небезопасны и их следует всячески избегать.

Старайтесь использовать синтаксис `as` вместо угловых скобок (`<>`), потому что он является однозначным, в то время как второй вариант может противоречить синтаксису *TSX* (см. подраздел «*TSX = JSX + + TypeScript*» на с. 251). Используйте правило TSLint `no-angle-bracket-type-assertion` для автоматического контроля этого нюанса в базе кода (https://palantir.github.io/tslint/rules/no-angle-bracket-type-assertion/).

### Ненулевые утверждения

Для типов, допускающих нулевые значения (`T | null` или `T | null | undefined`), в TypeScript есть специальный синтаксис утверждения, что значение типа — это `T`, а не `null` или `undefined`. Встречается он в нескольких местах.

Например, мы написали фреймворк для показа и скрытия диалогов в веб-приложении. Каждый диалог получает уникальный `ID` для возвращения ссылки на DOM-узел диалога. Как только диалог удаляется из DOM, мы стираем его `ID`, указывая, что в DOM он больше не существует:

```
type Dialog = {
    id?: string
}

function closeDialog(dialog: Dialog) {
    if (!dialog.id) { ❶
        return
    }
    setTimeout(() => ❷
        removeFromDOM(
          dialog,
          document.getElementById(dialog.id) // Ошибка TS2345:
                                             // аргумент типа 'string |
                                             // undefined' несовместим
                                             // с параметром типа
                                             // 'string'. ❸
        )
    )
}

function removeFromDOM(dialog: Dialog, element: Element) {
    element.parentNode.removeChild(element)  // Ошибка TS2531:
                                             // объект, вероятно,
                                             // 'null'. ❹
    delete dialog.id
}
```

❶ Если диалог уже удален (у него нет `id`), сразу делаем возврат.

❷ Удаляем диалог из DOM по завершении цикла события, чтобы любой другой зависящий от `dialog` код мог закончить свое выполнение.

❸ Поскольку мы находимся внутри стрелочной функции, то теперь мы в новой области. TypeScript не знает, был ли изменен `dialog` между ❶ и ❸, поэтому он не принимает произведенное нами в ❶ уточнение. Вдобавок к этому мы знаем, что если `dialog.id` определен, то элемент с этим `ID` точно существует в DOM (так настроен фреймворк), но все, что знает TypeScript, — это то, что вызов `document.getElementById` возвращает `HTMLElement | null`. Мы знаем, что `HTMLElement` всегда будет допускать нулевое значение, но TypeScript это неизвестно — он лишь знает о типах, которые мы ему передали.

❹ Схожий случай. Хоть мы и знаем, что диалог точно есть в DOM и у него точно есть родительский узел DOM, все, что знает TypeScript, — это то, что тип `element.parentNode` является `Node | null`.

Если вы не уверены, является что-либо `null` или нет, исправить это можно добавлением проверок `if (_===null)` повсюду. Для случаев, когда вы точно знаете, что элемент не является `null | undefined`, в TypeScript есть специальный синтаксис:

```
type Dialog = {
    id?: string
}

function closeDialog(dialog: Dialog) {
    if (!dialog.id) {
        return
    }
    setTimeout(() =>
        removeFromDOM(
            dialog,
            document.getElementById(dialog.id!)!
        )
    )
}

function removeFromDOM(dialog: Dialog, element: Element) {
    element.parentNode!.removeChild(element)
    delete dialog.id
}
```

Обратите внимание на вкрапления операторов ненулевого утверждения (`!`), которые указывают TypeScript на нашу уверенность, что `dialog.id` — результат вызова `document.getElementById`, и на то, что `element.parentNode` определены. Когда не-`null`-утверждение сопровождает тип, который может быть `null` или `undefined`, TypeScript предполагает, что тип определен: `T | null | undefined` станет `T`, `number | string | null` станет `number | string` и т. д.

Если вы вдруг заметите, что часто используете не-`null`-утверждения, значит, пора делать рефакторинг кода. Например, избавиться от утверждения и сделать `Dialog` объединением двух типов:

```
type VisibleDialog = {id: string}
type DestroyedDialog = {}
type Dialog = VisibleDialog | DestroyedDialog
```

Затем обновить `closeDialog`, чтобы воспользоваться возможностями объединения:

```
function closeDialog(dialog: Dialog) {
    if (!('id' in dialog)) {
        return
    }
    setTimeout(() =>
        removeFromDOM(
            dialog,
            document.getElementById(dialog.id)!
        )
    )
}

function removeFromDOM(dialog: VisibleDialog, element: Element) {
    element.parentNode!.removeChild(element)
    delete dialog.id
}
```

После проверки, что свойство `id` в `dialog` определено (подразумевается, что это `VisibleDialog`), даже внутри стрелочной функции TypeScript будет знать, что ссылка на `dialog` не изменилась, то есть `dialog` внутри стрелочной функции — это тот же `dialog`, что и вне функции. Поэтому уточнение пройдет, вместо того чтобы быть отклоненным, как в предыдущем примере.

### Утверждения явного присваивания

В TypeScript есть специальный синтаксис для особых случаев не-`null`-утверждений при проверках на явное присваивание (эти проверки уточняют, что все переменные на момент использования имеют присвоенные им значения). Например:

```
let userId: string

userId.toUpperCase()          // Ошибка TS2454: переменная 'userId'
                              // используется до получения значения.
```

Очевидно, что TypeScript очень нам помог, перехватив эту ошибку. Мы объявили переменную `userId`, но забыли присвоить ей значение до попытки преобразовать ее в верхний регистр. Если бы TypeScript не заметил проблему, то это привело бы к ошибке среды выполнения.

Но что, если код выглядит, к примеру, так?

```
let userId: string
fetchUser()

userId.toUpperCase()          // Ошибка TS2454: переменная 'userId'
                              // используется до получения значения.

function fetchUser() {
    userId = globalCache.get('userId')
}
```

Здесь мы имеем величайший в мире кеш, и когда мы запрашиваем этот кеш, то получаем стопроцентное попадание в него. Следовательно, после вызова `fetchUser` гарантированно будет определен `userId`. Но TypeScript не способен статически это обнаружить, поэтому он по-прежнему выдает ту же ошибку. Мы можем использовать утверждение явного присваивания, чтобы указать TypeScript время присваивания `userId` (обратите внимание на знак восклицания):

```
let userId!: string
fetchUser()
userId.toUpperCase() // OK
function fetchUser() {
    userId = globalCache.get('userId')
}
```

Так же как в случае с утверждениями типов и не-null-утверждениями, если вы заметите, что часто используете утверждение явного присваивания, то, скорее всего, вы что-то делаете не так.

## Имитация номинальных типов

Если бы я растолкал вас в три часа ночи с криком «КАКАЯ СИСТЕМА ТИПОВ В TYPESCRIPT? СТРУКТУРНАЯ ИЛИ НОМИНАЛЬНАЯ?», то вы бы в ответ заорали: «КОНЕЧНО ЖЕ, СТРУКТУРНАЯ! А ТЕПЕРЬ ВАЛИ ИЗ МОЕГО ДОМА, ПОКА Я НЕ ВЫЗВАЛ ПОЛИЦИЮ!» И до этого момента это был бы верный ответ.

Долой законы. Реальность такова, что иногда номинальные типы действительно полезны. Допустим, в приложении есть несколько типов ID, представляющих уникальные способы обращения к разным типам объектов в системе:

```
type CompanyID = string
type OrderID = string
type UserID = string
type ID = CompanyID | OrderID | UserID
```

Значением типа UserID может быть простой хеш, выглядящий примерно так: 'd211b1dbf'. Поэтому, в то время как вы можете псевдонимизировать его как UserId, на деле окажется, что это обычный тип string. Функция, получающая UserID, может выглядеть так:

```
function queryForUser(id: UserID) {
    // ...
}
```

Это отличное документирование, и оно помогает другим инженерам вашей команды знать наверняка, какой тип ID они должны передать. Но поскольку UserID — это всего лишь псевдоним для string, этот подход мало помогает предотвратить баги. Инженер может по случайности передать неверный тип ID, а система типов не поймет, что происходит.

```
let id: CompanyID = 'b4843361'
queryForUser(id) // OK (!!!)
```

Как раз здесь и пригождаются номинальные типы[1]. Хоть TypeScript и не поддерживает их по умолчанию, мы можем симулировать их с помощью

[1] В некоторых языках они называются непрозрачными типами.

функции маркировки типов. Маркировка типов требует некоторых усилий для настройки, а ее использование в TypeScript не настолько удобно, насколько в языках, имеющих встроенную поддержку псевдонимов номинальных типов. И все-таки использование маркированных типов может существенно повысить безопасность программы.

Актуальность использования этой техники определяется приложением и размером вашей команды инженеров (чем больше команда, тем более эффективно эта техника поможет предотвратить ошибки). Но иногда она может не пригодиться вовсе.

Начните с создания синтетической маркировки для каждого номинального типа:

```
type CompanyID = string & {readonly brand: unique symbol}
type OrderID = string & {readonly brand: unique symbol}
type UserID = string & {readonly brand: unique symbol}
type ID = CompanyID | OrderID | UserID
```

Пересечение `string` и `{readonly brand: unique symbol}` — это, конечно, бессмыслица. Я его выбрал потому, что невозможно естественным образом сконструировать этот тип — только утвердить. Это особенность маркированных типов: они препятствуют появлению на их месте неверного типа. Я выбрал в качестве маркировки `unique symbol`, потому что это один из двух поистине номинальных видов типов в TypeScript (второй — это `enum`). Пересечение же этой маркировки со `string` я сделал, чтобы мы могли утвердить, что этот тип `string` является данным маркированным типом.

Теперь нам нужен способ создать значения типов `CompanyID`, `OrderID` и `UserID`. Для этого мы используем паттерн объект-компаньон (см. подраздел «Паттерн объект-компаньон» на с. 176). Мы создадим конструктор для каждого маркированного типа, используя утверждение типа, которое позволит получить значения:

```
function CompanyID(id: string) {
    return id as CompanyID
}

function OrderID(id: string) {
```

```
    return id as OrderID
}

function UserID(id: string) {
    return id as UserID
}
```

В завершение давайте взглянем, как использовать такие типы:

```
function queryForUser(id: UserID) {
    // ...
}

let companyId = CompanyID('8a6076cf')
let orderId = OrderID('9994acc1')
let userId = UserID('d21b1dbf')

queryForUser(userId)     // OK
queryForUser(companyId)  // Ошибка TS2345: аргумент типа 'CompanyID'
                         // несовместим с параметром типа 'UserID'.
```

В этом подходе хорошо то, как мало хлопот он вызывает в среде выполнения: всего один вызов функции для каждой конструкции `ID`, которая будет встроена вашей виртуальной машиной JavaScript в любом случае. При выполнении каждый `ID` — это просто `string`, а маркировка — это всего лишь конструкция среды компиляции.

И снова отмечу, что для большинства приложений такой подход неоправдан. Однако для крупных проектов, а также при работе с легко путаемыми типами наподобие различных видов `ID` маркированные типы могут быть очень полезны для безопасности.

## Безопасное расширение прототипа

Традиционно считается, что при построении JavaScript-приложений небезопасно расширять прототипы встроенных типов. Это общее правило восходит к периоду, предшествующему jQuery, когда мудрые JavaScript-маги создавали библиотеки вроде MooTools (https://mootools.net/), которые напрямую расширяли и переписывали встроенные методы прототипов. Но, когда слишком много магов одновременно увеличивали прототипы, возникали конфликты. А при отсутствии статической системы типов об

этих конфликтах можно было узнать только от озлобленных пользователей в среде выполнения.

Если вы ранее не использовали JavaScript, то вас поразит, что в нем можно изменять любой встроенный метод (вроде `[].push, 'abc'.toUpperCase` или `Object.assign`) при выполнении. Будучи динамическим языком, JavaScript дает вам прямой доступ к прототипам для каждого встроенного объекта — `Array.prototype`, `Function.prototype`, `Object,prototype` и т. д.

В прошлом расширение прототипов было опасным занятием, но сегодня, если ваш код использует статическую систему типов вроде TypeScript, вы можете делать это безопасно[1].

В качестве примера добавим метод `zip` в прототип `Array`. Потребуются два шага для безопасного расширения этого прототипа. Сначала в файле .ts (например, zip.ts) расширим тип прототипа `Array`. Затем добавим к прототипу новый метод `zip`:

```
// Сообщаем TypeScript о .zip
interface Array<T> { ❶
    zip<U>(list: U[]): [T, U][]
}

// Реализуем .zip
Array.prototype.zip = function<T, U>(
    this: T[], ❷
    list: U[]
):  [T, U][] {
    return this.map((v, k) =>
        tuple(v, list[k]) ❸
    )
}
```

❶ Вначале сообщаем TypeScript, что добавляем `zip` в `Array`. Используем возможности слияния интерфейсов (см. раздел «Слияние деклараций» на с. 279), чтобы расширить глобальный интерфейс `Array<T>`, и добавляем `zip` к уже глобально определенному интерфейсу.

[1] Есть много причин избегать расширения прототипов: переносимость кода, создание более явных графов зависимостей или повышение производительности посредством загрузки только действительно используемых типов. Как бы то ни было, безопасность больше не входит в число этих причин.

Поскольку файл ничего явно не импортирует и не экспортирует, а находится в режиме скриптов (см. подраздел «Режим модулей против режима скриптов» на с. 273), мы смогли расширить глобальный интерфейс `Array` напрямую, объявив интерфейс с таким же именем, как и существующий `Array<T>`, и позволив TypeScript позаботиться об их слиянии. Если бы наш файл находился в модульном режиме (например, если нужно что-либо импортировать (`import`) для реализации `zip`), пришлось бы оборачивать глобальное расширение в декларацию типа `declare global` (см. раздел «Декларации типов» на с. 283):

```
declare global {
    interface Array<T> {
        zip<U>(list: U[]): [T, U][]
    }
}
```

`global` — это особое пространство имен, содержащее все глобально определенные значения (все, что вы можете использовать в модульном режиме без необходимости сначала это импортировать (см. главу 10), что дает возможность расширять имена глобально из файла, находящегося в модульном режиме.

❷ Затем реализуем метод `zip` в прототипе `Array`. Мы используем тип `this`, поэтому TypeScript правильно выводит тип `T` массива, в котором мы вызываем `.zip`.

❸ Поскольку TypeScript выводит возвращаемый тип отображающей функции как `(T | U)[]` (TypeScript не осознает, что на деле это всегда кортеж с `T` в нулевом индексе и `U` в первом), мы используем утилиту `tuple` (см. подраздел «Улучшение вывода типов для кортежей» на с. 177) для создания кортежа без утверждения типа.

Обратите внимание, что, объявляя интерфейс `Array<T>`, мы увеличиваем глобальное пространство имен `Array` для всего проекта TypeScript. Это означает, что, даже если мы не импортируем zip.ts из файла, TypeScript решит, что `[].zip` доступен. Но для расширения `Array.prototype` нужно гарантировать, что любой файл, использующий `zip`, сначала загружает zip.ts, чтобы установить метод `zip` в `Array.prototype`. Как это сделать?

Легко: мы обновим tsconfig.json, чтобы явно исключить zip.ts из проекта, и побудим потребителей явно импортировать его:

```
{
    *exclude*: [
        "./zip.ts"
    ]
}
```

Теперь мы можем безопасно использовать `zip` по своему усмотрению:

```
import './zip'

[1, 2, 3]
    .map(n => n * 2)         // number[]
    .zip(['a', 'b', 'c'])    // [number, string][]
```

При запуске этот код сначала производит отображение, а затем сжатие массива:

```
[
    [2, 'a'],
    [4, 'b'],
    [6, 'c']
]
```

## Итоги

В этой главе вы рассмотрели наиболее продвинутые возможности системы типов TypeScript: от вариантности до вывода типов на основе потока команд, уточнения, расширения типов, тотальности, а также отображенных и условных типов. Затем вы изучили несколько продвинутых паттернов работы с типами: имитацию номинальных типов с помощью маркировок, использование возможностей распределения свойств условных типов для оперирования ими на уровне типов — и в завершение научились безопасно расширять прототипы.

Это нормально, если вы не все поняли или не все запомнили, — вернитесь к этой главе позже и используйте ее как источник ответов, когда будете искать более безопасный способ выражения своих задумок.

# Упражнения к главе 6

1. Для каждой пары приведенных ниже типов определите их совместимость и объясните свой выбор. Рассуждайте выражениями подтипизации и вариантности, а в случае сомнений обратитесь к правилам из начала главы (если и это не поможет, то просто напечатайте их в редакторе, чтобы проверить):

   а) `1` и `number`;

   б) `number` и `1`;

   в) `string` и `number | string`;

   г) `boolean` и `number`;

   д) `number[]` и `(number | string)[]`;

   е) `(number | string)[]` и `number[]`;

   ж) `{a: true}` и `{a: boolean}`;

   з) `{a: {b: [string]}}` и `{a: {b: [number | string]}}`;

   и) `(a: number) => string` и `(b: number) => string`;

   к) `(a: number) => string` и `(a: string) => string`;

   л) `(a: number | string) => string` и `(a: string) => string`;

   м) `E.X` (определен в перечислении `enum E {X = 'X'}`) и `F.X` (определен в перечислении `enum F {X = 'X'}`).

2. При типе объекта `type O = {a: {b: {c: string}}}` каков тип `keyof O`? Что насчет `O['a']['b']`?

3. Напишите тип `Exclusive<T, U>`, который вычисляет типы, находящиеся либо в `T`, либо в `U`, но не в обоих сразу. Например, `Exclusive<1 | 2 | 3, 2 | 3 | 4>` должен разрешаться как `1 | 4`. Распишите пошагово процесс вычисления модулем проверки выражения `Exclusive<1 | 2, 2 | 4>`.

4. Перепишите пример из подраздела «Утверждения явного присваивания» на с. 190, исключив утверждение явного присваивания.

ГЛАВА 7

# Обработка ошибок

> Физик, инженер и программист ехали в машине, как вдруг в ней отказали тормоза. Произошло это на крутом альпийском перевале. Машина продолжала набирать скорость, водитель изо всех сил пытался вписаться в повороты, и от падения спасали лишь оградительные барьеры. Пассажиры были перед лицом неминуемой гибели и уже смирились со своей участью, но внезапно показалась аварийная полоса, на которой удалось благополучно остановиться.
>
> Физик сказал: «Нужно смоделировать трение в тормозных колодках и вызванный этим рост температуры. Так мы сможем разобраться, что произошло».
>
> Инженер продолжил: «У меня есть с собой кой-какие инструменты. Пойду покопаюсь в колодках».
>
> На что программист ответил им: «А давайте проверим воспроизводимость?»
>
> *Аноним*

TypeScript предоставляет массу возможностей, позволяющих сместить ошибки среды выполнения в среду компиляции: начиная с богатой системы типов и заканчивая мощным статическим и символическим анализом. Он работает изо всех сил, чтобы вам не пришлось коротать пятничные вечера за исправлением опечаток в именах переменных и исключений нулевых указателей (и чтобы ваш коллега не опоздал из-за этого на день рождения двоюродной бабушки).

К несчастью, независимо от того, в каком языке вы работаете, иногда исключения все же просачиваются в среду выполнения. TypeScript не под силу препятствовать сбоям в сети и файловой системе, ошибкам обработки пользовательского ввода, переполнению стека или недостаточной памяти. Тем не менее его отличная система типов, несомненно, помогает справляться с ошибками выполнения.

В этой главе я познакомлю вас с наиболее распространенными паттернами представления и обработки ошибок в TypeScript, такими как:

- Возврат `null`.
- Выбрасывание исключений.
- Возврат исключений.
- Тип `Option`.

Выбор тех или иных механизмов остается за вами и зависит от вашего приложения, а я покажу их плюсы и минусы.

## Возврат null

Создадим программу, спрашивающую пользователя о его дне рождения, чтобы интерпретировать эту дату в объект `Date`:

```
function ask() {
    return prompt('When is your birthday?')
}

function parse(birthday: string): Date {
    return new Date(birthday)
}

let date = parse(ask())
console.info('Date is', date.toISOString())
```

Вероятно, стоит проверить указанную пользователем дату, ведь это текстовый запрос:

```
// ...
function parse(birthday: string): Date | null {
    let date = new Date(birthday)
    if (!isValid(date)) {
        return null
    }
    return date
}
```

```
// Проверка допустимости указанной даты
function isValid(date: Date) {
    return Object.prototype.toString.call(date) === '[object Date]'
        && !Number.isNaN(date.getTime())
}
```

Получив результат, первым делом проверим, не является ли он `null`, и только потом применим его:

```
// ...
let date = parse(ask())
if (date) {
    console.info('Date is', date.toISOString())
} else {
  console.error('Error parsing date for some reason')
}
```

Возврат `null` — это наиболее легковесный способ обработки ошибок в типобезопасном режиме. Допустимые данные будут представлены как `Date`, а недопустимые — как `null`. Система типов проверит, чтобы оба варианта были обработаны.

Но в этом подходе мы упускаем некоторую информацию, ведь `parse` не сообщает точную причину сбоя операции. Из-за этого инженер, производящий отладку, будет вынужден копаться в логах, а пользователь получит всплывающее сообщение «Ошибка обработки даты по неизвестной причине» вместо «Введите дату в виде ГГГГ/ММ/ДД».

Еще возврат `null` нелегко реализовать: необходимо делать проверку на `null` после каждой операции. Это может вызвать многословность, поскольку именно вам придется делать вложение этих операций и создавать их цепочки.

## Выбрасывание исключений

Давайте вместо возврата `null` выбросим исключение, чтобы обработать конкретные признаки отказа и получить необходимые метаданные для упрощения процесса отладки.

```
// ...
function parse(birthday: string): Date {
    let date = new Date(birthday)
```

```
    if (!isValid(date)) {
        throw new RangeError('Enter a date in the form YYYY/MM/DD')
    }
    return date
}
```

Теперь при использовании этого кода можно с осторожностью перехватывать исключения, чтобы их обработать, не порушив все приложение:

```
// ...
try {
    let date = parse(ask())
    console.info('Date is', date.toISOString())
} catch (e) {
    console.error(e.message)
}
```

Следует внимательно перебросить другие исключения, чтобы не упустить возможные ошибки:

```
// ...
try {
    let date = parse(ask())
    console.info('Date is', date.toISOString())
} catch (e) {
    if (e instanceof RangeError) {
        console.error(e.message)
    } else {
      throw e
    }
}
```

Можно выделить подкласс более конкретных ошибок, чтобы, когда другой инженер изменит `parse` или `ask` для выбрасывания других `RangeError`, отличить наши ошибки от его ошибок:

```
// ...

// Кастомизированные типы ошибок
class InvalidDateFormatError extends RangeError {}
class DateIsInTheFutureError extends RangeError {}

function parse(birthday: string): Date {
```

```
    let date = new Date(birthday)
    if (!isValid(date)) {
      throw new InvalidDateFormatError('Enter a date in the form
      YYYY/MM/DD')
    }
    if (date.getTime() > Date.now()) {
        throw new DateIsInTheFutureError('Are you a timelord?')
    }
    return date
}

try {
    let date = parse(ask())
    console.info('Date is', date.toISOString())
} catch (e) {
    if (e instanceof InvalidDateFormatError) {
        console.error(e.message)
    } else if (e instanceof DateIsInTheFutureError) {
      console.info(e.message)
    } else {
      throw e
    }
}
```

Выглядит неплохо. Теперь можно не только выдавать неконкретный сигнал о сбое, но и использовать кастомизированные ошибки для отображения причины сбоя. Это полезно при вычитывании серверных логов во время отладки или при выдаче пользователям диалоговых окон, содержащих данные о неверных действиях и рекомендации по их устранению. Мы сможем эффективно строить цепочки и делать вложения операций, оборачивая любое их число в один `try... catch` (без проверки после каждой операции).

Удобно ли использовать такой код? Представьте, что большой `try... catch` находится в одном файле, а оставшаяся часть кода — в библиотеке, импортированной извне. Откуда инженер узнает о необходимости перехвата конкретных видов ошибок (`InvalidDateFormatError` и `DateInTheFutureError`) или просто о том, что нужно проверить `RangeError`? (Вспомните, что TypeScript не кодирует исключения как часть сигнатуры функции.) Можно указать это в имени функции (`parseThrows`) или включить в документацию:

```
/**
 * @throws {InvalidDateFormatError} Пользователь некорректно ввел
дату рождения.
 * @throws {DateIsInTheFutureError} Пользователь ввел дату рождения
из будущего.
 */
function parse(birthday: string): Date {
 // ...
```

Но на практике инженер не стал бы оборачивать этот код в `try... catch` и вовсе не проверял бы его на исключения, потому что инженеры ленивы (я — точно), а система типов не сообщает им, что они что-то упустили. Однако иногда, как в нашем примере, ошибки настолько ожидаемы, что последующий код действительно должен их обрабатывать, чтобы они не привели к сбою всей программы.

Как еще мы можем указать потребителям, что они должны обработать случаи как успеха, так и провала?

## Возврат исключений

TypeScript не Java, и он не поддерживает спецификаторы `throws`[1]. Но мы можем смоделировать их возможности с помощью типов объединений:

```
// ...
function parse(
    birthday: string
): Date | InvalidDateFormatError | DateIsInTheFutureError {
    let date = new Date(birthday)
    if (!isValid(date)) {
      return new InvalidDateFormatError('Enter a date in the form
      YYYY/MM/DD')
    }
    if (date.getTime() > Date.now()) {
        return new DateIsInTheFutureError('Are you a timelord?')
    }
    return date
}
```

[1] В Java спецификаторы `throws` указывают, какие типы исключений среды выполнения может выбросить метод, а потребитель должен обработать.

Теперь потребитель вынужден обработать все три случая: `InvalidDateFormatError`, `DateIsInTheFutureError` и удачное считывание. В противном случае при компиляции появится `TypeError`:

```
// ...
let result = parse(ask()) // Либо дата, либо ошибка.
if (result instanceof InvalidDateFormatError) {
    console.error(result.message)
} else if (result instanceof DateIsInTheFutureError) {
  console.info(result.message)
} else {
  console.info('Date is', result.toISOString())
}
```

Мы успешно воспользовались преимуществами системы типов, чтобы:

- Кодировать вероятные исключения в сигнатуре `parse`.
- Сообщить потребителям, какие конкретно исключения могут возникнуть.
- Вынудить потребителей обработать (или перебросить) каждое из исключений.

Ленивые потребители могут избежать обработки каждой отдельной ошибки, но им придется сделать это явно:

```
// ...
let result = parse(ask()) // Либо дата, либо ошибка.
if (result instanceof Error) {
    console.error(result.message)
} else {
    console.info('Date is', result.toISOString())
}
```

Конечно, программа по-прежнему может дать сбой из-за недостаточной памяти или исключения переполнения стека, но для подобных случаев нет эффективных решений.

Возврат исключений — это тоже легковесный подход, который не требует мудреных структур данных, но при этом он достаточно информативен, чтобы потребители понимали, какой вид сбоя представляет ошибка и куда обратиться для получения дополнительной информации.

Обратная же сторона в том, что связывание в цепочку и вложение операций, выдающих ошибки, может превратиться в громоздкий код. Если функция возвращает `T | Error`, то любая ее функция-потребитель имеет два выхода.

1. Явно обработать `Error1`.
2. Обработать `T` (успешный случай) и передать `Error1` далее для обработки ее потребителями. Если делать это в достаточном объеме, список ошибок, требующих обработки потребителем, быстро вырастет:

```
function x(): T | Error1 {
    // ...
}
function y(): U | Error1 | Error2 {
    let a = x()
    if (a instanceof Error) {
        return a
    }
    // Сделать что-нибудь с a
}
function z(): U | Error1 | Error2 | Error3 {
    let a = y()
    if (a instanceof Error) {
        return a
    }
    // Сделать что-нибудь с a
}
```

Это громоздкий код, но он дает высокий уровень безопасности.

## Тип Option

Исключения можно описывать с помощью особых типов данных. У этого подхода есть свои проблемы (в частности, код может не распознавать особые типы), но он дает возможность связывать цепочки операций над потенциально ошибочными вычислениями. Три наиболее популярные опции — это типы `Try`, `Option`[1] и `Either`. В этой главе мы рассмотрим только тип `Option`[2], потому что остальные типы с ним схожи.

[1] Альтернативное название — тип `Maybe`.

[2] Погуглите «тип `try`» или «тип `either`» для более подробного ознакомления с ними.

Обратите внимание, что `Try`, `Option` и `Either` не являются встроенными в среду JavaScript, подобно `Array`, `Error`, `Map` или `Promise`. Чтобы их использовать, найдите реализации на NPM или напишите их самостоятельно.

Тип `Option` пришел из таких языков, как Haskell, OCaml, Scala и Rust. Вместо возврата значения он обеспечивает возврат контейнера, который может содержать значение или нет. У контейнера есть ряд методов, позволяющих строить цепочки операций даже при отсутствии значения. Контейнер — это любая структура данных, способная содержать значение. Например, массив:

```
// ...
function parse(birthday: string): Date[] {
    let date = new Date(birthday)
    if (!isValid(date)) {
        return []
    }
    return [date]
}

let date = parse(ask())
date
    .map(_ => _.toISOString())
    .forEach(_ => console.info('Date is', _))
```

Как вы могли заметить, у `Option` такая же проблема, как у возврата `null`, — он не сообщает потребителю причину возникновения ошибки, а только сигнализирует о том, что что-то пошло не так.

Тип `Option` пригождается, когда нужно сделать несколько операций подряд, каждая из которых может провалиться.

Например, ранее мы предположили, что `prompt` всегда завершается успешно, а `parse` может провалиться. Но что, если `prompt` тоже допускает провал? Если пользователь отменил указание даты рождения, возникнет ошибка и вычисления прекратятся. Такой случай можно смоделировать с... другим `Option`!

```
function ask() {
    let result = prompt('When is your birthday?')
    if (result === null) {
        return []
    }
    return [result]
}
// ...
ask()
    .map(parse)
    .map(date => date.toISOString())
      // Ошибка TS2339: свойство 'toISOString' не существует
      // в типе 'Date[]'.
    .forEach(date => console.info('Date is', date))
```

Так, это не сработало. Поскольку мы отобразили массив `Date (Date[])` на массив массивов `Date (Date[][])`, то прежде, чем продолжить, нужно произвести уплощение обратно в массив `Date`:

```
flatten(ask()
    .map(parse))
    .map(date => date.toISOString())
    .forEach(date => console.info('Date is', date))
// Уплощает массив массивов в массив.
function flatten<T>(array: T[][]): T[] {
    return Array.prototype.concat.apply([], array)
}
```

Все перепуталось. Типы сообщают мало (перед нами обычный массив), и при беглом взгляде сложно понять, что происходит с кодом. Для исправления обернем наши действия — помещение значения в контейнер, выражение способа взаимодействия с этим значением и выражение способа получения его обратно из контейнера — в особый тип данных. Как только мы закончим с его реализацией, то сможем использовать его так:

```
ask()
    .flatMap(parse)
    .flatMap(date => new Some(date.toISOString()))
    .flatMap(date => new Some('Date is ' + date))
    .getOrElse('Error parsing date for some reason')
```

Определим тип `Option` следующим образом:

- `Option` — это интерфейс, реализованный двумя классами: `Some<T>` и `None` (рис. 7.1). `Some<T>` — это `Option`, содержащий значение типа `T`, а `None` — это `Option` без значения, представляющий сбой.
- `Option` — это и тип, и функция одновременно. В качестве типа — это интерфейс, который просто выступает в роли супертипа для `Some` и `None`. В качестве функции — это способ создать новое значение типа `Option`.

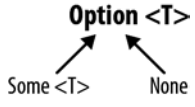


**Рис. 7.1.** Option<T> имеет два случая: Some<T> и None

Начнем с оформления типов:

```
interface Option<T> {} ❶
class Some<T> implements Option<T> { ❷
    constructor(private value: T) {}
}
class None implements Option<never> {} ❸
```

❶ `Option<T>` — это интерфейс, который мы разделим между `Some<T>` и `None`.

❷ `Some<T>` представляет успешную операцию, завершившуюся значением. Так же как и использованный недавно массив, `Some<T>` — это контейнер для данного значения.

❸ `None` представляет провалившуюся операцию и не содержит значения.

Эти типы равнозначны следующим типам, использованным в реализации `Option`, основанной на массиве:

- `Option<T>` — это `[T] | []`.
- `Some<T>` — это `[T]`.
- `None` — это `[]`.

Что можно делать с `Option`? В случае элементарной реализации определить всего две операции:

`flatMap`

Способ цепной обработки операций для потенциально пустого `Option`.

`getOrElse`

Способ извлечения значения из `Option`.

Определим эти операции в интерфейсе `Option`. `Some<T>` и `None` потребуют отдельных реализаций:

```
interface Option<T> {
    flatMap<U>(f: (value: T) => Option<U>): Option<U>
    getOrElse(value: T): T
}
class Some<T> extends Option<T> {
    constructor(private value: T) {}
}
class None extends Option<never> {}
```

Произойдет следующее:

- `flatMap` получит функцию `f`, которой присваивается значение типа `T` (тип значения, содержащегося в `Option`), и возвратит `Option U`. `flatMap` вызовет `f` со значением `Option` и возвратит новый `Option<U>`.
- `getOrElse` получит значение по умолчанию того же типа `T`, что и значение, содержащееся в `Option`, и возвратит либо значение по умолчанию (если `Option` является пустым `None`), либо значение `Option` (если `Option` — это `Some<T>`).

Вооружившись типами, реализуем эти методы для `Some<T>` и `None`:

```
interface Option<T> {
    flatMap<U>(f: (value: T) => Option<U>): Option<U>
    getOrElse(value: T): T
}
class Some<T> implements Option<T> {
    constructor(private value: T) {}
    flatMap<U>(f: (value: T) => Option<U>): Option<U> { ❶
        return f(this.value)
    }
    getOrElse(): T { ❷
        return this.value
    }
}
```

```
class None implements Option<never> {
    flatMap<U>(): Option<U> { ❸
        return this
    }
    getOrElse<U>(value: U): U { ❹
        return value
    }
}
```

❶ При вызове `flatMap` для `Some<T>` передаем ей функцию `f`, которую `flatMap` вызывает со значением `Some<T>`, чтобы произвести новую `Option` нового типа.

❷ Вызов `getOrElse` для `Some<T>` просто возвращает значение `Some<T>`.

❸ Поскольку `None` представляет проваленное вычисление, вызов для него `flatMap` возвращает `None`: как только вычисление проваливается, мы не можем произвести восстановление (по крайней мере не с текущей реализацией `Option`).

❹ Вызов `getOrElse` для `None` всегда возвращает значение, переданное в `getOrElse`.

Можно более качественно определить типы. Если известно только, что есть `Option` и функция из `T в Option<U>`, то `Option<T>` всегда будет производить `flatMap` в `Option<U>`. Но если вы знаете, что есть `Some<T>` или `None`, то можете быть более конкретными.

Таблица 7.1 показывает типы, которые нужны при вызове `flatMap` для двух типов `Option`.

**Таблица 7.1.** Результат вызова .flatMap(f) для Some<T> и None

| | Из Some<T> | Из None |
|---|---|---|
| В Some<U> | Some<U> | None |
| В None | None | None |

Мы знаем, что отображение `None` заканчивается `None`, а отображение `Some<T>` приводит либо к `Some<T>`, либо к `None`, в зависимости от того, что возвращает вызов `f`. Добавим к этим сведениям перегруженные сигнатуры, чтобы определить для `flatMap` более конкретные типы:

```
interface Option<T> {
    flatMap<U>(f: (value: T) => None): None
    flatMap<U>(f: (value: T) => Option<U>): Option<U>
    getOrElse(value: T): T
}
class Some<T> implements Option<T> {
    constructor(private value: T) {}
    flatMap<U>(f: (value: T) => None): None
    flatMap<U>(f: (value: T) => Some<U>): Some<U>
    flatMap<U>(f: (value: T) => Option<U>): Option<U> {
        return f(this.value)
    }
    getOrElse(): T {
        return this.value
    }
}
class None implements Option<never> {
    flatMap(): None {
        return this
    }
    getOrElse<U>(value: U): U {
        return value
    }
}
```

Осталось реализовать функцию `Option` для создания новых `Option`. Мы уже реализовали тип `Option` в качестве интерфейса и теперь займемся одноименной функцией (вспомните, что в TypeScript есть два раздельных пространства имен для типов и значений). Нечто подобное мы делали в подразделе «Паттерн объект-компаньон» на с. 176. Если пользователь передаст `null` или `undefined`, мы вернем ему `None`. В противном случае — вернем `Some`. Напомню, что для этого мы используем перегрузку сигнатуры:

```
function Option<T>(value: null | undefined): None ❶
function Option<T>(value: T): Some<T> ❷
function Option<T>(value: T): Option<T> { ❸
    if (value == null) {
        return new None
    }
    return new Some(value)
}
```

❶ Если пользователь вызывает `Option` с `null` или `undefined`, то возвращаем `None`.

❷ В противном случае возвращаем `Some<T>`, где `T` — это тип значения, переданного пользователем.

❸ Самостоятельно вычисляем верхнюю грань для двух перегруженных сигнатур. Для `null | undefined` и `T` — это `T | null | undefined`, что упрощается до `T`. Верхняя же грань для `None` и `Some<T>` — это `None | Some<T>`, и для нее уже есть имя: `Option<T>`.

Вот и все. Мы вывели полностью рабочий и минимальный тип `Option`, который позволит безопасно выполнять операции со значениями, допускающими `null`. Используется он так:

```
let result = Option(6)                              // Some<number>
    .flatMap(n => Option(n * 3))                    // Some<number>
    .flatMap(n => new None)                         // None
    .getOrElse(7)                                   // 7
```

Вернемся к примеру с днем рождения:

```
ask()                                               // Option<string>
    .flatMap(parse)                                 // Option<Date>
    .flatMap(date => new Some(date.toISOString())) // Option<string>
    .flatMap(date => new Some('Date is ' + date))  // Option<string>
    .getOrElse('Error parsing date for some reason')  // string
```

`Option` — это мощный способ работы с сериями операций, которые могут не завершиться успехом. Он обеспечивает типобезопасность и посредством системы типов сообщает потребителям о возможном провале операции.

Но `Option` имеет свои недостатки. Он сообщает о провале с помощью `None` и не дает информацию о причине сбоя. Также он не допускает взаимодействия с кодом, не использующим `Option` (для такого кода придется явно оборачивать API, чтобы они возвращали `Option`).

И тем не менее у нас получился аккуратный код. Добавленные перегрузки позволяют сделать то, что невозможно выразить в большинстве языков, и даже в тех, которые используют тип `Option` для работы с допускающими `null` значениями. Ограничив `Option` посредством перегруженных сигнатур вызовов везде, где можно, до `Some` или `None`, мы сделали код безопаснее

и создали Haskell-программистам лишний повод для зависти. Вы заслужили перерыв и бутылочку холодненького.

## Итоги

В этой главе вы рассмотрели сообщения об ошибках и восстановление кода после их обнаружения с помощью возврата `null`, выбрасывания исключений, возврата исключений и типа Option. Теперь у вас есть набор подходов для безопасной работы с допускающими провал элементами. Выбор за вами, и зависит он от следующего.

- Хотите ли вы просто сообщить о том, что произошел некий сбой (`null`, `Option`), или предпочли бы дать больше информации о причине ошибки (выбрасывание и возврат исключений).
- Хотите ли вы принудить потребителей явно обрабатывать каждое возможное исключение (возврат исключений) или писать меньше рутинного кода по обработке ошибок (выбрасывание исключений).
- Нужен ли вам способ для компоновки ошибок, или требуется только их обработка при появлении (`null`, исключения).

## Упражнение к главе 7

Разработайте способ обработки ошибок для следующего API с помощью одного из паттернов, рассмотренных в этой главе. В этом API все операции могут провалиться — смело обновляйте сигнатуры методов API, чтобы они допускали сбои (можете этого и не делать). Подумайте, в какой последовательности нужно выполнять действия, чтобы эффективно подключить к ним обработку ошибок (например, получение сначала ID зашедшего пользователя, затем списка его друзей и их имен):

```
class API {
    getLoggedInUserID(): UserID
    getFriendIDs(userID: UserID): UserID[]
    getUserName(userID: UserID): string
}
```

ГЛАВА 8

# Асинхронное программирование, конкурентность и параллельная обработка

До сих пор мы рассматривали в основном синхронные программы, то есть такие, которые получают вводные данные, обрабатывают их и выполняются вплоть до завершения за один заход. Пришло время поговорить о строительных блоках действующих приложений — программах, выполняющих сетевые запросы, взаимодействующих с базами данных и файловыми системами, отвечающих на запросы пользователей и разделяющих нагрузку процессора на отдельные потоки выполнения, — все они работают на асинхронных API вроде обратных вызовов, промисов и потоков.

В работе с асинхронными задачами JavaScript проявляет себя во всей красе и держится особняком от других популярных многопоточных языков вроде Java и C++. Движки JavaScript — V8 и SpiderMonkey — объединяют множество заданий в один поток, оставляя другие задачи в ожидании события. Я буду исходить из предположения, что вы используете подобный движок. С позиции конечного пользователя неважно, применяет ли ваш движок цикл событий или же многопоточность, но это имеет значение для нашего взаимопонимания в рамках книги.

Модель конкурентности в цикле событий позволяет JavaScript избегать расплаты за синхронизированные типы данных, мьютексы, семафоры и другие многопоточные словечки. Также при запуске JavaScript в несколько потоков редко используется общая память. Ее функцию выполняют передача сообщений между потоками и сериализация данных. Такое представление напоминает Erlang, модель акторов и другие чисто функциональные модели конкурентности, делающие многопоточное программирование в JavaScript надежным.

Но асинхронное программирование усложняет восприятие и анализ программ: вы не можете двигаться строка за строкой и вынуждены следить, когда остановить и переместить выполнение в другое место, а когда возобновить его на прежнем месте.

К счастью, типы TypeScript помогают отслеживать асинхронные действия, а встроенная поддержка async и await позволяет применять знакомое синхронное представление к асинхронным программам. Также TypeScript определяет строгие протоколы передачи сообщений для многопоточных программ (на деле это гораздо проще, чем кажется). Если ничего не поможет, TypeScript сделает вам массаж, пока вы допоздна отлаживаете асинхронный код коллеги (нужно установить флажок компилятора).

Но прежде, чем мы начнем работать с асинхронными программами, рассмотрим сам принцип асинхронности в современных движках JavaScript: как мы можем приостанавливать и возобновлять то, что, казалось бы, выполняется в одном потоке?

## Цикл событий

Начнем с примера. Один из двух таймеров сработает через миллисекунду, а другой — через две.

```
setTimeout(() => console.info('A'), 1)
setTimeout(() => console.info('B'), 2)
console.info('C')
```

Что мы увидим в консоли? Будет ли это `A`, `B`, `C`?

Если вы JavaScript-программист, то интуитивно знаете, что ответ — нет. В действительности эти значения появятся в порядке `C`, `A`, `B`. Если вы ранее не работали с JavaScript или TypeScript, то такое поведение может показаться странным. Но все просто. Здесь использована не та модель конкуренции, какую использовала бы команда `sleep` в `C` или планирование работы в другом потоке в *Java*.

На высоком уровне виртуальная машина JavaScript симулирует конкуренцию так (рис. 8.1).

- Главный поток делает вызов внутренних асинхронных API вроде `XMLHTTPRequest` (для запросов AJAX), `setTimeout` (для приостанов-

ки (`sleep`)), `readFile` (для чтения файла с диска) и т. д. Эти API представлены в платформе *JavaScript*, и вы не можете создавать их сами[1].

- Как только вы делаете вызов внутреннего асинхронного API, контроль возвращается главному потоку и выполнение продолжается, как если бы API и не был вызван.
- После завершения асинхронной операции платформа помещает задачу в очередь событий. Каждый поток имеет свою очередь, которая используется для перенаправления результатов асинхронных операций обратно в главный поток. Задача включает в себя метаинформацию вызова и ссылку на функцию обратного вызова из главного потока.
- Как только стек вызовов главного потока пустеет, платформа проверяет наличие ожидающих задач в очереди событий. Если такая задача обнаружена, платформа ее запускает. Это активирует вызов функции, и управление возвращается функции главного потока. Когда стек вызовов, полученный в результате этого вызова, снова опустеет, платформа опять проверит очередь событий на предмет ожидающих задач. Этот цикл повторяется, пока не опустеют стек вызовов и очередь событий и не завершатся все внутренние асинхронные вызовы API.

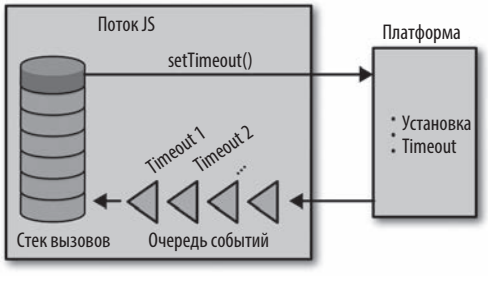


**Рис. 8.1.** Цикл событий в JavaScript: что происходит при вызове асинхронного API

Взяв на вооружение эту информацию, вернемся к примеру с `setTimeout`. Вот что происходит:

1. Мы вызываем `setTimeout`, который вызывает внутренний API таймаута со ссылкой на переданный нами обратный вызов и аргументом `1`.

[1] В принципе, вы это можете, если сделаете ответвление платформы браузера или создадите расширение C++ для NodeJS.

2. Снова вызываем `setTimeout`, который снова вызывает внутренний API тайм-аута со ссылкой на второй переданный нами обратный вызов и аргументом `2`.
3. Выводим `C` в консоль.
4. На заднем плане, спустя по меньшей мере одну миллисекунду, наша платформа JavaScript добавляет в очередь событий задачу, сигнализируя о том, что первый `setTimeout` прошел и его обратный вызов может быть вызван.
5. Спустя еще одну миллисекунду платформа добавляет в очередь вторую задачу для второго обратного вызова `setTimeout`.
6. Поскольку после завершения шага 3 стек вызовов пуст, платформа обращается к очереди событий в поиске ожидающих задач, которые она обнаружит, если шаги 4 и 5 были завершены. Затем для каждой задачи она производит вызов соответствующей функции обратного вызова.
7. Как только оба таймера истекут и очередь событий со стеком вызовов опустеет, программа завершится.

Вот почему в консоли мы увидели `C`, `A`, `B`, а не `A`, `B`, `C`. Миновав эту стартовую точку, мы можем начинать говорить о том, как безопасно типизировать асинхронный код.

## Работа с обратными вызовами

Главный элемент асинхронных *JavaScript*-программ — обратный вызов — привычная функция, которая передается другой функции в виде аргумента. В асинхронных программах эта другая функция задействует обратный вызов по завершении всех своих действий (выполнения сетевого запроса и т. п.). Обратные вызовы, задействованные асинхронным кодом, — это просто функции, и в их сигнатурах типов нет ничего указывающего на то, что они вызываются асинхронно.

Внутренние API NodeJS вроде `fs.readFile` (для асинхронного считывания содержимого файла на диске) и `dns.resolveCname` (для асинхронного разрешения записей CNAME) подчиняются правилу, согласно которому первый параметр — это ошибка или `null`, а второй параметр — это результат `null`.

Так выглядит сигнатура типа `readFile`:

```
function readFile(
    path: string,
    options: {encoding: string, flag?: string},
    callback: (err: Error | null, data: string | null) => void
): void
```

Обратите внимание, что нет ничего особенного как в типе `readFile`, так и в типе `callback`: оба являются регулярными JavaScript-функциями. Если рассмотреть сигнатуру, то в ней ничто не указывает на то, что `readFile` вызывается асинхронно и управление будет передано следующей строке сразу после вызова `readFile` (без ожидания результата вызова).

Для самостоятельного запуска следующего примера установите декларации типов для NodeJS:

```
npm install @types/node --save-dev
```

Подробная информация о сторонних декларациях типов содержится в подразделе «JavaScript, имеющий декларации типов на Definitely-Typed» на с. 302.

Напишем NodeJS-программу, производящую считывание и запись в журнал доступа *Apache*:

```
import * as fs from 'fs'

// Считывание данных из журнала доступа сервера Apache.
fs.readFile(
    '/var/log/apache2/access_log',
    {encoding: 'utf8'},
    (error, data) => {
        if (error) {
            console.error('error reading!', error)
            return
        }
        console.info('success reading!', data)
    }
)
```

```
// Параллельное записывание данных в тот же журнал доступа.
fs.appendFile(
    '/var/log/apache2/access_log',
    'New access log entry',
    error => {
        if (error) {
            console.error('error writing!', error)
        }
    })
```

Если вы не TypeScript- или JavaScript-разработчик, не знакомы с принципом асинхронной работы встроенных API NodeJS и не знаете, что очередность вызовов API в коде не определяет очередность фактического выполнения операций файловой системы, то вы не заметили небольшой баг. Дело в том, что первый вызов `readFile` может не вернуть журнал доступа с добавленной нами строкой. Это будет зависеть от степени загруженности файловой системы в момент выполнения кода.

Возможно, вы поняли, что функция `readFile` асинхронна, на основе опыта, документации NodeJS или принципа NodeJS: функция асинхронная, если ее последний аргумент — это функция, получающая два аргумента — `Error | null` и `T | null`, именно в таком порядке. А может, вы просто узнали об этом от соседа-программиста, который поведал вам свою историю решения связанных с этими нюансами проблем.

Как бы то ни было, типы тут были ни при чем.

Мало того что типы не помогают постичь синхронность функций, обратные вызовы тоже сложно упорядочиваемы и могут выстраиваться в так называемые пирамиды обратных вызовов:

```
async1((err1, res1) => {
    if (res1) {
        async2(res1, (err2, res2) => {
            if (res2) {
                async3(res2, (err3, res3) => {
                    // ...
                })
            }
        })
    }
})
```

При упорядочении операций обычно по завершении одной вы автоматически переходите к другой ниже по цепочке, выходя при возникновении ошибки. С обратными вызовами все это нужно делать вручную. Если вы начнете учитывать синхронные ошибки (к примеру, NodeJS при передаче неверно типизированного аргумента производит `throw` вместо того, чтобы вызвать предоставленный вами обратный вызов с объектом `Error`), правильное упорядочение обратных вызовов может породить множество новых ошибок.

Упорядочение — это всего лишь одна из операций, которые вам может понадобиться запускать посредством асинхронных задач. Также можно запускать функции параллельно, чтобы знать, когда они завершаются, или устраивать им проверку скорости, чтобы узнать, какая первая финиширует, и т. д.

При отсутствии более продуманных абстрактных решений для оперирования с асинхронными задачами работа с множественными обратными вызовами, так или иначе зависящими друг от друга, может быстро привести к беспорядку.

Подытожим:

- Используйте обратные вызовы для простых асинхронных задач.
- Обратные вызовы перерастают в проблему при работе с большим числом асинхронных задач.

## Промисы как здоровая альтернатива

К счастью, мы не первые из программистов, кто столкнулся с ограничениями обратных вызовов. В этом разделе мы исследуем промисы, которые позволяют абстрагироваться от асинхронности для ее создания, упорядочения и т. д. Даже если вы уже работали с промисами или фьючерсами (будущими значениями), вы найдете здесь полезные упражнения для понимания принципов их работы.

Начнем с примера использования промисов для добавления данных в файл и последующего считывания результата:

```
function appendAndReadPromise(
                    path: string, data: string): Promise<string> {
```

```
    return appendPromise(path, data)
        .then(() => readPromise(path))
        .catch(error => console.error(error))
}
```

Большинство современных JavaScript-платформ имеют встроенную поддержку промисов. В этом разделе мы разработаем собственную частичную реализацию `Promise` в качестве упражнения, но на практике лучше использовать встроенный или готовый ее вариант. Проверьте по ссылке https://kangax.github.io/compat-table/es6/#test-Promise, поддерживает ли ваша любимая платформа промисы, или перескочите к разделу «Библиотеки», чтобы узнать об использовании полифилов в платформах, которые не поддерживают промисы по умолчанию.

Заметьте, что здесь отсутствует пирамида обратных вызовов, поскольку мы эффективно линеаризовали нужные действия в единую понятную цепочку асинхронных заданий. Как только одно из них завершается успешно — запускается следующее. Если же первое проваливается, мы перескакиваем к условию `catch`. В случае с API, основанным на обратных вызовах, это выглядело бы примерно так:

```
function appendAndRead(
    path: string,
    data: string
    cb: (error: Error | null, result: string | null) => void
) {
    appendFile(path, data, error => {
        if (error) {
            return cb(error, null)
        }
        readFile(path, (error, result) => {
            if (error) {
                return cb(error, null)
            }
            cb(null, result)
        })
    })
}
```

Создадим API Promise.

Все начинается весьма скромно:

```
class Promise {
}
```

`new Promise` получает функцию `executor` (исполнитель). Реализация `Promise` будет вызывать ее с двумя аргументами: функцией `resolve` и функцией `reject`:

```
type Executor = (
    resolve: Function,
    reject: Function
) => void

class Promise {
    constructor(f: Executor) {}
}
```

Как работают `resolve` и `reject`? Прежде чем разобраться в этом, подумайте, как самостоятельно обернуть основанный на обратных вызовах API NodeJS вроде `fs.readFile`, в API, основанный на `Promise`. Мы используем `API fs.readFile`, встроенный в NodeJS, следующим образом:

```
import {readFile} from 'fs'

readFile(path, (error, result) => {
    // ...
})
```

После обертывания этого API в нашу реализацию `Promise` он будет выглядеть так:

```
import {readFile} from 'fs'

function readFilePromise(path: string): Promise<string> {
return new Promise((resolve, reject) => {
    readFile(path, (error, result) =>
        {
            if (error) {
                reject(error)
            } else {
```

```
                resolve(result)
            }
        })
    })
}
```

Тип параметра функции `resolve` зависит от API (в этом случае тип его параметра будет таким же, как и тип `result`), а тип параметра функции `reject` всегда является типом `Error`. Возвращаясь к нашей реализации, обновим код, заменив небезопасные типы `Function` на более конкретные:

```
type Executor<T, E extends Error> = (
    resolve: (result: T) => void,
    reject: (error: E) => void
) => void
// ...
```

Если мы хотим с первого взгляда понимать, к какому типу будет приводиться `Promise` (например, `Promise<number>` представляет асинхронную задачу, завершающуюся `number`), то нужно сделать `Promise` обобщенным и передать его параметры типа ниже, в тип `Executor`, находящийся в конструкторе:

```
// ...
class Promise<T, E extends Error> {
    constructor(f: Executor<T, E>) {}
}
```

Пока все хорошо. Мы определили API конструктора `Promise` и разобрались, какие здесь фигурируют типы. Теперь составим цепочку: какие операции нужно выразить, чтобы выполнить последовательность нескольких `Promise`, распространить их результаты и перехватить исключения? Если вы вернетесь к первому примеру кода в начале раздела, то увидите `then` и `catch`, которые как раз для этого и используются. Добавим их в наш тип `Promise`:

```
// ...
class Promise<T, E extends Error> {
    constructor(f: Executor<T, E>) {}
    then<U, F extends Error>(g: (result: T) =>
                             Promise<U, F>): Promise<U, F>
    catch<U, F extends Error>(g: (error: E) =>
                              Promise<U, F>): Promise<U, F>
}
```

then и catch — это два способа упорядочения нескольких Promise: then отображает успешный результат Promise в новый Promise[1], а catch производит восстановление при отказе, отображая ошибку также в новый Promise.

Использование then:

```
let a: () => Promise<string, TypeError> = // ...
let b: (s: string) => Promise<number, never> = // ...
let c: () => Promise<boolean, RangeError> = // ...

a()
    .then(b)
    .catch(e => c()) // b не будет ошибкой, так что это случай ошибки a
    .then(result => console.info('Done', result))
    .catch(e => console.error('Error', e))
```

Поскольку тип второго аргумента b — это never (значит, b никогда не выбросит ошибку), то первое условие catch будет вызвано, только если ошибка произойдет в a. Но при использовании Promise нам не нужно беспокоиться о том, что a может выбросить ошибку, а b — нет. Если a завершится успешно, мы отобразим Promise на b — или в противном случае снова перескочим к первому условию catch и отобразим Promise на c. Если c завершится успешно, мы зарегистрируем Done, а если произойдет отказ, то снова произведем catch. Это имитация работы регулярных инструкций try... catch, которая выполняет для асинхронных задач ту же функцию, что и try... catch для синхронных (рис. 8.2).

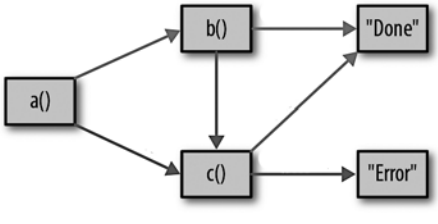


**Рис. 8.2.** Диаграмма состояний Promise

[1] Внимательные читатели заметят, насколько этот API похож на API flatMap, разработанный в разделе «Тип Option» на с. 205. Это не совпадение. И Promise, и Option происходят из паттерна монада, известного благодаря функциональному языку Haskell.

Также нужно обработать случай промисов, выбрасывающих действительные исключения (например, `throw Error('foo')`). При реализации `then` и `catch` мы обертываем код в несколько `try... catch` и делаем отказ в условии `catch`. Тем не менее здесь есть некоторые сложности:

1. Каждый `Promise` имеет вероятность отказа, и мы не можем статически это проверить (TypeScript не поддерживает указание в сигнатурах функции, какие она может выбросить исключения).
2. `Promise` не будет постоянно выдавать отказ с `Error`. Поскольку TypeScript вынужден здесь повторить поведение JavaScript, а в JavaScript при выполнении `throw` можно выбросить что угодно — строку, функцию, массив, `Promise` и `Error`, —мы предполагаем, что отказ будет подтипом `Error`. Это жертва, дарующая потребителям возможность не выполнять `try... catch` для каждой цепочки промисов (которая может распространятся на несколько файлов или модулей).

С учетом сказанного немного упростим наш тип `Promise`, исключив типизацию ошибок:

```
type Executor<T> = (
    resolve: (result: T) => void,
    reject: (error: unknown) => void

) => void

class Promise<T> {
    constructor(f: Executor<T>) {}
    then<U>(g: (result: T) => Promise<U>): Promise<U> {
        // ...
    }
    catch<U>(g: (error: unknown) => Promise<U>): Promise<U> {
        // ...
    }
}
```

Теперь это полноценно проработанный интерфейс `Promise`.

Связывание же его с реализациями `then` и `catch` я оставляю для вас в качестве упражнения. Написать корректную реализацию `Promise` поистине непросто. Если вы амбициозны и располагаете несколькими часами свободного времени, то загляните в спецификацию ES2015 для

ознакомления со всеми глубинными принципами построения диаграммы состояний промисов.

## async и await

Промисы — это действительно мощная абстракция для работы с асинхронным кодом. Они настолько популярны, что даже имеют свой собственный синтаксис в JavaScript (следовательно, и в TypeScript) в виде `async` и `await`, благодаря которому можно взаимодействовать с асинхронными операциями тем же способом, что и с синхронными.

Рассматривайте `await` как синтаксический сахар для `.then` на уровне языка. Когда вы ожидаете (`await`) `Promise`, то должны делать это в блоке `async`. И вместо `.catch` вы можете обернуть `await` в регулярный блок `try... catch`.

Скажем, у вас есть следующий промис (мы не рассмотрели `finally` в предыдущем разделе, но он ведет себя ожидаемо и срабатывает после возможного срабатывания `then` и `catch`):

```
function getUser() {
    getUserID(18)
        .then(user => getLocation(user))
        .then(location => console.info('got location', location))
        .catch(error => console.error(error))
        .finally(() => console.info('done getting location'))
}
```

Чтобы преобразовать этот код в `async` и `await`, сначала поместите его в функцию `async`, а затем ожидайте (`await`) результата промиса:

```
async function getUser() {
    try {
        let user = await getUserID(18)
        let location = await getLocation(user)
        console.info('got location', user)
    } catch(error) {
        console.error(error)
    } finally {
```

```
        console.info('done getting location')
    }
}
```

Поскольку `async` и `await` являются особенностями JavaScript, мы не станем на них останавливаться, достаточно будет добавить, что TypeScript их полностью поддерживает и они типобезопасны. Используйте их везде, где есть промисы, чтобы облегчить восприятие цепочек операций и избежать множества `then`. Подробнее об `async` и `await` читайте в документации на MDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function).

## Async-потоки

Промисы подходят для моделирования, упорядочения и создания будущих значений. Но как представить несколько значений, которые станут доступны в нескольких точках в будущем? Это не такой уж экзотический случай — части файла, считываемые из файловой системы; пикселы видеопотока, поступающего с сервера Netflix на ваш ноутбук; множество нажатий клавиш при заполнении формы; друзья, приходящие к вам домой на вечеринку, или бюллетени, вкладываемые в урну для голосования в течение Супервторника. Хоть упомянутые примеры и непохожи, все они являются асинхронными потоками — списками событий, происходящими в некоторых точках будущего.

Есть несколько способов их моделирования. Наиболее распространенные — использование отправителей событий (вроде EventEmiter в NodeJS) и библиотек для реактивного программирования вроде RxJS[1]. Различие этих подходов напоминает различие между обратными вызовами и промисами: события быстры и легковесны, в то время как библиотеки для реактивного программирования более мощны и дают возможность создавать и упорядочивать потоки событий.

[1] Observables — это основополагающий конструктивный элемент в подходе реактивного программирования, позволяющий производить действия со значениями по времени. На момент написания книги рассматривается предложение стандартизировать Observables (https://tc39.es/proposal-observable/). Как только эта особенность пополнит инструментарий JavaScript-инженеров, она обязательно будет включена в одну из следующих редакций этой книги.

В следующем разделе мы рассмотрим отправители событий. Чтобы узнать больше о реактивном программировании, воспользуйтесь документацией предпочтительной для вас библиотеки. Например, RxJS (https://www.npmjs.com/package/@reactivex/rxjs), MostJS (https://github.com/mostjs/core) или xstream (https://www.npmjs.com/package/xstream).

## Отправители событий

На высоком уровне отправители событий предоставляют API, поддерживающие отправку событий в канал и прослушивание событий в этом канале:

```
interface Emitter {

    // Отправка события
    emit(channel: string, value: unknown): void

    // Сделать что-либо после отправки события
    on(channel: string, f: (value: unknown) => void): void

}
```

Отправители событий — это популярный паттерн проектирования в JavaScript. Вы могли сталкиваться с ними при использовании событий DOM, JQuery или NodeJS модуля `EventEmitter`.

В большинстве языков отправители событий, подобные упомянутым, небезопасны. Это вызвано тем, что тип `value` зависит от конкретного канала, а в большинстве языков нельзя использовать типы для выражения такой связи. Если ваш язык не поддерживает одновременно и перегруженные сигнатуры функций, и типы литералов, вам будет сложно говорить: «Это тип события, отправленного в этот канал». Макросы, генерирующие методы для отправки событий и прослушивания каждого канала, являются распространенным обходным решением этой проблемы, но в TypeScript вы можете выразить это естественно и безопасно при помощи системы типов.

Например, клиент NodeRedis (https://github.com/NodeRedis/node-redis) — это NodeJS API, используемый для популярного хранилища данных Redis:

```
import Redis from 'redis'

// Создание нового экземпляра клиента Redis
let client = redis.createClient()

// Прослушивание новых событий, отправленных клиентом
client.on('ready', () => console.info('Client is ready'))
client.on('error', e => console.error('An error occurred!', e))
client.on('reconnecting', params => console.info('Reconnecting...',
                                                 params))
```

Как программисты, использующие библиотеку Redis, мы хотим знать, какие типы аргументов ожидать в обратных вызовах, когда мы используем API `on`. Но поскольку тип каждого аргумента зависит от канала, в который производит отправку Redis, единый тип тут не подойдет. Если бы авторами библиотеки были мы, то простейшим путем добиться безопасности было бы использование перегруженного типа:

```
type RedisClient = {
    on(event: 'ready', f: () => void): void
    on(event: 'error', f: (e: Error) => void): void
    on(event: 'reconnecting',
        f: (params: {attempt: number, delay: number}) => void): void
}
```

Работает неплохо, но выглядит громоздко. Выразим этот код с помощью отображенного типа (см. подраздел «Отображенные типы» на с. 173), выделив определения событий в отдельный тип `Events`:

```
type Events = { ❶
    ready: void
    error: Error
    reconnecting: {attempt: number, delay: number}
}

type RedisClient = { ❷
    on<E extends keyof Events>(
        event: E,
        f: (arg: Events[E]) => void
    ): void
}
```

❶ Определяем единый тип объекта, перечисляющего каждое событие, которое может отправить клиент Redis наряду с аргументами для этого события.

❷ Отображаем тип `Events`, сообщая TypeScript, что `on` может быть вызван с любыми событиями, которые мы определили.

Затем мы можем использовать этот тип, чтобы сделать безопаснее библиотеку Node-Redis, типизировав оба его метода — `emit` и `on` — максимально безопасным образом:

```
// ...
type RedisClient = {
    on<E extends keyof
        Events>( event: E,
        f: (arg: Events[E]) => void
    ): void
    emit<E extends keyof
        Events>( event: E,
        arg: Events[E]
    ): void
}
```

Этот паттерн выделения имен событий и аргументов в отдельную форму и отображение этой формы для генерации слушателей и отправителей на практике является распространенным в TypeScript. Он лаконичен и безопасен, предотвращает опечатки в ключе или аргументе и не дает забыть о передаче аргумента. Также он служит в качестве документации для инженеров, использующих ваш код, ведь их редакторы будут предлагать возможные события, которые они могут прослушивать, и типы параметров обратных вызовов этих событий.

## Типобезопасная многопоточность

Мы поговорили об асинхронных программах, запускаемых в одном потоке CPU. Это класс программ, к которому будет относиться большинство создаваемых вами в JavaScript и TypeScript проектов. Но иногда, при выполнении задач, требующих повышенной производительности CPU, лучше обратиться к параллелизму — разделению работы между несколькими потоками для ускорения их выполнения или сохранения главного

потока свободным и доступным. В этом разделе мы изучим несколько паттернов для написания безопасных параллельных программ как для браузера, так и для сервера.

### ОТПРАВИТЕЛИ В РЕАЛЬНОЙ ЖИЗНИ

Использование отображенных типов для построения типобезопасных отправителей событий — это популярный паттерн. Например, так типизированы события DOM в стандартной библиотеке TypeScript. WindowEventMap — это отображение имени события на тип события, который API `.addEventListener` и `.removeEventListener` отображают для создания лучших, более конкретных типов событий, чем предустановленный тип `Event`:

```
// lib.dom.ts
interface WindowEventMap extends GlobalEventHandlersEventMap {
    // ...
    contextmenu: PointerEvent
    dblclick: MouseEvent
    devicelight: DeviceLightEvent
    devicemotion: DeviceMotionEvent
    deviceorientation: DeviceOrientationEvent
    drag: DragEvent
    // ...
}
interface Window extends EventTarget, WindowTimers,
   WindowSessionStorage, WindowLocalStorage,
   WindowConsole, GlobalEventHandlers, IDBEnvironment,
   WindowBase64, GlobalFetch {
   // ...
addEventListener<K extends keyof WindowEventMap>(
   type: K,
   listener: (this: Window, ev: WindowEventMap[K]) =>
   any, options?: boolean | AddEventListenerOptions
): void
removeEventListener<K extends keyof WindowEventMap>(
   type: K,
   listener: (this: Window, ev: WindowEventMap[K]) =>
   any, options?: boolean | EventListenerOptions
): void
}
```

### В браузере: с помощью веб-работников

Веб-работники — это широко поддерживаемый способ осуществления многозадачности в браузере. Несколько работников являются особыми ограниченными фоновыми потоками, отделенными от основного JavaScript-потока, и используются для выполнения действий, которые иначе привели бы к блокировке основного потока и сделали бы UI недоступным (например, задачи, зависящие от процессора). В то время как асинхронные API вроде `Promise` и `setTimeout` запускают код конкурентно, работники дают возможность запустить код параллельно в другом потоке CPU. Они могут отправлять сетевые запросы, производить запись в файловую систему и т. д., имея при этом всего несколько ограничений.

Чтобы идти в ногу с примерами этого раздела, обязательно обозначьте для TSC, что собираетесь запускать этот код в браузере. Для этого включите `lib dom` в tsconfig.json:

```
{
    "compilerOptions": {
        "lib": ["dom", "es2015"]
    }
}
```

А для кода, запускаемого в веб-работнике, используйте `lib webworker`:

```
{
    "compilerOptions": {
        "lib": ["webworker", "es2015"]
    }
}
```

Если вы используете один tsconfig.json и для скрипта веб-работника, и для главного потока, то включайте сразу обе опции.

Поскольку веб-работники являются браузерным API, их разработчики сделали большой уклон в сторону безопасности — не безопасности типов, которую мы все знаем и любим, а безопасности памяти. Любой, кто использовал C, C++, Objective-C или многопоточные Java и Scala, знаком с проблемами конкурентного управления общей памятью. Когда у вас есть несколько потоков, производящих считывание и запись на один

и тот же участок памяти, легко столкнуться с различными проблемами конкуренции вроде нестабильности, зависания и т. д.

Чтобы сделать код браузера безопасным, минимизировать шансы сбоя и причинения пользователю неприятностей, основным способом взаимодействия между главным потоком и веб-работниками, а также между самими веб-работниками должна стать *передача сообщений*.

Рассмотрим работу API, передающего сообщения. Сначала выделим веб-работник из потока:

```
// MainThread.ts
let worker = new Worker('WorkerScript.js')
```

Затем передадим этому работнику сообщение:

```
// MainThread.ts
let worker = new Worker('WorkerScript.js')

worker.postMessage('some data')
```

Можно передавать практически любой вид данных другому потоку с помощью `API postMessage`[1].

Главный поток клонирует получаемые от вас данные, прежде чем передать их потоку работника[2]. Со стороны веб-работника вы прослушиваете входящие события посредством глобально доступного API `onmessage`:

```
// WorkerScript.ts
onmessage = e => {
    console.log(e.data) // Выводит в консоль 'некие данные'
}
```

---

[1] За исключением функций, ошибок, узлов DOM, дескрипторов свойств, методов получения и изменения значений, а также методов и свойств прототипов. Более подробно можно ознакомиться с этим в спецификации HTML5 (https://html.spec.whatwg.org/multipage/infrastructure.html#safe-passing-of-structured-data).

[2] Также можно использовать API Transferable для передачи конкретных типов данных (вроде `ArrayBuffer`) между потоками по ссылке. В этом разделе мы не будем использовать `Transferable` для явной передачи принадлежности объекта между потоками, но это лишь деталь реализации. Если вы используете `Transferable` в каком-то своем случае, то с точки зрения типобезопасности ничего не потеряете.

Для обратного взаимодействия работника с главным потоком используем глобально доступный `postMessage` и метод `.onmessage` в главном потоке для прослушивания входящих сообщений. Получилось:

```
// MainThread.ts
let worker = new
Worker('WorkerScript.js')
worker.onmessage = e => {
    console.log(e.data) // Выводит в консоль 'Подтверждаю получение:
    "неких данных"'
}
worker.postMessage('some data')

// WorkerScript.ts
onmessage = e => {
    console.log(e.data) // Выводит в консоль 'некие данные'
    postMessage(Ack: "${e.data}")
}
```

Этот API во многом схож с API отправителя событий, рассмотренным в подразделе «Отправители событий» на с. 229. Это простой способ передачи сообщений, но при отсутствии типов мы не знаем, верно ли обработаны все возможные типы сообщений, доступных для отправки.

Поскольку этот API, по сути, всего лишь отправитель событий, то для его типизации можно применить соответствующие техники. В качестве примера создадим простой уровень сообщений для клиента чата, который запустим в потоке работника. Уровень сообщений будет передавать обновления в главный поток, и нам не придется беспокоиться о таких вещах, как обработка ошибок, разрешений и т. д. Мы начнем с определения некоторых типов входящих и исходящих сообщений (главный поток посылает `Commands` потоку работника, который обратно отправляет ему `Events`):

```
// MainThread.ts
type Message = string
type ThreadID = number
type UserID = number
type Participants = UserID[]
type Commands = {
    sendMessageToThread: [ThreadID, Message]
    createThread: [Participants]
```

```
    addUserToThread: [ThreadID, UserID]
    removeUserFromThread: [ThreadID, UserID]
}

type Events = {
    receivedMessage: [ThreadID, UserID, Message]
    createdThread: [ThreadID, Participants]
    addedUserToThread: [ThreadID, UserID]
    removedUserFromThread: [ThreadID, UserID]
}
```

Как можно применить эти типы к API сообщений веб-работников? Проще всего объединить все возможные типы сообщений, а затем включить тип `Message`. Но это довольно утомительно. Для нашего типа `Command` такой вариант будет выглядеть следующим образом:

```
// WorkerScript.ts
type Command = ❶
  | {type: 'sendMessageToThread', data: [ThreadID, Message]} ❷
  | {type: 'createThread', data: [Participants]}
  | {type: 'addUserToThread', data: [ThreadID, UserID]}
  | {type: 'removeUserFromThread', data: [ThreadID, UserID]}

onmessage = e => ❸
    processCommandFromMainThread(e.data)

function processCommandFromMainThread( ❹
    command: Command
) {
    switch (command.type) { ❺
        case 'sendMessageToThread':
            let [threadID, message] =
            command.data console.log(message)
        // ...
    }
}
```

❶ Определяем объединение всех возможных команд и аргументов для них, которые главный поток может отправить потоку работника.

❷ Это просто стандартный тип объединения. При определении длинных типов объединений сопровождение их разделением ( | ) облегчит читаемость.

❸ Берем сообщения, отправленные через нетипизированный API `onmessage`, и делегируем их типизированному API `processCommandFromMainThread`.

❹ `processCommandFromMainThread` берет на себя обработку всех входящих сообщений от главного потока. Это безопасная типизированная обертка для нетипизированного API `onmessage`.

❺ Поскольку тип `Command` является типом размеченного объединения (см. раздел «Типы размеченного объединения» на с. 162), мы используем `switch` для тщательной обработки каждого возможного типа сообщения, которое может отправить главный поток.

Давайте отделим причудливые снежинки API веб-работников от знакомого API `EventEmitter` и сделаем входящие и исходящие типы сообщений менее громоздкими.

Начнем с построения типобезопасной обертки для API `EventEmitter`, которая доступна для браузера в пакете событий на NPM (https://www.npmjs.com/package/events):

```
import EventEmitter from 'events'

class SafeEmitter<
    Events extends Record<PropertyKey, unknown[]> ❶
> {
    private emitter = new EventEmitter ❷
    emit<K extends keyof Events>( ❸
        channel: K,
        ...data: Events[K]
    ) {
        return this.emitter.emit(channel, ...data)
    }
    on<K extends keyof Events>( ❹
        channel: K,
        listener: (...data: Events[K]) => void
    ) {
        return this.emitter.on(channel, listener)
    }
}
```

❶ `SafeEmitter` объявляет обобщенный тип `Events` и `Record`, отображающий `Property Key` (встроенный в TypeScript тип для действительных ключей объектов: `string`, `number` или `Symbol`) в список параметров.

❷ Объявляем `emitter` приватным членом `SafeEmitter` вместо расширения `SafeEmitter`, поскольку сигнатуры для `emit` и `on` более ограничивающие, чем их перегруженные ответные части в `EventEmitters`. Функции являются контрвариантными в их параметрах (вспомните, чтобы функция `a` была совместима с функцией `b`, ее параметры должны быть супертипами их ответных частей в `b`), поэтому TypeScript не позволит объявить перегрузки.

❸ `emit` получает `channel` с аргументами, соответствующими списку параметров, определенному нами в типе `Events`.

❹ Схожим образом `on` получает `channel` и `listener`. `listener` получает переменное число аргументов, соответствующих списку параметров, определенному нами в типе `Events`.

Мы можем использовать `SafeEmitter`, чтобы существенно сократить количество рутинного кода, необходимого для реализации уровня слушателя. Со стороны работника мы делегируем все вызовы `onmessage` отправителю и представляем удобный и безопасный API слушателя потребителям:

```
// WorkerScript.ts
type Commands = {
    sendMessageToThread: [ThreadID, Message]
    createThread: [Participants]
    addUserToThread: [ThreadID, UserID]
    removeUserFromThread: [ThreadID, UserID]
}

type Events = {
    receivedMessage: [ThreadID, UserID, Message]
    createdThread: [ThreadID, Participants]
    addedUserToThread: [ThreadID, UserID]
    removedUserFromThread: [ThreadID, UserID]
}

// Прослушивание событий, поступающих из главного потока.
let commandEmitter = new SafeEmitter   <Commands>()

// Отправка событий обратно в главный поток.
let eventEmitter = new SafeEmitter   <Events>()
```

```
// Обертывание команд, поступающих от главного потока,
// с помощью типобозопасного отправителя событий.
onmessage = command =>
    commandEmitter.emit(
        command.data.type,
            ...command.data.data
)

// Прослушивание событий, созданных работником, и отправка их
// обратно главному потоку.
eventEmitter.on('receivedMessage', data =>
    postMessage({type: 'receivedMessage', data})
)
eventEmitter.on('createdThread', data =>
    postMessage({type: 'createdThread', data})
)
// и т.д.

// Ответ на команду из главного потока sendMessageToThread
commandEmitter.on('sendMessageToThread', (threadID, message) =>
    console.log(OK, I will send a message to threadID ${threadID})
)

// Отправка события обратно главному потоку.
eventEmitter.emit('createdThread', 123, [456, 789])
```

С другой стороны, мы можем использовать API, основанный на `EventEmitter`, для отправки команд обратно из главного потока потоку работника. Заметьте, что если вы используете этот паттерн в собственном коде, то можете рассмотреть применение более полноценного отправителя (вроде `EventEmitter2` от Паоло Фрагомени: https://www.npmjs.com/package/eventemitter2), который поддерживает подстановочные получатели событий, что избавит вас от необходимости самостоятельно добавлять получателей для каждого типа события:

```
// MainThread.ts
type Commands = {
    sendMessageToThread: [ThreadID, Message]
    createThread: [Participants]
    addUserToThread: [ThreadID, UserID]
    removeUserFromThread: [ThreadID, UserID]
}
```

```
type Events = {
    receivedMessage: [ThreadID, UserID, Message]
    createdThread: [ThreadID, Participants]
    addedUserToThread: [ThreadID, UserID]
    removedUserFromThread: [ThreadID, UserID]
}

let commandEmitter = new SafeEmitter   <Commands>()
let eventEmitter = new SafeEmitter   <Events>()

let worker = new Worker('WorkerScript.js')

// Прослушивание событий, поступающих от работника,
// и их переотправка посредством типобезопасного отправителя событий
worker.onmessage = event =>
    eventEmitter.emit(
        event.data.type,
        ...event.data.data
)

// Прослушивание команд, заданных этим потоком, и их отправка работнику
commandEmitter.on('sendMessageToThread', data =>
    worker.postMessage({type: 'sendMessageToThread', data})
)
commandEmitter.on('createThread', data =>
    worker.postMessage({type: 'createThread', data})
)
// и т.д.

// Сделать что-либо, когда работник сообщает о создании нового потока.
eventEmitter.on('createdThread', (threadID, participants) =>
    console.log('Created a new chat thread!', threadID, participants)
)

// Отправка команды работнику
commandEmitter.emit('createThread', [123, 456])
```

Вот и все. Мы создали простую типобезопасную обертку для привычной абстракции отправителя событий, которую можно использовать в различных установках, начиная с событий курсора в браузере и заканчивая взаимодействием между потоками, и сделали передачу сообщений между

ними безопасной. С TypeScript вы можете обернуть в типобезопасный API что угодно.

#### Типобезопасные протоколы

Мы рассмотрели передачу сообщений между двумя потоками. Как расширить эту технику и сообщить, что конкретная команда всегда получает конкретное событие в качестве ответа?

Создадим простой протокол «запрос — ответ» для перемещения вычисления функции между потоками. Просто передать функцию из потока в поток не получится, но можно определить функцию в потоке работника и отправить аргументы ему, а затем вернуть результаты. Например, создадим движок матричной алгебры, поддерживающий три операции: нахождение определителя матрицы, вычисление скалярного произведения двух матриц и обращение матрицы.

Вы уже знаете порядок — начнем с оформления типов для трех операций:

```
type Matrix = number[][]

type MatrixProtocol = {
      determinant: {
        in: [Matrix]
        out: number
    }
    'dot-product': {
        in: [Matrix, Matrix]
        out: Matrix
    }
    invert: {
        in: [Matrix]
        out: Matrix
    }
}
```

Мы определяем матрицы в главном потоке и запускаем все вычисления в работниках. Напомню, что наша цель — обернуть небезопасную операцию (отправку и получение нетипизированных сообщений работником) в безопасную через предоставление потребителю правильно определенного типизированного API. В наивной реализации определяем простой протокол «запрос — ответ», создающий список операций, которые работник

может выполнить, а также указываем ожидаемые типы ввода и вывода[1]. Затем определяем обобщенную функцию `createProtocol`, которая передает `Protocol` и путь к файлу работника и возвращает функцию, получающую `command` в этом протоколе и возвращающую заключительную функцию. Последнюю функцию можно вызвать, чтобы вычислить `command` для конкретного набора аргументов. Хорошо, вот что получилось:

```
type Protocol = { ❶
    [command: string]: {
        in: unknown[]
        out: unknown
    }
}

function createProtocol<P extends Protocol>(script: string) { ❷
  return <K extends keyof P>(command: K) => ❸
     (...args: P[K]['in']) => ❹
    new Promise<P[K]['out']>((resolve, reject) => { ❺
            let worker = new
            worker(script) worker.onerror = reject
            worker.onmessage = event => resolve(event.data.data)
            worker.postMessage({command, args})
        })
}
```

❶ Определяем общий тип `Protocol`, неспецифичный для `MatrixProtocol`.

❷ Вызывая `createProtocol`, передаем ей путь к скрипту работника вместе с конкретным `Protocol`.

❸ `createProtocol` возвращает анонимную функцию, которую затем можно вызвать с `command`. Она является ключом в `Protocol`, который мы привязали в ❷.

❹ Вызываем эту функцию с любым конкретным типом `in` для команды, которую мы передали в ❸.

❺ Согласно нашему протоколу возвращается `Promise` для конкретного типа `out` этой команды. Заметьте, что нужно явно привязать параметр типа к `Promise`, иначе он будет по умолчанию `{}`.

[1] Эта реализация наивна, потому что она создает нового работника каждый раз, когда мы отдаем команду. В реальной жизни вы, вероятно, захотите использовать механизм пула, который будет всегда иметь наготове работников и повторно задействовать освободившихся.

Теперь применим тип `MatrixProtocol` и путь к скрипту веб-работника к `createProtocol` (не будем углубляться в детали вычисления определителя, и я предположу, что вы реализовали это в MatrixWorkerScript.ts). Вернемся к функции, которая запустит конкретную команду в этом протоколе:

```
let runWithMatrixProtocol = createProtocol<MatrixProtocol>(
    'MatrixWorkerScript.js'
)
let parallelDeterminant = runWithMatrixProtocol('determinant')

parallelDeterminant([[1, 2], [3, 4]])
.then(determinant =>
    console.log(determinant) // -2
)
```

Круто, правда? Мы взяли абсолютно небезопасную нетипизированную передачу сообщений между потоками и абстрагировались от нее с помощью полностью типобезопасного протокола «запрос — ответ». Все команды, которые можно запускать с помощью этого протокола, расположены в одном месте (`MatrixProtocol`), а базовая логика (`createProtocol`) находится отдельно от реализации конкретного протокола (`runWithMatrixProtocol`).

Всегда, когда нужно установить сообщение между двумя процессами (неважно, на одной машине или между различными компьютерами сети), используйте типобезопасные протоколы. Хотя этот раздел и помог сформировать некоторое представление о проблемах, решаемых протоколами, в реальной жизни вы, скорее всего, предпочтете использовать существующий инструмент вроде Swagger, gRPC, Thrift или GraphQL (см. раздел «Типобезопасные API» на с. 260).

### В NodeJS: с помощью дочерних процессов

Чтобы следовать примерам этого раздела, установите декларации типов для NodeJS из NPM:

```
npm install @types/node --save-dev
```

Узнать больше об использовании деклараций типов вы можете в подразделе «JavaScript, имеющий декларации типов на DefinitelyTyped» на с. 302.

Типобезопасный параллелизм в NodeJS работает так же, как и в случае с потоками веб-работников в браузере (подраздел «Типобезопасные протоколы» на с. 241). Хотя сам по себе уровень передачи сообщений небезопасен, поверх него легко создать безопасные API. API дочернего процесса NodeJS выглядит так:

```
// MainThread.ts
import {fork} from 'child_process'

let child = fork('./ChildThread.js') ❶

child.on('message', data => ❷
    console.info('Child process sent a message', data)
)

child.send({type: 'syn', data: [3]}) ❸
```

❶ Используем `API fork` для создания нового дочернего процесса.

❷ Прослушиваем сообщения, поступающие от дочернего процесса, с помощью `API on`. Есть несколько вариантов сообщений, которые дочерний процесс может отправить своему родителю. В данном случае нас интересует только сообщение `'message'`.

❸ Для отправки сообщений дочернему процессу используем `API send`.

В дочернем потоке мы прослушиваем сообщения, поступающие из главного потока, при помощи `API process.on`, а отправляем их посредством `process.send`:

```
// ChildThread.ts
process.on('message', data => ❶
    console.info('Parent process sent a message', data)
)

process.send({type: 'ack', data: [3]}) ❷
```

❶ Используем `API on` в глобально определенном `process`, чтобы прослушивать сообщения, поступающие от потока родителя.

❷ Используем `API send` в `process` для отправки сообщений родительскому процессу.

Этот механизм похож на механизм веб-работников, поэтому я оставлю в качестве упражнения реализацию типобезопасного протокола для абстрагирования от межпроцессного взаимодействия в NodeJS.

## Итоги

В этой главе вы рассмотрели основы циклов событий в JavaScript, а также асинхронный код и безопасное выражение его частей в TypeScript: обратных вызовов, промисов, `async` и `await` и отправителей событий. Затем вы изучили многопоточность, а именно передачу сообщений между потоками (в браузере и на сервере) и создание протоколов для обмена данными между потоками.

И снова выбор за вами:

- Для простых асинхронных задач обратные вызовы будут максимально подходящими и простыми.
- Для более сложных задач, требующих упорядочения и параллельности, хорошо подойдут промисы и `async` и `await`.
- Когда промис не справляется (например, при многократном запуске события), обратитесь к отправителям событий или библиотекам реактивных потоков вроде RxJS.
- Для расширения этих техник на несколько потоков используйте отправители событий, типобезопасные протоколы или типобезопасные API (см. раздел «Типобезопасные API» на с. 260).

# Упражнения к главе 8

1. Реализуйте общую функцию `promisify`, получающую любую функцию, которая получает только один аргумент и обратный вызов, а затем оборачивает их в функцию, возвращающую промис. Нужно использовать `promisify` следующим образом (установите декларации типов для NodeJS: `npm install @types/node -- save-dev`):

```
import {readFile} from 'fs'

let readFilePromise = promisify(readFile)
readFilePromise('./myfile.ts')
    .then(result => console.log('success reading file',
          result.toString()))
    .catch(error => console.error('error reading file', error))
```

2. В подразделе «Типобезопасные протоколы» на с. 241 мы вывели половину протокола для типобезопасной матричной алгебры. Учитывая эту половину протокола, выполняемую в главном потоке, реализуйте вторую половину, выполняемую в потоке веб-работника.

3. Используйте отображенный тип (как в подразделе «В браузере: с помощью веб-работников» на с. 233) для реализации типобезопасного протокола передачи сообщений для `child process` в NodeJS.

ГЛАВА 9

# Фронтенд- и бэкенд-фреймворки

Вы можете самостоятельно создать каждую часть приложения с нуля — сетевые уровни и уровни баз данных на сервере, фреймворк пользовательского интерфейса и программное решение по управлению состояниями во фронтенде, — но лучше этого не делать, ведь потом будет сложно разбираться с деталями. К счастью, множество таких деталей уже описано другими инженерами фронтенд- и бэкенд-разработки. Пользуясь существующими инструментами, библиотеками и фреймворками, вы сможете быстро переключаться и обрести уверенность при создании собственных приложений.

Эта глава посвящена некоторым наиболее популярным инструментам и фреймворкам для решения распространенных проблем, возникающих как на стороне клиента, так и на стороне сервера. Мы обсудим назначение каждого фреймворка и способы их безопасной интегрирации в TypeScript-приложение.

## Фронтенд-фреймворки

TypeScript очень хорошо вписывается в мир фронтенд-приложений. Имея богатую поддержку JSX и возможность безопасно моделировать изменяемость, TypeScript предоставляет приложениям структуру и безопасность, а также облегчает написание верного, легко обслуживаемого кода в быстро меняющейся среде фронтенд-разработки.

Все встроенные API DOM типобезопасны. Чтобы использовать их из TypeScript, просто включите их декларации типов в файл tsconfig.json проекта:

```
{
    "compilerOptions": {
        "lib": ["dom", "es2015"]
    }
}
```

Это побудит TypeScript добавить при проверке типов встроенные декларации типов браузера и DOM — lib.dom.d.ts.

Опция `lib` в tsconfig.json просто добавляет набор определенных деклараций типов при обработке кода в проекте. Она не производит дополнительного кода или кода JavaScript, который будет существовать при выполнении. К примеру, она не заставит магическим образом работать DOM в вашей среде NodeJS (код будет компилироваться, но при выполнении произойдет сбой). Позаботьтесь о том, чтобы ваши декларации типов соответствовали декларациям, поддерживаемым в используемой среде JavaScript (см. раздел «Создание проекта в TypeScript» на с. 306).

При включенных декларациях типов DOM можно безопасно применять DOM и API браузера, чтобы делать, например, следующее:

```
// Считывать свойства из глобального объекта окна.
let model = {
    url: window.location.href
}

// Создать элемент <input />.
let input = document.createElement('input')

// Наделить его CSS классами.
input.classList.add('Input', 'URLInput')

// Обновлять модель в процессе ввода данных пользователем.
input.addEventListener('change', () =>
     model.url = input.value.toUpperCase()
)

// Внедрить <input /> в DOM.
document.body.appendChild(input)
```

Конечно, этот код прошел проверку типов и поддерживает такие стандартные примочки, как автозаполнение в редакторе. Рассмотрим пример:

```
document.querySelector('.Element').innerHTML // Ошибка TS2531:
// Объект, вероятно, 'null'.
```

TypeScript выбросит ошибку, потому что возвращаемый тип `querySelector` допускает нулевое значение.

Для простых фронтенд-приложений таких низкоуровневых API DOM будет достаточно: они дадут все необходимое ориентированному на типы программированию для браузера. Но большинство реальных фронтенд-приложений используют фреймворки для абстрагирования от принципа работы представления и повторного представления DOM, привязки данных и событий. Следующий раздел даст некоторые указания на то, как эффективно использовать TypeScript с популярными фреймворками.

## React

React один из самых востребованных сегодня фреймворков, обеспечивающих безопасность типов.

Дело в том, что, в отличие от конкурентов, React содержит компоненты (основные структурные элементы приложений), которые определяются и потребляются в TypeScript, поэтому и определения компонентов, и их потребители проходят проверку типов. Вы можете использовать типы, чтобы выразить ограничения, например: «Этот компонент получает ID пользователя и цвет» или «Этот компонент может иметь только дочерние элементы списка».

Такая безопасность в отношении определений компонентов и их потребителей — уровня представления во фронтенд-приложениях — более чем важна. Уровень представления — это традиционно то место, где опечатки, упущенные атрибуты, ошибочно типизированные параметры и неверно вложенные элементы заставляют программистов тысячи часов обновлять браузеры, дергая себя за волосы. День, когда вы начнете типизировать уровни представления с помощью TypeScript и React, — это день, когда вы удвоите свою производительность и производительность всей вашей команды во фронтенд-разработке.

### Праймер JSX

При использовании React вы определяете представления с помощью специального DSL, который называется JavaScript XML (JSX) и внешне очень напоминает HTML. Вы внедряете его прямо в код JavaScript, а затем пропускаете код через компилятор JSX, который переписывает вложенный синтаксис в регулярные вызовы функций JavaScript.

Представьте, что вы создаете приложение-меню для ресторана вашего друга и добавляете список из нескольких позиций в раздел «Ланч» с помощью следующего JSX:

```
<ul class='list'>
    <li>Homemade granola with yogurt</li>
    <li>Fantastic french toast with fruit</li>
    <li>Tortilla Española with salad</li>
</ul>
```

После обработки этого кода компилятором *JSX* вроде плагина Babel transform-react-jsx вы получаете следующий вывод:

```
React.createElement(
    'ul',
    {'class': 'list'},
    React.createElement(
        'li',
        null,
        'Homemade granola with yogurt'
    ),
    React.createElement(
        'li',
        null,
        'Fantastic French toast with fruit'
    ),
    React.createElement( 'li',
        null,
        'Tortilla Española with salad'
    )
);
```

Приятная особенность JSX в том, что вы можете писать код, во многом схожий с обычным HTML, а затем автоматически компилировать его в дружественный для движка JavaScript-формат. Будучи инженером, вы используете только знакомый высокоуровневый описательный DSL без необходимости связываться с деталями реализации.

Не обязательно иметь JSX для работы с React (вы можете писать скомпилированный код непосредственно, и он будет работать). Также можно использовать JSX без React (специфичный вызов функции, в который компилируются теги JSX — `React.createElement` — можно настроить).

Но сочетание React и JSX просто магическое. Оно позволяет прописывать уровни представления не только безопасно, но и с удовольствием.

**TSX = JSX + TypeScript**

Файлы, содержащие JSX, имеют расширение .jsx. А файлы TypeScript, содержащие JSX, имеют расширение .tsx. TSX для JSX — это то же, что и TypeScript для JavaScript, — безопасность в процессе компиляции и вспомогательный уровень, позволяющий повысить производительность и создать код с меньшим числом ошибок. Для включения поддержки TSX в проекте добавьте следующую строку в файл tsconfig.json:

```
{
    "compilerOptions": {
        "jsx": "react"
    }
}
```

Директива `jsx` на момент написания книги имеет три режима.

`react`

Компилирует JSX в файл .js с помощью JSX-программы (по умолчанию это `React.createElement`).

`react-native`

Оставляет JSX без компиляции, но все равно создает файл с расширением .js.

`preserve`

Проверяет типы JSX, но не производит его компиляцию, а создает файл с расширением .jsx.

В начинке TypeScript предусмотрено несколько перехватчиков для типизации TSX подключаемым способом. Ими являются особые типы в пространстве имен global.JSX, куда TypeScript обращается за данными о типах TSX во всей программе. Если вы используете только React, то вам не нужно опускаться на этот уровень. Но если вы создаете собственную библиотеку TypeScript, которая применяет TSX (и не использует React), или вам любопытно узнать, как это делают декларации типов React, — откройте приложение Ж.

### TSC-ФЛАГ: ESMODULEINTEROP

*JSX* производит компиляцию в вызов `React.createElement`, поэтому обязательно импортируйте библиотеку React в каждый файл, где используете *JSX*, чтобы в области действия появилась переменная React:

```
import React from 'react'
```

Ничего страшного, если вы об этом забудете, — TypeScript напомнит:

```
<ul /> // Ошибка TS2304: невозможно найти имя 'React'.
```

Также обратите внимание, что я установил `{"esModuleInterop@: true}` в tsconfig.json для поддержки React без импортирования символа подстановки (*). Если вы повторяете за мной, то либо включите `esModuleInterop` в tsconfig.json, либо импортируйте вместо этого символ подстановки:

```
import * as React from 'react'
```

## Использование TSX с React

React позволяет объявлять два вида компонентов: компоненты функций и компоненты классов. Оба вида получают некоторые свойства и представляют TSX. С позиции потребителя они идентичны.

Объявление и представление компонента функции выглядит так:

```
import React from 'react' ❶

type Props = { ❷
    isDisabled?: boolean
    size: 'Big' | 'Small'
    text: string
    onClick(event: React.MouseEvent<HTMLButtonElement>): void ❸
}

export function FancyButton(props: Props) { ❹
    const [toggled, setToggled] = React.useState(false) ❺
    return <button
        className={'Size-' + props.size}
        disabled={props.isDisabled || false}
```

```
        onClick={event => {
            setToggled(!toggled)
            props.onClick(event)
        }}
    >{props.text}</button>
}
let button = <FancyButton ❻
    size='Big'
    text='Sign Up Now'
    onClick={() => console.log('Clicked!')}
/>
```

❶ Чтобы использовать TSX с React, переносим переменную React в текущую область. Поскольку TSX компилируется в вызовы функции `React.createElement`, нужно импортировать React для ее определения при выполнении.

❷ Объявляем конкретные наборы свойств, которые можно передать в компонент `FancyButton`. `Props` всегда является типом объекта и именуется Props согласно условным обозначениям. Для компонента `FancyButton` свойство `isDisabled` является опциональным, в то время как остальные свойства обязательны.

❸ В React есть собственные наборы оберточных типов для событий DOM. При использовании событий React обязательно пользуйтесь типами событий React, а не стандартными типами событий DOM.

❹ Компонент функции — это просто регулярная функция, имеющая до одного параметра (объект `props`) и возвращающая тип, допускающий представление через `React`. `React` достаточно либерален и может выполнить представление многих типов: TSX, строк, чисел, логических значений, `null` и `undefined`.

❺ Используем перехватчик `useState`, чтобы объявить локальное состояние для компонента функции. `useState` — это один из немногих доступных в react перехватчиков, которые можно совмещать для создания собственных перехватчиков. Заметьте, что мы передали в `useState` изначальное значение `false`, поэтому TypeScript смог вывести эту часть состояния как `boolean`. Если бы вместо этого мы использовали тип, который TypeScript не смог бы вывести (например, массив), нам пришлось бы привязывать тип явно (к примеру, с помощью `useState<number[]>([])`).

❻ Используем синтаксис TSX для создания экземпляра `FancyButton`. Синтаксис `<FancyButton />` почти идентичен вызову `FancyButton`, но он позволяет `React` управлять жизненным циклом `FancyButton` за нас.

Вот и все. TypeScript гарантирует, что:

- JSX правильно сформирован, теги закрыты и верно вложены, а в их именах нет опечаток.
- Когда мы инстнанцируем `<FancyButton />`, то передаем все необходимые и дополнительные опциональные свойства в `FancyButton (size, text` и `onClick)`, причем все эти свойства правильно типизированы.
- Мы не передаем никаких излишних свойств в `FancyButton`, а только необходимые.

Аналогично для компонента класса:

```
import React from 'react' ❶
import {FancyButton} from './FancyButton'

type Props = { ❷
    firstName: string
    userId: string
}

type State = { ❸
    isLoading: boolean
}

class SignupForm extends React.Component<Props, State> { ❹
    state = { ❺
        isLoading: false
}
render() { ❻
    return <> ❼
        <h2>Sign up for a 7-day supply of our tasty
            toothpaste now, {this.props.firstName}.</h2>
        <FancyButton
          isDisabled={this.state.isLoading} size='Big'
          text='Sign Up Now'
          onClick={this.signUp}
        />
```

```
      </>
    }
    private signUp = async () => { ❽
        this.setState({isLoading: true})
        try {
            await fetch('/api/signup?userId=' + this.props.userId)
        } finally {
          this.setState({isLoading: false})
        }
    }
}

let form = <SignupForm firstName='Albert' userId='13ab9g3' /> ❾
```

❶ Как и раньше, импортируем переменную React, чтобы поместить ее в область.

❷ Объявляем тип `Props`, чтобы определить, какие данные нужно передать при создании экземпляра `<SignupForm />`.

❸ Объявляем тип `State` для моделирования локального состояния компонента.

❹ Для объявления компонента класса расширяем базовый класс `React.Component`.

❺ Используем инициализатор свойств, чтобы объявить для локального состояния значения по умолчанию.

❻ Как и с компонентами функций, метод `render` компонента класса возвращает нечто допускающее представление посредством React: TSX, строку, число, логическое значение, `null` или `undefined`.

❼ TSX поддерживает фрагменты с помощью специального синтаксиса `<>…</>`. Фрагмент — это безымянный элемент TSX, который оборачивает другой TSX и является способом избежать представления лишних элементов DOM в тех местах, где нужно вернуть один элемент TSX. Например, метод компонента `React render` должен возвращать один элемент TSX. Для этого можно обернуть код посредством `<div>` или любого другого элемента, но это повлечет за собой ненужную перегрузку в процессе представления.

❽ Определяем `signUp` с помощью стрелочной функции, чтобы убедиться, что `this` в функции не будет перепривязан.

❾ В завершение создаем экземпляр `SignupForm`. Как и при инстанцировании компонентов функции, можно напрямую сделать это посредством `new`, то есть `new SignupForm({firstName: 'Albert', userId: '13ab9g3'})`, но в таком случае `React` не сможет управлять циклом жизни экземпляра `SignupForm` за нас.

Обратите внимание, как в этом примере мы смешиваем и сопоставляем компоненты, основанные на значениях (`FancyButton`, `SignupForm`) с родными (`section, h2`) компонентами. TypeScript же проверяет условия.

- Все обязательные поля определены либо в инициализаторе `state`, либо в конструкторе.
- Все, к чему мы обращаемся в `props` и `state`, действительно существует и имеет тип, который мы предполагаем.
- Мы не производим прямую запись в `this.state`, потому что в React обновления состояний должны производиться через `API setState`.
- Вызов `render` действительно возвращает некий JSX.

С помощью TypeScript можно обеспечить коду React безопасность, а себе — опыт и счастье.

Мы не использовали функцию React `PropTypes`, которая является способом объявления и проверки типов в среде выполнения, потому что TypeScript уже проверил типы при компиляции.

## Angular

*При участии Шьяма Шешадри*

Angular — это более богатый по возможностям фронтенд-фреймворк, чем React, который поддерживает как уровни представления, так и отправку и управление сетевыми запросами, роутинг и внедрение зависимостей. Он изначально создан для работы с TypeScript (по правде говоря, он и написан в нем).

Главной технологической особенностью Angular является компилятор Ahead-of-Time (AoT, или «заблаговременный»), встроенный в его CLI — утилиту командной строки Angular, которая собирает информацию типов,

данную вами в виде аннотаций TypeScript, и использует ее для компиляции кода в обычный JavaScript. Angular применяет к коду множество оптимизаций и трансформаций, прежде чем окончательно передает его в TypeScript и компилирует в JavaScript.

Рассмотрим, как Angular использует TypeScript и свой компилятор AoT, чтобы обезопасить написание фронтенд-приложения.

#### Первая настройка

Чтобы инициализировать новый проект Angular, начните с глобальной установки Angular CLI при помощи NPM:

```
npm install @angular/cli --global
```

Затем используйте Angular CLI для инициализации нового приложения:

```
ng new my-angular-app
```

Следуйте указаниям, и Angular CLI настроит для приложения самое необходимое.

В этой книге мы не будем углубляться в структуру Angular — обратитесь к официальной документации (https://angular.io/docs).

#### Компоненты

Компоненты Angular, как и компоненты React, содержат способ описания структуры DOM компонента, стилизацию и управляющий объект. В Angular вы генерируете шаблон компонента с помощью интерфейса командной строки (CLI), детали которой заполняете сами. Компонент Angular состоит из нескольких файлов:

- Заготовки, описывающей DOM, представляемую компонентом.
- Набора стилей CSS.
- Класса компонента, являющегося классом TypeScript, определяющим логику функционирования компонента.

Начнем с класса компонента:

```
import {Component, OnInit} from '@angular/core'

@Component({
    selector: 'simple-message',
```

```
    styleUrls: ['./simple-message.component.css'],
    templateUrl: './simple-message.component.html'
})
export class SimpleMessageComponent implements OnInit {
    message: string
    ngOnInit() {
        this.message = 'No messages, yet'
    }
}
```

По большей части это стандартный класс TypeScript, имеющий всего несколько отличий, раскрывающих схему воздействия Angular на TypeScript. А именно:

- Перехватчики жизненного цикла Angular доступны в качестве интерфейсов TypeScript — просто объявите, какие вы реализуете (`implement`): `ngOnChanges`, `ngOnInit` и т. д. Затем TypeScript проследит, чтобы вы реализовали методы, отвечающие требованиям выбранных вами перехватчиков. В этом примере мы реализовали интерфейс `OnInit`, который требует реализации метода `ngOnInit`.
- Angular очень активно использует декораторы TypeScript (см. раздел «Декораторы» на с. 135), чтобы объявлять метаданные, относящиеся к вашим компонентам Angular, сервисам и модулям. В этом примере мы использовали `selector`, чтобы объявить, как пользователи могут потребить компонент, а также `templateUrls` и `styleUrl`, чтобы соединить HTML-заготовку и таблицу стилей CSS с компонентом.

**TSC-ФЛАГ: FULLTEMPLATETYPECHECK**

Чтобы активировать проверку типов для заготовок Angular (это стоит сделать), включите `fullTemplateTypeCheck` в tsconfig.json:

```
{
    "angularCompilerOptions": {
        "fullTemplateTypeCheck": true
    }
}
```

Заметьте, что инструкция `angularCompilerOptions` не определяет опции для TSC. Вместо этого она определяет флаги, относящиеся к компилятору AoT.

### Службы

Angular имеет встроенный инструмент внедрения зависимостей (DI), который позволяет фреймворку позаботиться о создании экземпляров служб и их передаче в качестве аргументов компонентам и зависящим от них службам. Это упрощает инстанцирование и тестирование сервисов и компонентов.

Обновим `SimpleMessageComponent`, чтобы внедрить зависимость `MessageService`, ответственную за получение сообщений от сервера:

```
import {Component, OnInit} from '@angular/core'
import {MessageService} from '../services/message.service'

@Component({
    selector: 'simple-message',
    templateUrl: './simple-message.component.html',
    styleUrls: ['./simple-message.component.css']
})
export class SimpleMessageComponent implements OnInit {
    message: string
    constructor(
        private messageService: MessageService
    ) {}
    ngOnInit() {
        this.messageService.getMessage().subscribe(response =>
            this.message = response.message
        )
    }
}
```

Компилятор AoT смотрит на параметры, получаемые `constructor` компонента, извлекает их типы (например, `MessageService`) и ищет в соответствующей карте зависимостей DI зависимость конкретного типа. Затем он создает экземпляр этой зависимости (с помощью `new`), если тот не был создан ранее, и передает его в конструктор экземпляра `SimpleMessageComponent`. Весь этот процесс достаточно сложен, но он может пригодиться при развитии вашего приложения, когда в нем уже будет несколько зависимостей, которые можно использовать исходя из настройки приложения (например, `ProductionAPIService` вместо `DevelopmentAPIService`) или при его тестировании (`MockAPIService`).

Теперь рассмотрим, как определить сервис:

```
import {Injectable} from '@angular/core'
import {HttpClient} from '@angular/common/http'

@Injectable({
    providedIn: 'root'
})
export class MessageService {
    constructor(private http: HttpClient) {}
    getMessage() {
        return this.http.get('/api/message')
    }
}
```

При создании сервиса в Angular мы всегда используем декораторы TypeScript для регистрации его как `Injectible` (допускающим внедрение зависимостей) и определяем, предоставляется ли он на корневом уровне приложения или в субмодуле. Здесь мы зарегистрировали сервис MessageService, позволяющий внедрять его в любую часть приложения. В конструкторе любого компонента сервиса мы можем просто запросить MessageService, и Angular позаботится о его доставке.

Разобравшись с использованием этих двух популярных фреймворков, пора переходить к типизации интерфейса между фронтендом и бэкендом.

## Типобезопасные API

*При участии Ника Нэнса*

Независимо от того, какой фронтенд- или бэкенд-фреймворк вы решите использовать, вам наверняка захочется установить безопасное сообщение между устройствами — от клиента к серверу, от сервера к клиенту, между серверами и между клиентами.

В этой области присутствует несколько сопоставимых по возможностям инструментов и стандартов. Но прежде, чем мы приступим к их изучению, подумайте, как создать собственное решение и какие у него могут быть достоинства и недостатки (мы же все-таки инженеры).

Необходимо решить, как клиенты и серверы со стопроцентной безопасностью типов могут типобезопасно взаимодействовать через нетипизированные сетевые протоколы вроде HTTP, TCP или иные основанные на сокетах аналоги?

Хорошей отправной точкой станет использование типобезопасного протокола (см. подраздел «Типобезопасные протоколы» на с. 241). Он может выглядеть так:

```
type Request =
    | {entity: 'user', data: User}
    | {entity: 'location', data: Location}

// client.ts
async function get<R extends Request>(entity: R['entity'])
: Promise<R['data']> {
    let res = await fetch(/api/${entity})
    let json = await res.json()
    if (!json) {
        throw ReferenceError('Empty response')
    }
    return json
}

// app.ts
async function startApp() {
    let user = await get('user') // User
}
```

Если создать функции `post` и `put` для обратной записи в API REST и добавить тип для каждой сущности, поддерживаемой вашим сервером, то со стороны бэкенда реализуется набор обработчиков для каждого типа сущности, производящий считывание из базы данных, чтобы отправлять клиенту ту сущность, которую он запросил.

Но что случится, если ваш сервер написан не в TypeScript? Или у вас нет возможности разделить тип `Request` между клиентом и сервером (что со временем приведет к утрате их синхронности)? Или вы не используете REST (заменив его на GraphQL)? Или вы осуществляете поддержку и других клиентов, вроде клиентов Swift в iOS или клиентов Java на Android?

Вот где в дело вступают типизированные, кодогенерированные API. Они бывают разными, каждый со своими библиотеками, доступными во множестве языков (включая TypeScript). Например:

- Swagger для API REST.
- Apollo и Relay для GraphQL.
- gRPC и Apache Thrift для RPC.

Эти инструменты опираются на общий источник данных как для сервера, так и для клиента — модели данных для Swagger, схемы GraphQL для Apollo, буферы протоколов для gRPC. Все они затем компилируются в языковые привязки для любого языка (в нашем случае TypeScript).

Эта кодогенерация не дает утратить синхронность между клиентом и сервером. Поскольку каждая платформа использует общую схему, вы не столкнетесь со случаем, когда вы добавили поддержку поля в вашем iOS-приложении, но забыли нажать Merge (Слияние) в пул-реквесте, чтобы добавить для него поддержку сервера.

Углубление в детали каждого фреймворка выходит за рамки этой книги. Выберите один для своего проекта и изучите его документацию.

## Бэкенд-фреймворки

Если вы создаете приложение, взаимодействующее с базой данных, то можете начать с сырых и изначально нетипизированных вызовов SQL или API:

```
// PostgreSQL, используя node-postgres
let client = new Client
let res = await client.query(
    'SELECT name FROM users where id = $1',
    [739311]
) // any

// MongoDB, используя node-mongodb-native
db.collection('users')
    .find({id: 739311})
    .toArray((err, user) =>
        // Пользователь является any
    )
```

При небольшой доле самостоятельного типизирования можно сделать эти API безопаснее и избавиться от большинства any:

```
db.collection('users')
    .find({id: 739311})
    .toArray((err, user: User) =>
        // Пользователь является any
    )
```

Тем не менее сырые API SQL по-прежнему откровенно низкоуровневые и подвержены появлению ошибочных типов или any.

Вот где пригождаются средства объектно-реляционного отображения (ORM)! Они генерируют код на основе схемы базы данных, предоставляя высокоуровневые API для выражения запросов, обновлений, удалений и т. д. В статически типизированных языках эти API типобезопасны, поэтому вам не придется заботиться о корректной типизации элементов и самостоятельно привязывать параметры обобщенных типов.

Рассмотрите применение ORM для обращения к базе данных из TypeScript. На момент написания книги наиболее полноценный ORM для TypeScript — это TypeORM (https://www.npmjs.com/package/typeorm), разработанный Умедом Худойбердиевым. Этот инструмент поддерживает MySQL, PostgreSQL, Microsoft SQL Server, Oracle и даже MongoDB. При его использовании ваш запрос на получение имени пользователя может выглядеть так:

```
let user = await UserRepository
    .findOne({id: 739311}) // User | undefined
```

Заметьте, что высокоуровневый API не только безопасен (исключает воздействия вроде атак по внедрению в SQL), но и типобезопасен по умолчанию (тип, который возвращает `findOne`, известен без необходимости его аннотирования). При работе с базами данных всегда используйте ORM — он легко избавит вас от пробуждений среди ночи из-за того, что поле `saleAmount` оказалось `null`. А произошло это потому, что ночью раньше вы обновили его в `orderAmount`, а ваш коллега затем решил запустить миграцию базы данных в преддверии завершения вашего пул-реквеста, но после этого в районе полуночи пул-реквест провалился, хотя миграция прошла успешно. В итоге сотрудники вашего отдела продаж в Нью-Йорке проснулись с новостями о том, что все заказы клиентов стоят `null` долларов (история из жизни... друга).

## Итоги

В этой главе вы рассмотрели прямое управление DOM, использование React и Angular, добавление безопасности типов в ваши API с помощью инструментов вроде Swagger, gRPC и GraphQL, а также использование TypeORM для безопасного взаимодействия с базой данных.

JavaScript-фреймворки меняются очень быстро, и ко времени, когда вы будете это читать, конкретные API и фреймворки, описанные здесь, могут уже быть на пути в музей. Воспользуйтесь своей выработанной интуицией относительно того, *какие проблемы решают типобезопасные фреймворки*, и определите места, где вы можете воспользоваться другими разработками, чтобы сделать код более безопасным, абстрактным и модульным. Главное, что стоит вынести из этой главы, — это не то, какой из фреймворков лучше, а то, какие виды проблем легче решать с их помощью.

Сочетая типобезопасный код UI, типизированный уровень API и типобезопасный бэкенд, вы избавитесь от целых классов ошибок в приложениях и обеспечите себе здоровый сон.

ГЛАВА 10

# Пространства имен и модули

При написании программы можно выражать инкапсуляцию на нескольких уровнях. На нижнем уровне функции инкапсулируют поведения, а структуры данных (вроде объектов и списков) инкапсулируют данные. Затем можно группировать функции и данные в классы или оставлять их отдельно как утилиты пространства имен с отдельной базой данных или хранилищем. Обычно для каждого файла предполагается один класс или набор утилит. В дальнейшем можно группировать несколько классов или утилит в пакет, чтобы опубликовать на NPM.

Когда мы говорим о модулях, важно провести различие между тем, как их обрабатывают компилятор (TSC) и ваша система сборки, и тем, как в действительности модули загружаются в приложение при выполнении (теги `<script />`, `SystemJS` и т. д.). В JavaScript для каждой из этих задач предусмотрена отдельная программа, что может усложнить осмысление модулей. Стандарты модулей CommonJS и ES2015 упрощают взаимодействие трех программ, а мощные бандлеры вроде Webpack помогают абстрагироваться от трех видов происходящего.

Эта глава сосредоточена на том, как TypeScript разрешает и компилирует модули. Мы отложим рассмотрение работы системы сборки и загрузчиков среды выполнения до главы 12, а здесь обсудим следующее:

- Разделение кода на модули и пространства имен.
- Импортирование и экспортирование кода.
- Масштабирование описанных ранее подходов по мере роста базы данных.
- Различие модульного режима и режима скриптов.
- Слияние деклараций.

Но сначала немного предыстории.

## Краткая история модулей JavaScript

Взаимодействие с JavaScript (например, компиляция) вынуждает TypeScript поддерживать стандарты модулей, используемые JavaScript-программистами.

В самом начале (1995 год) JavaScript не поддерживал никакую систему модулей. Без модулей все объявлялось в глобальном пространстве имен, что серьезно усложняло создание и масштабирование приложений из-за нехватки имен переменных. А без выражения явных API для каждого модуля сложно понять, какие из его частей предполагается использовать, а какие являются приватными деталями реализации.

Для разрешения этих проблем применялись имитации модулей с помощью либо объектов, либо функций-выражений, вызываемых сразу после создания (IIFE) и присваиваемых глобальному `window`. Они были доступны для других «модулей» в приложении и других приложений, размещенных на той же веб-странице. Выглядело это примерно так:

```
window.emailListModule =
    { renderList() {}
    // ...
}

window.emailComposerModule
    = { renderComposer() {}
    // ...
}

window.appModule = {
     renderApp() {
        window.emailListModule.renderList()
        window.emailComposerModule.renderComposer()
    }
}
```

Загрузка и запуск JavaScript блокируют UI браузера, поэтому по мере роста приложения и увеличения его кода браузер пользователя начинает работать все медленнее и медленнее. По этой причине сообразительные программисты начали загружать JavaScript динамически после загрузки страницы, вместо того чтобы делать это единовременно. Спустя пример-

но десять лет после первого релиза JavaScript, библиотеки Dojo (Алекса Рассела, 2004 год), YUI (Томаса Ша, 2005 год) и LABjs (Кайла Симпсона, 2009 год) предоставили загрузчики модулей — способ загружать JavaScript-код по требованию (зачастую в асинхронном режиме) после завершения загрузки начальной страницы. Такой режим подразумевал три условия.

1. Модули должны быть тщательно инкапсулированы, иначе на странице произойдет сбой во время передачи зависимостей.
2. Зависимости между модулями должны быть явными, иначе неясно, какие модули и в каком порядке надо загружать.
3. Модули должны иметь индивидуальный идентификатор внутри приложения, иначе не получится уверенно определить порядок их загрузки.

Загрузка модуля с помощью LABjs выглядела так:

```
$LAB
    .script('/emailBaseModule.js').wait()
    .script('/emailListModule.js')
    .script('/emailComposerModule.js')
```

Примерно в то же время разрабатывалась платформа NodeJS (Райаном Далем, 2009 год). При ее создании были учтены все наболевшие проблемы JavaScript и других языков, поэтому система модулей была внедрена прямо в платформу. Как и любая качественная система модулей, она должна была соответствовать тем же трем критериям, что и загрузчики LABjs и YUI. NodeJS осуществила это с помощью стандарта модулей CommonJS, что выглядело так:

```
// emailBaseModule.js
var emailList = require('emailListModule')
var emailComposer = require('emailComposerModule')

module.exports.renderBase = function() {
    // ...
}
```

Тем временем в сети набирал обороты стандарт модулей AMD (Джеймса Бурка, 2008 год), предоставленный Dojo и RequireJS. Он поддерживал ту же функциональность и имел собственную систему сборки для связки кода JavaScript:

```
define('emailBaseModule',
    ['require', 'exports', 'emailListModule', 'emailComposerModule'],
    function(require, exports, emailListModule, emailComposerModule) {
        exports.renderBase = function() {
            // ...
        }
    }
)
```

Несколько лет спустя появился Browserfy (Джеймса Холлидея, 2011 год), дающий фронтенд-разработчикам возможность использовать CommonJS, де-факто ставший стандартом для связки модулей, а также импорта и экспорта синтаксиса.

Однако в использовании CommonJS присутствовал ряд проблем. Среди них была необходимость синхронности вызовов `require`, а также недочеты алгоритма разрешения модулей при использовании в сети. Вдобавок к этому код, использующий его, в некоторых случаях не допускал статический анализ (TypeScript-программисту здесь надо насторожиться), поскольку `module.exports` может появиться где угодно (даже в мертвых ветках кода) и вызовы `require` могут также появиться в любом месте и содержать произвольные строки и выражения, запрещая статически связать программу JavaScript, проверить, все ли ссылочные файлы действительно существуют, и экспортировать то, что для них заявлено.

Шестое издание языка ECMAScript — ES2015 представило новый стандарт для импорта и экспорта, который имел чистый синтаксис и был статически анализируем. Выглядит это так:

```
// emailBaseModule.js
import emailList from 'emailListModule'
import emailComposer from 'emailComposerModule'

export function renderBase() {
    // ...
}
```

Сегодня мы используем в JavaScript- и TypeScript-коде именно этот стандарт. Но на момент написания книги по умолчанию он поддерживается не в каждой среде выполнения JavaScript. Поэтому нам приходится компилировать его в поддерживаемый формат (CommandJS для сред

NodeJS, глобальные объекты или формат, поддерживающий загрузку модулей для сред браузера).

TypeScript предоставляет несколько способов потреблять и экспортировать код в модуль: с помощью глобальных деклараций, стандартных инструкций ES2015 `import` и `export`, а также посредством обратносовместимых import из модулей CommoJS. Помимо этого, система сборки TSC позволяет компилировать модули для различных сред: глобальных объектов, ES2015, CommonJS, AMD, SystemJS или UMD (сочетания CommonJS, AMD и глобальных объектов — в зависимости от того, какой из них будет доступен в среде потребления).

## import, export

При любых обстоятельствах в TypeScript-коде лучше использовать инструкции ES2015 `import` и `export`, чем CommonJS, глобальные объекты или модули с областями имен. Выглядят они в точности как в старом JavaScript:

```
// a.ts
export function foo() {}
export function bar() {}

// b.ts
import {foo, bar} from './a'
foo()
export let result = bar()
```

Стандарт модулей ES2015 поддерживает экспорт по умолчанию:

```
// c.ts
export default function meow(loudness: number) {}

// d.ts
import meow from './c' // Обратите внимание на отсутствие {фигурных
// скобок}.
meow(11)
```

Он также поддерживает импортирование из модуля с помощью подстановочного импорта (*):

```
// e.ts
import * as a from './a' a.foo()
a.bar()
```

А также повторный экспорт некоторых (или всех) экспортов из модуля:

```
// f.ts
export * from './a'
export {result} from './b'
export meow from './c'
```

Поскольку мы пишем TypeScript, а не JavaScript, то можем экспортировать типы и интерфейсы так же, как и значения. Типы и значения существуют в раздельных пространствах имен, поэтому приемлемо экспортировать два элемента с одинаковым именем — один на уровне значений, а второй на уровне типов. TypeScript сам выведет тип или значение при использовании элемента:

```
// g.ts
export let X = 3
export type X = {y: string}

// h.ts
import {X} from './g'

let a = X + 1          // X относится к значению X
let b: X = {y: 'z'}    // X относится к типу X
```

Пути модулей — это имена файлов в файловой системе. На основе их местоположения загрузчики модулей преобразовывают имена в файлы.

## Динамический импорт

По мере роста приложения времени для его начального представления будет требоваться все больше. Это особенно проблематично для фронтенд-приложений, где сеть может оказаться узким местом, но также касается и бэкенд-приложений, которым требуется больше времени для старта, поэтому бóльшая часть его кода импортируется на высшем уровне — такой код должен быть загружен из файловой системы, разрешен, скомпилирован и вычислен, и во время всего этого процесса невозможен запуск другого кода.

В случае с фронтендом один из способов решить эту проблему (кроме написания меньшего количества кода) — это разделить код на множе-

ство сгенерированных файлов JavaScript. Так вы получите выгоду от параллельной загрузки частей, что облегчит тариф на большие сетевые запросы (рис. 10.1).

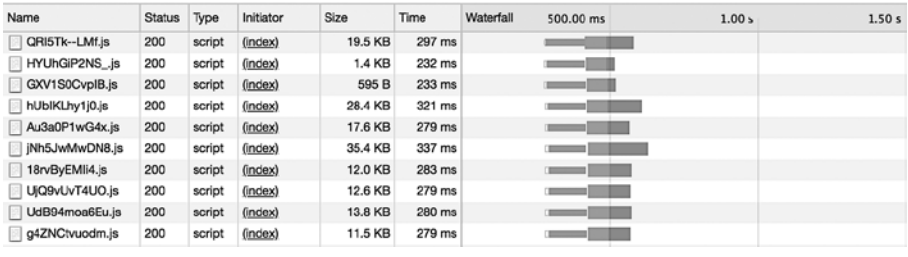


**Рис. 10.1.** Сетевой каскад для JavaScript, загруженного с facebook.com

Дальнейшая оптимизация заключается в загрузке частей кода по требованию. Крупные фронтенд-приложения, вроде находящихся на Facebook или Google, используют этот вид оптимизации как сам собой разумеющийся. Без него клиенты могут столкнуться с загрузкой гигабайтов JavaScript-кода для открытия начальной страницы, что может занять минуты и даже часы (и люди, скорее всего, перестанут использовать эти сервисы, как только получат счета за мобильную связь).

Загрузка по требованию также полезна и по другим причинам. Например, популярная библиотека для управления датами Moment.js (https://momentjs.com/) содержит пакеты для поддержки разных форматов дат, используемых по всему миру. Каждый пакет соответствует определенной местности и весит около 3 Кб. Загрузка всех местностей для каждого пользователя может оказаться ударом по производительности и пропускной способности, поэтому лучше определить местонахождение пользователя, а затем загрузить соответствующий пакет дат.

LABjs и ее потомки представили концепцию загрузки кода по требованию, когда он действительно нужен. Этот принцип был формально обозначен как динамический импорт. Выглядит же он так:

```
let locale = await import('locale_us-en')
```

Вы можете использовать `import` либо как инструкцию для статического внедрения кода (как мы и делали до этого момента), либо как функцию, которая возвращает для модуля `Promise` (как мы сделали в текущем примере).

Можете передать произвольное выражение, которое вычисляется как строка в `import`, но тогда вы утратите безопасность типов. Для безопасного использования динамического импорта выполните одно из следующих действий:

1. Передайте строчный литерал непосредственно в `import`, не присваивая ему перед этим переменную.
2. Передайте в `import` выражение и самостоятельно аннотируйте сигнатуру модуля.

При использовании второго варианта стандартным решением будет статически импортировать модуль, но использовать его только в позиции типа, чтобы TypeScript компилировал статический импорт (см. подраздел «Директива типов» на с. 323). Например:

```
import {locale} from './locales/locale-us'

async function main() {
    let userLocale = await getUserLocale()
    let path = ./locales/locale-${userLocale}
    let localeUS: typeof locale = await import(path)
}
```

Мы импортировали `locale` из ./locales/locale-us, но использовали его только для его типа, который мы извлекли с помощью `typeof locale`. TypeScript не мог статически найти тип `import(path)`, поскольку `path` — это вычисляемая переменная, а не статическая строка. Поскольку мы не использовали `locale` в качестве переменной, TypeScript скомпилировал статический импорт (в этом примере TypeScript вообще не генерирует экспорт верхнего уровня), обеспечив не только безопасность типов, но и динамически вычисленный импорт.

**TSC-УСТАНОВКА: MODULE**

TypeScript поддерживает динамический импорт только в режиме модуля `esnext`. Чтобы использовать динамический импорт, установите `{"module": "esnext"}` в пункте `compilerOptions` файла tsconfig.json. За дополнительной информацией обратитесь к разделам «Запуск TypeScript на сервере», с. 317, и «Запуск TypeScript в браузере», с. 318.

### Использование кода CommonJS и AMD

При потреблении JavaScript-модуля, использующего стандарт CommonJS или AMD, вы можете просто импортировать из него имена, так же как и для модулей ES2015:

```
import {something} from './a/legacy/commonjs/module'
```

По умолчанию предустановленный экспорт в CommonJS не взаимодействует с предустановленным импортом ES2015. Чтобы использовать этот экспорт, потребуется применить подстановочный импорт:

```
import * as fs from 'fs'
fs.readFile('some/file.txt')
```

Чтобы взаимодействие было более гладким, установите `{"esModuleInterop": true}` в `compilerOptions` вашего tsconfig.json. Теперь можете обойтись без подстановки:

```
import fs from 'fs'
fs.readFile('some/file.txt')
```

Как я уже говорил в начале главы, даже несмотря на то, что этот код компилируется, это еще не значит, что он будет работать в среде выполнения. Какой бы стандарт модулей вы ни использовали — `import` и `export`, CommonJS, AMD, UMD или глобальные объекты браузера, — ваш бандлер модулей и их загрузчик должны знать об этом формате, чтобы иметь возможность корректно пакетировать и разделять код во время компиляции, а также исправно загружать его в среде выполнения (см. главу 12).

### Режим модулей против режима скриптов

TypeScript разрешает файлы в одном из двух режимов: *в режиме модулей* или *в режиме скриптов*. Он решает, какой режим использовать, на основе простого эвристического алгоритма: если в файле есть `import` и `export`, он применяет режим модулей, в противном случае использует режим скриптов.

Пока мы применяли режим модулей, и в целом он более востребован. В нем можно использовать `import` и `import()`, чтобы запрашивать код из

других файлов, и `export` — чтобы делать код доступным для других файлов. Если вы используете любые сторонние UMD-модули (напомню, что они применяют CommonJS, RequireJS или глобальные объекты браузера, в зависимости от того, что поддерживает среда), то первым делом необходимо их импортировать (`import`), но нельзя напрямую использовать их глобальный экспорт.

В режиме скриптов любые объявляемые вами переменные верхнего уровня будут доступны для других файлов вашего проекта без необходимости их явного импорта, и вы сможете безопасно принимать глобальный экспорт от сторонних UMD-модулей без их первоначального явного импорта. Вот пара случаев применения режима скриптов:

- Для быстрого создания прототипа кода браузера, который вы собираетесь компилировать в систему без модулей (`{'module': 'none'}` в tsconfig.json) и включать в виде сырых тегов `<script />` в ваш HTML-файл.
- Для создания деклараций типов (см. раздел «Декларации типов» на с. 283).

Практически всегда вы будете придерживаться режима модулей, который TypeScript будет выбирать автоматически при написании вами реального кода и который импортирует другой код и *экспортирует* собственные компоненты для использования другими файлами.

## Пространства имен

TypeScript дает нам еще один инструмент для инкапсуляции кода — ключевое слово `namespace`. Пространства имен покажутся знакомыми для программистов Java, C#, C++, PHP и Python.

Если вы ранее использовали язык, имеющий пространства имен, то заметьте, что в TypeScript они не являются предпочтительным способом инкапсуляции кода. Если есть возможность, используйте вместо них модули.

Пространства имен абстрагируются от скучных рутинных подробностей того, как файлы расположены в файловой системе. Благодаря этому вам не нужно знать, что функция `.mine` находится в каталоге схемы/мошенничество/биткоин/приложения, вместо этого вы можете обратиться к ней с помощью

короткого и удобного пространства имен вроде Схемы.Мошенничество.Биткоин.Приложения.мое[1].

Предположим, у нас есть два файла — модуль для создания запросов HTTP GET и потребитель, который использует этот модуль для создания запросов:

```
// Get.ts
namespace Network {
    export function get<T>(url: string): Promise<T> {
        // ...
    }
}

// App.ts
namespace App {
    Network.get<GitRepo>('https://api.github.com/repos/Microsoft/
    typescript')
}
```

Пространство имен должно само иметь имя (вроде Network) и экспортировать функции, переменные, типы, интерфейсы или другие пространства имен. Любой код в блоке `namespace`, не экспортированный явно, является приватным для этого блока. Пространства имен могут экспортировать другие пространства имен, поэтому можно легко моделировать их вложенные варианты. Например, наш модуль Network становится большим и мы хотим разделить его на несколько субмодулей. Для этого можно использовать пространства имен:

```
namespace Network {
    export namespace HTTP {
        export function get <T>(url: string): Promise<T> {
            // ...
        }
    }
    export namespace TCP {
        listenOn(port: number): Connection {
            //...
        }
```

[1] Я надеюсь, что эта шутка не потеряет актуальности, и не жалею, что не вложился в Биткоин.

```
        // ...
    }
    export namespace UDP {
        // ...
    }
    export namespace IP {
        // ...
    }
}
```

Так все связанные с сетью утилиты помещаются в пространства имен под Network. Теперь мы можем, например, вызвать Network.HTTP.get и Network.TCP.listenOn из любого файла. Подобно интерфейсам, пространства имен могут быть дополнены, что делает удобным их разделение между файлами. TypeScript будет сам рекурсивно производить слияние пространств имен с идентичными именами:

```
// HTTP.ts
namespace Network {
    export namespace HTTP {
        export function get<T>(url: string): Promise<T> {
            // ...
        }
    }
}

// UDP.ts
namespace Network {
    export namespace UDP {
        export function send(url: string, packets: Buffer): Promise<void> {
            // ...
        }
    }
}

// MyApp.ts
Network.HTTP.get<Dog[]>('http://url.com/dogs')
Network.UDP.send('http://url.com/cats', new Buffer(123))
```

Если вы столкнетесь с длинными иерархиями пространств имен, то можете использовать псевдонимы, чтобы сократить их. Обратите внимание, что, несмотря на схожий синтаксис, деструктурирование (по-

добное производимому при импорте модулей ES2015) для псевдонимов не поддерживается:

```
// A.ts
namespace A {
    export namespace B {
        export namespace C {
            export let d = 3
        }
    }
}

// MyApp.ts
import d = A.B.C.d

let e = d * 3
```

## Коллизии

Коллизии между экспортами с одинаковыми именами не допускаются:

```
// HTTP.ts
namespace Network {
    export function request<T>(url: string): T {
        // ...
    }
}

// HTTP2.ts
namespace Network {
    // Ошибка TS2393: повторяющаяся реализация функции.
    export function request<T>(url: string): T {
        // ...
    }
}
```

Исключением из этого правила являются внешние перегруженные декларации функций, используемые для уточнения типов функций:

```
// HTTP.ts
namespace Network {
    export function request<T>(url: string): T
```

```
}

// HTTP2.ts
namespace Network {
    export function request<T>(url: string, priority: number): T
}

// HTTPS.ts
namespace Network {
    export function request<T>(url: string, algo: 'SHA1' | 'SHA256'): T
}
```

### Скомпилированный вывод

В отличие от импорта и экспорта пространства имен не учитывают ваши установки `module` в tsconfig.json и всегда компилируются в глобальные переменные. Рассмотрим скомпилированный вывод на примере следующего модуля:

```
// Flowers.ts
namespace Flowers {
    export function give(count: number) {
        return count + ' flowers'
    }
}
```

После прохождения TSC скомпилированный вывод выглядит так:

```
let Flowers
(function (Flowers) { ❶
    function give(count) {
        return count + ' flowers'
    }
    Flowers.give = give ❷
})(Flowers || (Flowers = {})) ❸
```

❶ `Flowers` объявлен внутри сразу вызываемого функционального выражения — самовызывающейся функции, — чтобы создать замыкание и предотвратить утечку не экспортированных явно переменных из модуля `Flowers`.

❷ TypeScript присваивает функцию `give`, которую мы экспортировали, к пространству имен `Flowers`.

❸ Если пространство имен `Flowers` уже определено глобально, то TypeScript расширяет его (`Flowers`). В противном случае он создает и расширяет только что созданное пространство имен (`Flowers = {}`).

**ЛУЧШЕ ИСПОЛЬЗОВАТЬ МОДУЛИ, А НЕ ПРОСТРАНСТВА ИМЕН**

Предпочитайте регулярные модули (вида `import` и `export`) в качестве способа следования стандартам JavaScript и делайте зависимости более явными.

Явные зависимости имеют множество преимуществ в отношении читаемости, обеспечивая изоляцию модуля (модули, в отличие от пространств имен, не допускают автоматического слияния) и статический анализ, что важно для фронтенд-проектов, где отделение мертвого кода и разделение скомпилированного на несколько файлов может сильно повлиять на производительность.

При запуске программ TypeScript в среде NodeJS модули также предпочтительнее из-за встроенной поддержки CommonJS. В браузерных средах некоторые программисты выбирают пространства имен из-за их простоты, но для проектов среднего и большого размера лучше придерживаться модулей.

## Слияние деклараций

Мы затронули три типа слияния, которые TypeScript делает за нас.

- ❑ Слияние значений и типов, из-за которого одно и то же имя может относиться либо к значению, либо к типу в зависимости от использования (см. подраздел «Паттерн объект-компаньон» на с. 176).
- ❑ Слияние нескольких пространств имен в одно.
- ❑ Слияние нескольких интерфейсов в один.

Вы могли догадаться, что это всего три отдельных случая из гораздо более обширных функциональных возможностей TypeScript, слияние имен в котором раскрывает множество паттернов, которые сложно выразить иными способами (табл. 10.1).

**ФЛАГ MODULERESOLUTION**

Особо внимательные читатели могли заметить флаг `moduleResolution`, доступный в tsconfig.json. Этот флаг управляет алгоритмом, используемым TypeScript для разрешения имен модулей в приложении, и поддерживает два режима:

- `node`. Всегда используйте этот режим. Он разрешает модули, применяя тот же алгоритм, что и NodeJS. Модули с префиксами `.` , `/` или `~` (вроде `./my/file`) разрешаются из локальной файловой системы либо относительно текущего файла, либо посредством полного пути (относительно вашего каталога или каталога, установленного в `baseUrl` в файле tsconfig.json) в зависимости от используемого префикса. Пути модулей, не имеющих префикса, загружаются из каталога `node_modules`, так же как и в NodeJS. TypeScript использует стратегию разрешения, свойственную NodeJS, двумя способами:

  1. Помимо поля `main` в пакетном package.json, куда NodeJS смотрит, чтобы найти в каталоге файл, импортируемый по умолчанию, TypeScript также смотрит на свойство `types` (см. раздел «Поиск типов для JavaScript», с. 298).

  2. При импортировании файла с неопределенным расширением TypeScript сначала ищет файл с таким именем и расширением .ts, затем .tsx, .d.ts и в завершение — .js.

- `classic`. Этот режим не рекомендуется использовать совсем. В нем относительные пути разрешаются в режиме node, а для имен без префиксов TypeScript будет искать файл с заданным именем в текущем каталоге, а затем подниматься по одному каталогу вверх по дереву, пока не найдет совпадающий. Для тех, кто пришел из мира NodeJS или JavaScript, такое поведение покажется удивительным, причем оно плохо взаимодействует с другими инструментами разработки.

То есть если, к примеру, вы объявите значение и псевдоним типа в одной области, TypeScript это позволит и выведет то, что вы подразумевали, — тип или значение, исходя из того, используете вы имя в позиции значения или типа. Это как раз и позволяет нам реализовать паттерн, описанный в подразделе «Паттерн объект-компаньон» на с. 176. Это также означает, что вы можете использовать интерфейс и пространство имен для реализации объектов-компаньонов и не ограничены только значениями и псевдонимами типов.Или же вы можете воспользоваться преимуществами

слияния модулей для расширения деклараций сторонних модулей (см. раздел «Расширение модуля» на с. 337). Еще вы можете добавлять статические методы к перечислениям посредством слияния этих перечислений с пространствами имен (обязательно попробуйте).

**Таблица 10.1.** Возможно ли слияние декларации

| | | В | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Знч. | Класс | Перечисл. | Функция | Псвд. типа | Интерфейс | Пр-во имен | Модуль |
| Из | Знч. | Нет | Нет | Нет | Нет | Да | Да | Нет | — |
| | Класс | — | Нет | Нет | Нет | Нет | Да | Да | — |
| | Перечисл. | — | — | Да | Нет | Нет | Нет | Да | — |
| | Функция | — | — | — | Нет | Да | Да | Да | — |
| | Псвд. типа | — | — | — | — | Нет | Нет | Да | — |
| | Интерфейс | — | — | — | — | — | Да | Да | — |
| | Пр-во имен | — | — | — | — | — | — | Да | — |
| | Модуль | — | — | — | — | — | — | — | Да |

# Итоги

В этой главе вы изучили систему модулей TypeScript, начав с краткой истории системы модулей JavaScript, модулей ES2015, безопасной загрузки кода по требованию посредством динамического импорта, взаимодействия с модулями CommonJS и AMD, а затем сравнили систему модулей с системой скриптов. Также вы познакомились с темой пространств имен, их слияния и рассмотрели принцип работы слияния деклараций в TypeScript.

По мере разработки приложений старайтесь придерживаться модулей ES2015. TypeScript неважно, какую систему модулей использовать, но ES2015 облегчает интеграцию с инструментами сборки (см. главу 12).

## Упражнение к главе 10

Поработайте над слиянием деклараций, чтобы:

а) повторно реализовать паттерн объект-компаньон (см. подраздел «Паттерн объект-компаньон» на с. 176), используя пространства имен и интерфейсы вместо значений и типов;

б) добавить статические методы в перечисление.

ГЛАВА 11

# Взаимодействие с JavaScript

Мир несовершенен. Кофе бывает чересчур горячим, родители со своими голосовыми — слишком назойливыми, яма на дороге все еще на месте, несмотря на кучу жалоб в дорожную службу, а код может не до конца быть заполнен статическими типами.

Чаще вы будете начинать проект в TypeScript не с нуля — скорее он станет маленьким островком безопасности, погруженным в крупную менее безопасную кодовую базу. Возможно, у вас есть хорошо изолированный компонент, в котором вы хотите попробовать TypeScript, даже несмотря на то, что ваша компания везде использует обычный ES6 JavaScript, а может быть, вам уже надоело получать сообщения в шесть утра, из-за того что вы отрефакторили код и забыли обновить точку вызова (сейчас семь утра, и вы тайком производите слияние TSC с кодовой базой, пока ваши коллеги не проснулись). Так или иначе, вы наверняка начнете с островка TypeScript в бестипном море.

До сих пор я училл вас, как писать код TypeScript правильно. Эта же глава посвящена написанию его практичным способом, а именно в реальных базах кода, находящихся в процессе миграции из нетипизированных языков, использующих сторонние библиотеки JavaScript, которые иногда жертвуют безопасностью типов ради быстрого патча, чтобы исправить продакшен. Эта глава посвящена работе с JavaScript, и в ней мы изучим:

- Использование деклараций типов.
- Постепенную миграцию из JavaScript в TypeScript.
- Использование стороннего кода JavaScript и TypeScript.

## Декларации типов

*Декларация типа* — это файл с расширением `.d.ts`. Наряду с аннотациями JSDoc (см. раздел «Шаг 2б: добавление аннотаций JSDoc (по желанию)» на с. 295) это способ прикрепить типы TypeScript к коду JavaScript, который в противном случае останется нетипизированным.

Синтаксис деклараций типов похож на обычный TypeScript, но в некоторых моментах отличен от него.

- Он может содержать только типы, но не значения, к которым относятся реализации функций, классов, объектов и переменных и значения по умолчанию для параметров.
- Декларации типов не могут определять значения, но могут видеть, что существует значение, определенное где-то в коде JavaScript. Для этого используется специальное ключевое слово `declare`.
- Декларации типов объявляют типы только для элементов, которые видимы потребителям. Мы не включаем в синтаксис деклараций такие элементы, как типы, которые не экспортируются, или типы для локальных переменных внутри тел функций.

Рассмотрим часть TypeScript кода (.ts) и эквивалентную ей декларацию типа (d.ts). Этот пример представляет честно позаимствованный кусок кода из популярной библиотеки RxJS. Можете не углубляться в детали производимых действий, но обратите внимание на то, какие возможности языка использованы (импорт, классы, интерфейсы, поля классов, перегрузки функций и т. д.):

```
import {Subscriber} from './Subscriber'
import {Subscription} from './Subscription'
import {PartialObserver, Subscribable, TeardownLogic} from './types'

export class Observable<T> implements Subscribable<T> {
    public _isScalar: boolean = false
    constructor(
        subscribe?: (
            this: Observable<T>,
            subscriber:
            Subscriber<T>
        ) => TeardownLogic
    ) {
        if (subscribe) {
            this._subscribe = subscribe
        }
    }
    static create<T>(subscribe?: (subscriber: Subscriber<T>) =>
    TeardownLogic) {
```

```
        return new Observable<T>(subscribe)
    }
    subscribe(observer?: PartialObserver<T>): Subscription
    subscribe(
        next?: (value: T) => void,
        error?: (error: any) => void,
        complete?: () => void
    ): Subscription
    subscribe(
        observerOrNext?: PartialObserver<T> | ((value: T) => void),
        error?: (error: any) => void,
        complete?: () => void
    ): Subscription {
        // ...
    }
}
```

Пропуск этого кода через TSC при включенном флаге `declarations` (`tsc -d Observable.ts`) производит следующие декларации типов Observable.d.ts:

```
import {Subscriber} from './Subscriber'
import {Subscription} from './Subscription'
import {PartialObserver, Subscribable, TeardownLogic} from './types'

export declare class Observable<T> implements Subscribable<T> { ❶
    _isScalar: boolean
    constructor(
        subscribe?: (
            this: Observable<T>,
            subscriber: Subscriber<T>
        ) => TeardownLogic
    );
    static create<T>(
        subscribe?: (subscriber: Subscriber<T>) => TeardownLogic
    ): Observable<T>
    subscribe(observer?: PartialObserver<T>): Subscription
    subscribe(
        next?: (value: T) => void,
        error?: (error: any) => void,
        complete?: () => void
    ): Subscription ❷
}
```

❶ Обратите внимание на ключевое слово `declare`, расположенное перед `class`. На самом деле мы не можем определять класс в декларации типа, но можем объявить, что определили класс в файле JavaScript, соотнесенном с файлом .d.ts. Рассматривайте `declare` как утверждение: «Я обещаю, что мой код JavaScript экспортирует класс этого типа».

❷ Декларации типов не содержат реализаций, поэтому мы храним только две перегрузки для `subscribe`, но не сигнатуру ее реализации.

Заметьте, что Observable.d.ts — это тот же Observable.ts, но без реализации. Другими словами, это просто типы из Observable.ts.

Эта декларация типа не пригодится для других файлов библиотеки RxJS, которые используют Observable.ts, поскольку они имеют доступ к исходному TypeScript-файлу Observable.ts и могут применять его напрямую. Тем не менее она приходится кстати, если вы используете RxJS в TypeScript-приложении.

Подумайте, если бы авторы RxJS хотели создать пакет с информацией типов на NMP для своих TypeScript-пользователей (RxJS может быть использована как в JavaScript-, так и в TypeScript-приложениях), что бы они выбрали: упаковать исходные файлы TypeScript (для пользователей TypeScript) и скомпилированные файлы JavaScript (для пользователей JavaScript) или предоставить скомпилированные файлы JavaScript совместно с декларациями типов для пользователей TypeScript? Последний вариант предполагает меньший размер файла и точно определяет, какой импорт подойдет. Он также помогает сократить время на компиляцию приложения, поскольку TSC не должен перекомпилировать RxJS при каждой компиляции приложения (благодаря этому работает стратегия оптимизации, которую я представляю в подразделе «Проектные ссылки» на с. 314).

Файлы деклараций типов применяются в нескольких вариантах:

1. Когда кто-то другой тоже использует ваш скомпилированный код TypeScript из своего TypeScript-приложения, его TSC будет искать файлы .d.ts, соответствующие вашим сгенерированным файлам JavaScript. Это сообщит TypeScript, какие типы относятся к вашему проекту.
2. Редакторы кода с поддержкой TypeScript (вроде VSCode) будут считывать файлы .d.ts, чтобы предоставить пользователям полез-

ные подсказки в процессе типизации, даже если они не используют TypeScript.

3. Для ускорения процесса компиляции, поскольку они избегают необязательных перекомпиляций кода TypeScript.

Декларации типов — это способ сообщить TypeScript, что «там-то существует то-то, определенное в JavaScript-коде, и я сейчас это опишу». Если мы говорим о декларациях типов, то часто называем их *сторонними*, чтобы отличать от регулярных деклараций, содержащих значения. Например, *сторонняя декларация переменной* использует ключевое слово `declare`, чтобы объявить, что переменная определена где-то в коде JavaScript, в то время как регулярная декларация — это нормальная декларация `let` или `const`, которая объявляет переменную без ключевого слова `declare`.

Используйте декларации типов, чтобы:

- Сообщить TypeScript о глобальной переменной, которая определена где-то в коде JavaScript. Например, если вы использовали полифилы для глобального `Promise` или определили process.env в среде браузера, то можете использовать *стороннюю декларацию переменной*, чтобы держать TypeScript в курсе развития событий.
- Определить тип, доступный глобально по всему проекту, чтобы для его использования вам не приходилось сначала его импортировать (сторонняя декларация типа).
- Сообщить TypeScript о стороннем модуле, установленном с помощью NPM (*сторонняя декларация модуля*).

Независимо от цели использования декларация типа должна существовать в файле .ts или .d.ts с режимом скриптов (см. подраздел «Режим модулей против режима скриптов» на с. 273). Согласно принятым обозначениям мы задаем файлу расширение .d.ts, если он имеет соответствующий файл .js. В противном случае мы используем расширение .ts. При этом неважно, как вы назовете файл. Например, мне нравится придерживаться одного файла type.ts верхнего уровня, пока он не становится слишком громоздким, хотя один файл деклараций типов может содержать любое количество деклараций.

В завершение отмечу, что в то время, как значения верхнего уровня в файле деклараций типов нуждаются в ключевом слове `declare` (`declare let`, `declare function`, `declare class` и т. д.), для типов и интерфейсов верхнего уровня оно не требуется.

Прояснив эти основные правила, рассмотрим некоторые примеры каждого вида деклараций типов.

### Внешние декларации переменных

Внешние декларации переменных — это способ сообщить TypeScript о глобальной переменной, которая может быть использована в любом файле .ts или .d.ts проекта без необходимости ее явного импорта.

Предположим, вы запускаете программу NodeJS в браузере, которая в определенный момент проверяет process.env.NODE_ENV (он либо `"development"`, либо `"production"`). При запуске программы вы получаете неприятную ошибку среды выполнения:

```
Неперехваченная ReferenceError: процесс не определен.
```

Вы обращаетесь за помощью к Stack Overflow и выясняете, что скорейший метод запустить программу — это самостоятельно ввести полифилы для process.env.NODE_ENV и захардкодить его. Итак, вы создаете новый файл, polyfill.ts, и глобально определяете process.env:

```
process = {
     env: {
        NODE_ENV: 'production'
    }
}
```

Конечно, на помощь приходит TypeScript, подчеркивая то, что нужно, красной волнистой, чтобы уберечь вас от ошибки, которую вы, очевидно, совершаете, расширяя глобальный объект `window`:

```
Ошибка TS2304: невозможно найти имя 'process'.
```

Но в этом случае TypeScript излишне предусмотрителен. Вы действительно хотите расширить `window` и делаете это безопасно.

Как быть? Вы открываете polyfills.ts в Vim (видите, к чему все идет) и типизируете:

```
declare let process: {
    env: {
        NODE_ENV: 'development' | 'production'
    }
```

```
}

process =
    { env: {
        NODE_ENV: 'production'
    }
}
```

Вы объявляете для TypeScript, что существует глобальный объект `process`, у которого есть свойство `env`, имеющее свойство `NODE_ENV`. Как только вы сообщили об этом, красная волнистая исчезает и вы можете безопасно определять ваш глобальный объект `process`.

**TSC-УСТАНОВКА: LIB**

В TypeScript есть изначальный набор деклараций типов для описания стандартной библиотеки JavaScript, который включает встроенные типы JavaScript вроде `Array` и `Promise`, а также методы для встроенных типов вроде `.toUpperCase`. Помимо этого, он содержит глобальные объекты вроде `window`, `document` (в среде браузера) и `onmessage` (в среде веб-работника).

Вы можете извлечь встроенные декларации типов TypeScript с помощью поля `lib` файла tsconfig.json (см. подраздел «Библиотеки» на с. 312).

## Внешние декларации типов

Внешние декларации типов следуют тем же правилам, что и внешние декларации переменных: они должны существовать в файле .ts или .d.ts с режимом скриптов и быть доступными глобально для других файлов проекта без явного импорта. Например, объявим глобальный тип общего назначения `ToArray<T>`, который поднимает `T` до массива, если он еще не является таковым. Мы можем определить этот тип в любом файле проекта с режимом скриптов. В данном примере определим его в файле верхнего уровня types.ts:

```
type ToArray<T> = T extends unknown[] ? T : T[]
```

Теперь можно использовать этот тип из любого файла проекта без его явного импорта.

```
function toArray<T>(a: T): ToArray<T> {
    // ...
}
```

Рассмотрите использование сторонних деклараций типов для моделирования типов данных, которые применяются по всему приложению. Например, можно использовать их, чтобы сделать глобально доступным тип `UserID`, разработанный в разделе «Имитация номинальных типов» на с. 191.

```
type UserID = string & {readonly brand: unique symbol}
```

Теперь `UserID` доступен для использования в любой части приложения без необходимости его явного импорта.

## Внешние декларации модулей

Когда вы потребляете модуль JavaScript и хотите быстро объявить в нем типы для безопасного использования, но при этом не отправлять декларации типов сначала обратно в репозиторий JavaScript-модулей на GitHub, выходом будет использование внешних деклараций модулей.

Внешняя декларация модулей — это регулярная декларация модулей, окруженная особым синтаксисом `declare module`:

```
declare module 'module-name' {
    export type MyType = number
    export type MyDefaultType = {a: string}
    export let myExport: MyType
    let myDefaultExport: MyDefaultType
    export default myDefaultExport
}
```

Имя модуля (в текущем примере `'module-name'`) точно соответствует пути `import`. Когда вы импортируете этот путь, ваша внешняя декларация модуля сообщит TypeScript, что именно доступно:

```
import ModuleName from 'module-name'
ModuleName.a // строка
```

Если у вас есть вложенный модуль, то обязательно включите весь путь import в его декларацию:

```
declare module '@most/core' {
    // Декларация типа
}
```

Если вы просто хотите быстро сообщить TypeScript: «Я импортирую этот модуль, а типизирую его позже, пока предположи, что это any», тогда оставьте заголовок, но опустите саму декларацию:

```
// Объявление модуля, который может быть импортирован и у которого
// все импорты имеют  тип any.
declare module 'unsafe-module-name'
```

Теперь потребление такого модуля будет менее безопасным:

```
import {x} from 'unsafe-module-name'
x // any
```

Декларации модулей поддерживают подстановочные импорты, и можно задать тип любому пути import, который соответствует заданному паттерну. Для этого используйте подстановочный знак (*)[1]:

```
// Файлы типа JSON, импортированные с помощью Webpack загрузчика json.
declare module 'json!*' {
    let value: object
    export default value
}

// Файлы типа CSS, импортированные с помощью Webpack загрузчика стиля.
declare module '*.css' {
    let css: CSSRuleList
    export default css
}
```

Теперь можно загружать JSON и CSS файлы:

```
import a from 'json!myFile'
a // объект

import b from './widget.css'
b // CSSRuleList
```

Перейдите к подразделу «JavaScript, не имеющий деклараций типов на DefinitelyTyped» на с. 303, чтобы ознакомиться с примером использования внешних деклараций модулей для объявления типов в нетипизированном стороннем коде JavaScript.

---

[1] Сопоставление символа подстановки с * следует тем же правилам, что и обычное сопоставление с образцом по маске (https://ru.wikipedia.org/wiki/Шаблон_поиска)).

Чтобы работали последние два примера, нужно настроить систему сборки для загрузки файлов .json и .css. Вы можете объявить TypeScript, что эти паттерны путей можно импортировать безопасно, но TypeScript не сможет создать их сам.

## Поэтапная миграция из JavaScript в TypeScript

TypeScript был разработан с учетом возможности взаимодействия с JavaScript. Поэтому хоть это и не полностью безболезненно, но все же миграция в TypeScript — это приятный опыт, позволяющий преобразовать кодовую базу по файлу за раз, получить более глубокий уровень безопасности и коммит за коммитом и удивлять босса и коллег, насколько мощным может быть статически типизированный код.

На высоком уровне кодовая база должна быть полностью написана в TypeScript и строго типизирована, а сторонние библиотеки JavaScript, от которых вы зависите, должны быть хорошего качества и иметь собственные строгие типы. Процесс написания кода ускорится вдвое благодаря перехвату ошибок во время компиляции, а также богатой системе автозаполнения TypeScript. Чтобы достичь успешного результата миграции, потребуется совершить несколько небольших шагов:

- Добавить TSC в проект.
- Начать проверку типов имеющегося кода JavaScript.
- Перенести JavaScript-код в TypeScript файл за файлом.
- Установить декларации типов для зависимостей. То есть выделить типы для зависимостей, которые их не имеют, либо прописать декларации типов для нетипизированных зависимостей и отправить их обратно на DefinitelyTyped[1].
- Включить для базы кода режим `strict`.

Этот процесс может занять некоторое время, но вы сразу обнаружите прирост безопасности и производительности, а также откроете и другие преимущества позже. Рассмотрим перечисленные шаги.

[1] DefinitelyTyped — это открытый репозиторий для деклараций типов JavaScript. Читайте далее, чтобы узнать больше.

### Шаг 1: добавление TSC

При работе с базой кода, объединяющей TypeScript и JavaScript, сначала позвольте TSC компилировать JavaScript-файлы вместе с файлами TypeScript в настройках tsconfig.json:

```
{
    "compilerOptions": {
    "allowJs": true
}
```

Одно это изменение уже позволит использовать TSC для компиляции кода JavaScript. Просто добавьте TSC в процесс сборки и либо запустите через него каждый файл JavaScript[1], либо продолжайте запускать устаревшие файлы JavaScript через процесс сборки, а новые файлы TypeScript —через TSC.

При `allowJs`, установленном как `true`, TypeScript не будет проверять типы в текущем коде JavaScript, но будет транспилировать этот код в ES3, ES5 или в версию, которая установлена как `target` в файле tsconfig.json, используя систему модулей, запрошенную вами (в поле `module` файла tsconfug.json). Первый шаг выполнен. Сделайте его коммит и похлопайте себя по плечу — теперь ваша кодовая база использует TypeScript.

### Шаг 2a: активация проверки типов для JavaScript (по желанию)

Теперь, когда TSC обрабатывает код JavaScript, почему бы не проверить его типы? Даже если там нет явных аннотаций типов, вспомните, что TypeScript может вывести типы для JavaScript-кода так же, как и для кода TypeScript. Включите необходимую опцию в tsconfig.json:

```
{
    "compilerOptions": {
    "allowJs": true,
    "checkJs": true
}
```

[1] Для крупных проектов пропуск каждого файла через TSC может занять много времени. О способе увеличения производительности в больших проектах читайте в подразделе «Проектные ссылки» на с. 314.

Теперь, когда бы TypeScript ни компилировал файл JavaScript, он будет стараться вывести типы и произвести их проверку так же, как он делает это для кода TypeScript.

Если ваша база кода велика и при включении `checkJs` обнаруживается слишком много ошибок за раз, выключите ее. Вместо нее включите проверку файлов JavaScript по одному, добавив директиву `// @ts-check` (обычный комментарий в верхней части файла). Либо, если большие файлы выбрасывают кучу ошибок, которые вы пока не хотите исправлять, оставьте включенной `checkJs` и добавьте директиву `// @ts-nocheck` именно для этих файлов.

TypeScript не может вывести типы для всего (например, не выводит типы для параметров функций), поэтому он выведет множество типов в JavaScript-коде как `any`. Если у вас включен режим `strict` в tsconfig.json (рекомендую), то вы можете предпочесть на время миграции разрешить неявные `any`. Добавьте в tsconfig.json следующее:

```
{
    "compilerOptions": {
    "allowJs": true,
    "checkJs": true,
    "noImplicitAny": false
}
```

Не забудьте снова включить `noImplicitAny`, когда завершите миграцию основной части кода в TypeScript. При этом, скорее всего, будет обнаружено множество упущенных ошибок (если только вы не Зенидар — послушник JavaScript-ведьмы Бавморды, который может проверять типы силой мысленного взора с помощью зелья из полыни).

Когда TypeScript выполняет код JavaScript, он использует более мягкий алгоритм вывода, чем в случае с кодом TypeScript. А именно:

- Все параметры функций опциональны.
- Типы свойств функций и классов выводятся на основе их использования (вместо необходимости быть объявленными заранее):

```
class A {
    x = 0 // number | string | string[], вывод на основе использования.
    method() {
        this.x = 'foo'
    }
    otherMethod() {
        this.x = ['array', 'of', 'strings']
    }
}
```

- После объявления объекта, класса или функции вы можете присвоить им дополнительные свойства. За кадром TypeScript делает это посредством генерирования соответствующего пространства имен для каждой декларации функции и автоматического добавления сигнатуры индекса каждому объектному литералу.

### Шаг 2б: добавление аннотаций JSDoc (по желанию)

Возможно, вы спешите и вам просто нужно добавить одну аннотацию типа для новой функции, внесенной в старый файл JavaScript. Для этого можно использовать аннотацию `JSDoc`, пока у вас не появится возможность конвертировать этот файл в TypeScript.

Вы, вероятно, встречали JSDoc ранее. Это такие комментарии в верхней части кода с аннотациями, начинающимися с `@`, вроде `@param`, `@returns` и т. д. TypeScript понимает `JSDoc` и использует его в качестве вводных данных для модуля проверки типов наравне с явными аннотациями типов.

Предположим, у вас есть служебный файл на 3000 строк (да, знаю, его написал ваш «друг»). Вы добавляете в него новую сервисную функцию:

```
export function toPascalCase(word) {
    return word.replace(
        /\w+/g,
        ([a, ...b]) => a.toUpperCase() + b.join('').toLowerCase()
    )
}
```

Без полноценного преобразования `utils.js` в TypeScript, которое наверняка вскроет кучу багов, вы можете аннотировать только функцию `toPascaleCase`, создав маленький островок безопасности в море нетипизированного JavaScript:

```
/**
    * @param word {string} Строка ввода для конвертации.
    * @returns {string} Строка в PascalCase
    */
export function toPascalCase(word) {
    return word.replace(
        /\w+/g,
        ([a, ...b]) => a.toUpperCase() + b.join('').toLowerCase()
    )
}
```

Без этой аннотации JSDoc TypeScript вывел бы тип `toPascaleCase` как `(word: any) => string`. Теперь же при компиляции он будет знать, что тип `toPascaleCase` — это `(word: string) => string`. А вы при этом получите полезное документирование.

Для более подробного ознакомления с аннотациями JSDoc посетите ресурс TypeScript Wiki (https://github.com/Microsoft/TypeScript/wiki/JSDoc-support-in-JavaScript).

### Шаг 3: переименование файлов в .ts

Как только вы добавили TSC в процесс сборки и начали опционально проверять типы и аннотировать код JavaScript везде, где это возможно, настало время переключения на TypeScript.

Файл за файлом обновляйте разрешения файлов с .js (или .coffee, es6 и т. д.) в .ts. Сразу после переименования файлов в редакторе вы увидите появление красных волнистых друзей, указывающих на ошибки типов, пропущенные случаи, забытые проверки на `null`, а также опечатки в именах переменных. Есть два способа убрать их.

1. Сделать все правильно. Выделите время для корректной типизации форм, полей и функций, чтобы перехватывать ошибки во всех файлах, потребляющих их. Если у вас включена опция `checkJs`, включите `noImplicitAny` в tsconfig.json, чтобы обнаружить все `any` и типизировать их, а затем снова выключите, чтобы проверка типов оставшихся файлов JavaScript не высыпала столько же ошибок.
2. Быстро массово переименовать файлы в расширение .ts и оставить настройки tsconfig.json мягкими (установить `strict` как `false`), чтобы

после переименования было выброшено как можно меньше ошибок. Типизируйте сложные типы как `any`, чтобы успокоить модуль проверки типов. Исправьте оставшиеся ошибки и сделайте коммит. Как только с этим покончено, один за другим включите флаги режима `strict` (`noImplicitAny`, `noImplicitThis`, `strictNullChecks` и т. д.), каждый раз исправляя всплывающие ошибки. (В приложении Д приведен полный список флагов.)

Если вы предпочтете короткий маршрут, то определите внешнюю декларацию типа `TODO` в качестве псевдонима типа для `any` и используйте ее вместо `any`, чтобы впоследствии было проще отыскать упущенные типы. Вы можете назвать ее более определенно, чтобы облегчить поиск по проекту:

```
// globals.ts
type TODO_FROM_JS_TO_TS_MIGRATION = any

// MyMigratedUtil.ts
export function mergeWidgets(
    widget1: TODO_FROM_JS_TO_TS_MIGRATION,
    widget2: TODO_FROM_JS_TO_TS_MIGRATION
): number {
    // ...
}
```

Оба подхода вполне актуальны, и уже вам решать, какой предпочтительнее. TypeScript — поэтапно типизируемый язык, который изначально создан для взаимодействия с нетипизированным JavaScript в максимально безопасной форме. Неважно, взаимодействуете ли вы с нетипизированным JavaScript или со слабо типизированным TypeScript, строго типизированный TypeScript всегда будет следить за тем, чтобы взаимодействие происходило максимально безопасно.

### Шаг 4: активация строгости

Как только критическая масса JavaScript-кода будет перенесена, вы захотите сделать его безопасным по всем параметрам, поочередно задействовав более строгие флаги TSC (полный список флагов — в приложении Д).

По окончании вы можете отключить TSC-флаги, отвечающие за взаимодействие с JavaScript, подтверждая, что весь ваш код написан в строго типизированном TypeScript:

```
{
    "compilerOptions": {
    "allowJs": false,
    "checkJs": false
}
```

Это вскроет все оставшиеся ошибки типов. Исправьте их и получите безупречную безопасную базу кода, за которую большинство суровых инженеров OCaml похлопали бы вас по плечу.

Следование этим шагам поможет вам далеко продвинуться при добавлении типов в контролируемый вами код JavaScript. Но что насчет кодов, которые контролируете не вы? Вроде тех, что устанавливаются с NPM. Но прежде, чем мы изучим этот вопрос, давайте немного отвлечемся...

## Поиск типов для JavaScript

Когда вы импортируете файл JavaScript из TypeScript-файла, TypeScript производит для него поиск деклараций типов с помощью следующего алгоритма (вспомните, что в TypeScript понятия «файл» и «модуль» взаимозаменяемы)[1]:

1. Ищет потомка файла .d.ts с тем же именем, что и у файла .js. Найденный потомок используется в качестве декларации типов для этого файла .js.

   Например, при такой структуре каталога:

   ```
   my-app/
   ├──src/
   │ ├──index.ts
   │ └──legacy/
   │  ├──old-file.js
   │  └──old-file.d.ts
   ```

[1] Строго говоря, это верно для режима модулей, но не для режима скриптов (подраздел «Режим модулей против режима скриптов» на с. 273).

импортируется old-file (старый файл) из index.ts:

```
// index.ts
import './legacy/old-file'
```

TypeScript использует src/legacy/old-file.d.ts в качестве источника деклараций типов для ./legacy/old-file.

2. В противном случае, если `allowJs` и `checkJs` установлены как `true`, будет сделан вывод типов файла .js (на основе представленных в нем аннотаций `JSDoc`) или весь модуль будет обозначен как `any`.

### TSC-УСТАНОВКИ: ТИПЫ И TYPEROOTS (ИСТОЧНИКИ ТИПОВ)

По умолчанию TypeScript ищет сторонние декларации типов в node modules/@types каталога проекта, а также в его подкаталогах (../node modules/@types и т. д.). В большинстве случаев стоит оставлять такое его поведение без изменений.

Если же понадобится его изменить для глобальных деклараций типов, укажите в пункте typeRoots в tsconfig.json массив каталогов, в которых нужно искать декларации типов. Например, укажите TypeScript искать их в каталоге typings наряду с node modules/@types:

```
{
    "compilerOptions": {
        "typeRoots" : ["./typings", "./node modules/@types"]
    }
}
```

Для еще более точечного контроля используйте опцию `types` в tsconfig.json, чтобы определить, для каких пакетов TypeScript должен искать типы. Например, следующая настройка игнорирует сторонние декларации типов, за исключением необходимых для `React`:

```
{
    "compilerOptions": {
        "types" : ["react"]
    }
}
```

При импортировании стороннего модуля JavaScript (пакета NPM, который вы установили в `node modules`) TypeScript использует несколько иной алгоритм:

1. Ищет для модуля локальную декларацию типа и, если таковая существует, использует ее.

   Например, ваша структура каталога выглядит так:

   ```
   my-app/
   ├──node_modules/
   │  └──foo/
   ├──src/
   │  ├──index.ts
   │  └──types.d.ts
   ```

   А так выглядит type.d.ts:

   ```
   // types.d.ts
   declare module 'foo' {
       let bar: {}
       export default bar
   }
   ```

   Если затем вы импортируете `foo`, то в качестве источника типов для него TypeScript использует внешнюю декларацию модуля в types.d.ts:

   ```
   // index.ts
   import bar from 'foo'
   ```

2. В противном случае он будет искать декларацию в файле package.json, принадлежащем модулю. Если в нем определено поле `types` или `typings`, то он использует файл .d.ts, на который это поле указывает, и возьмет декларации типов из него.

3. Или он будет поочередно просматривать каталоги в поиске каталога `node modules/@types`, где содержатся декларации типов для модуля.

   Например, вы установили `React`:

   ```
   npm install react --save
   npm install @types/react --save-dev

   my-app/
   ├──node_modules/
   ```

```
│ ├──@types/
│ │ └──react/
│ └──react/
├──src/
│ └──index.ts
```

При импорте `React` TypeScript найдет каталог `@types/react` и использует его в качестве источника деклараций типов для него:

```
// index.ts
import * as React from 'react'
```

4. В противном случае он перейдет к шагам 1–3 алгоритма локального поиска типов.

Я перечислил немало шагов, но вы к ним привыкнете.

# Использование стороннего кода JavaScript

Предположу, что для установки стороннего кода JavaScript вы используете менеджер пакетов вроде NPM или Yarn. Если же вы относитесь к тем уникумам, которые копируют и вставляют код вручную, то вам должно быть стыдно.

Когда вы размещаете (`npm install`) в проект сторонний код JavaScript, то возможны три варианта развития событий.

1. Установленный код уже содержит декларации типов.
2. Код не содержит декларации типов, но их можно взять из DefinitelyTyped.
3. Код не содержит декларации типов, и их нет на DefiniteTyped.

Рассмотрим каждый случай.

## JavaScript с декларациями типов

Вы узнаете, содержит ли пакет декларации типов, если выполните его `import` с `{"noImplicitAny": true}` и TypeScript не выбросит вам красную волнистую.

Если код, который вы устанавливаете, скомпилирован из TypeScript или его авторы позаботились о включении деклараций типов в его NPM-пакет, тогда вам повезло. Просто установите код и можете его использовать с полноценной поддержкой типов.

Вот некоторые примеры NPM-пакетов, имеющих встроенные декларации типов:

```
npm install rxjs
npm install ava
npm install @angular/cli
```

Если устанавливаемый код не был скомпилирован из TypeScript, то возможно несоответствие поставляемых с ним деклараций типов коду, который они описывают. Это редкий случай (особенно для популярных пакетов), но учитывать его стоит.

## JavaScript, имеющий декларации типов на DefinitelyTyped

Даже если импортируемый код не содержит декларации типов, они, вероятно, доступны на DefinitelyTyped (https://github.com/DefinitelyTyped/DefinitelyTyped) — в централизованном и поддерживаемом сообществом репозитории, содержащем внешние декларации модулей для открытых проектов.

Чтобы проверить, имеет ли установленный пакет декларации на Defibitely-Typed, либо воспользуйтесь поиском в TypeSearch (https://microsoft.github.io/TypeSearch/), либо попробуйте установить декларации. Все декларации DefinitelyTyped опубликованы на NPM в области @types. Поэтому можете применить к ним `npm install`:

```
npm install lodash –save                 # Установить Lodash
npm install @types/lodash --save-dev     # Установить декларации
                                           типов для Lodash
```

В большинстве случаев при `npm install` вы будете использовать флаг `–save-dev`, чтобы добавить установленные декларации типов в поле `devDependencies` файла package.json.

Поскольку декларации типов на DefinitelyTyped зависят от сообщества, они могут оказаться неполными, неточными или устаревшими. Несмотря на то что наиболее популярные пакеты содержат правильно оформленные декларации, если вы посчитаете, что их можно улучшить, то займитесь этим и затем верните обратно на DefinitelyTyped (https://github.com/DefinitelyTyped/DefinitelyTyped/pulls), чтобы другие пользователи TypeScript смогли оценить результат ваших усилий.

## JavaScript, не имеющий деклараций типов на DefinitelyTyped

Это самый редкий случай. При нем у вас есть несколько вариантов реагирования, начиная с простейшего и менее безопасного и заканчивая более длительным и более безопасным:

1. *Поместите в белый список нетипизированный импорт*, добавив над ним директиву `// @ts-ignore`. TypeScript позволит использовать этот нетипизированный модуль, но сам модуль, включая все его содержимое, будет типизирован как `any`:

   ```
   // @ts-ignore
   import Unsafe from 'untyped-module'

   Unsafe // any
   ```

2. *Поместите в белый список все применения этого модуля*, создав пустой файл деклараций типов и заглушив модуль. Например, если вы установили редко используемый пакет nearby-ferret-alerter, то можете создать новую декларацию типов (types.d.ts или др.) и добавить в нее внешнюю декларацию типов:

   ```
   // types.d.ts
   declare module 'nearby-ferret-alerter'
   ```

   Это сообщит TypeScript, что существует модуль, который вы можете импортировать (`import alert from 'nearby-ferret-alerter'`), но ничего не сообщит о содержащихся в нем типах. Этот подход несколько лучше первого за счет появления центрального файла types.d.ts, который перечисляет все нетипизированные модули в приложении, но он в той же степени небезопасен, поскольку nearby-ferret-alert и все его экспорты будут по-прежнему типизированы как `any`.

3. *Создайте внешнюю декларацию модуля.* Как и в предыдущем подходе, создайте файл с именем types.d.ts и добавьте в него пустую декларацию (`declare module 'nearby-ferret-alerter'`). Теперь заполните декларацию типов. Результат может выглядеть так:

```
// types.d.ts
declare module 'nearby-ferret-alerter' {
    export default function alert(loudness: 'soft' | 'loud'):
                                    Promise<void>
    export function getFerretCount(): Promise<number>
}
```

   Теперь, когда вы произведете `import alert from 'nearby-ferret-alerter'`, TypeScript будет точно знать тип `alert`, который теперь не `any`, а `(loudness: 'quiet' | 'loud') => Promise<void>`.

4. *Создайте декларацию типов и отправьте ее на NPM.* Если вы достигли третьего варианта, теперь у вас есть локальная декларация типов для модуля. Рассмотрите ее отправку назад на NPM, чтобы следующий нуждающийся в декларации типов для пакета nearby-ferret-alert мог тоже ею воспользоваться. Для этого можно либо создать пул-реквест в Git-репозитории nearby-ferret-alert и напрямую внести декларацию типов, либо, если сопровождающие этот репозиторий не захотят поддерживать еще и декларации TypeScript, внести ваши декларации на DefinitelyTyped.

Написание деклараций типов для стороннего JavaScript-кода достаточно просто, но каким способом это лучше сделать, зависит от типа модуля, который вы типизируете. Существует несколько распространенных шаблонов, встречающихся при типизации различных JavaScript-модулей (начиная с модулей NodeJS, расширений jQuery и примесей в Lodash и заканчивая компонентами `React` и `Angular`). Обратитесь к приложению Г за списком вариантов типизации сторонних JavaScript-модулей.

Автоматическая генерация деклараций типов для нетипизированного кода Javascript сегодня является предметом разработки. Проверяйте dts-gen (https://www.npmjs.com/package/dts-gen) на наличие возможности автоматической генерации деклараций типов для заполнения сторонних JavaScript-модулей.

# Итоги

Есть несколько способов использования JavaScript-кода из TypeScript (табл. 11.1).

**Таблица 11.1.** Способы использования JavaScript из TypeScript

| Подход | Флаги | tsconfig.json |
|---|---|---|
| Импорт нетипизированного JavaScript | `{"allowJs": true}` | Слабая |
| Импорт и проверка JavaScript | `{"allowJs": true, "checkJs": true}` | Средняя |
| Импорт и проверка JavaScript с аннотациями JSDoc | `{"allowJs": true, "checkJs": true, "strict": true}` | Отличная |
| Импорт JavaScript с декларациями типов | `{"allowJs": false, "strict": true}` | Отличная |
| Импорт TypeScript | `{"allowJs": false, "strict": true}` | Отличная |

В этой главе вы рассмотрели аспекты совместного использования JavaScript и TypeScript, начиная с различных видов деклараций типов и их применения до пошаговой миграции существующего проекта JavaScript в TypeScript и использования стороннего JavaScript-кода безопасным и небезопасным способом. Взаимодействие с JavaScript может оказаться одним из самых замысловатых аспектов TypeScript. Но, имея в своем распоряжении все необходимые инструменты, вы сможете проводить его в собственных проектах.

ГЛАВА 12

# Создание и запуск TypeScript

Если вам доводилось развертывать и запускать приложение JavaScript в продакшене, считайте, что вы уже знаете, как запускать приложение TypeScript. Компиляция сделает эти приложения идентичными. Данная глава посвящена разработке и созданию TypeScript-приложений, но в этом процессе не так уж много отличий от разработки приложений на JavaScript. Разобьем тему на четыре раздела и рассмотрим:

- Что необходимо для создания любого приложения в TypeScript.
- Создание и запуск приложений на сервере.
- Создание и запуск приложений в браузере.
- Создание приложений для NPM и их публикацию.

## Создание проекта в TypeScript

Создание TypeScript-проекта достаточно просто. В текущем разделе мы изучим основные идеи для запуска такого проекта в любой среде.

### Схема проекта

Предлагаю хранить исходный код в каталоге верхнего уровня `src/` и компилировать его в `dist/`, являющийся также каталогом верхнего уровня. Такая структура каталогов очень популярна, а разделение исходного кода на два верхнеуровневых каталога облегчит интегрирование других инструментов и поможет исключить сгенерированные артефакты из системы управления версиями.

Старайтесь по возможности придерживаться этого условного соглашения:

```
my-app/
├──dist/
│  ├──index.d.ts
│  ├──index.js
│  └──services/
│    ├──foo.d.ts
│    ├──foo.js
│    ├──bar.d.ts
│    └──bar.js
├──src/
│  ├──index.ts
│  └──services/
│    ├──foo.ts
│    └──bar.ts
```

## Артефакты

При компиляции TypeScript-программы в JavaScript TSC может произвести за вас различные артефакты (табл. 12.1).

**Таблица 12.1**. Артефакты, которые может сгенерировать за вас TSC

| Тип | Расширение файла | Флаг tsconfig.json | По умолчанию |
|---|---|---|---|
| Файлы JavaScript | .js | `{"emitDeclarationOnly": false}` | Да |
| Карты кода | .js.map | `{"sourceMap": true}` | Нет |
| Декларации типов | .d.ts | `{"declaration": true}` | Нет |
| Карты деклараций | .d.ts.map | `{"declarationMap": true}` | Нет |

Первый вид артефактов — файлы JavaScript — вам знаком. TSC компилирует код TypeScript в JavaScript, который вы затем запускаете с помощью JavaScript-платформы вроде NodeJS или Chrome. Если вы запустите tsc yourfile.ts, TSC произведет для него проверку типов и скомпилирует его в JavaScript.

Второй вид артефактов — это карты кода, являющиеся особыми файлами, связывающими каждый элемент сгенерированного кода JavaScript с конкретной колонкой или строкой в изначальном файле TypeScript. Это помогает отладке кода (Chrome DevTools показывают код TypeScript вместо итогового JavaScript), а также позволяет делать отображение строк и колонок в трассировках стека исключений JavaScript обратно на код

TypeScript (инструменты вроде упомянутых в подразделе «Мониторинг ошибок» на с. 316 сделают поиск автоматическим, если вы предоставите им карты кода).

Третий артефакт — это декларации типов, позволяющие другим TypeScript-проектам пользоваться сгенерированными вами типами.

Последний вид — карты деклараций , которые используются для ускорения процесса компиляции проекта (см. подраздел «Проектные ссылки» на с. 314).

Оставшуюся часть главы мы будем изучать, как и зачем генерируются описанные артефакты.

## Регулировка компиляции для целевой среды

JavaScript имеет свои особенности, например быстро развивающуюся спецификацию, обновляемую ежегодно, и невозможность всегда контролировать, какую версию JavaScript реализует платформа, на которой вы запускаете программу. Вдобавок к этому многие программы JavaScript изоморфны, а значит, вы можете запускать их одинаково со стороны сервера и со стороны клиента. Например:

- Если вы запускаете бэкенд-программу на сервере, который контролируете, то можете сами решать, в какой версии JavaScript она будет выполняться.
- Если затем вы выпустите вашу бэкенд-программу JavaScript в качестве открытого проекта, то уже не будете знать, какая версия поддерживается на платформах потребителей. Лучшее, что вы можете сделать в среде NodeJS, — это объявить диапазон поддерживаемых версий, но не в среде браузера.
- Если вы запускаете JavaScript-программу в браузере, то уже не имеете представления о том, в каком браузере будут запускать ее пользователи — в последнем Chrome, Firefox или Edge, поддерживающем большинство новейших возможностей JavaScript, а может, в устаревшей версии одного из упомянутых браузеров, которая лишена таких возможностей, или в старомодном Internet Explorer 8, или во встроенном браузере на PlayStation 4 у вас в гараже. Лучшее, что вы можете сделать, — это определить минимальный набор функций, которые должны поддерживать браузеры пользователей для запуска

приложения, отдельно предоставить использование полифилов для максимального числа этих функций и попытаться определить пользователей действительно старых браузеров, на которых ваше приложение точно не запустится, чтобы показать им сообщение о том, что необходимо обновление.

- Если вы выпускаете изоморфную JavaScript-библиотеку (например, библиотеку журналирования, которая запускается и в браузере, и на сервере), то она должна поддерживать минимальную версию NodeJS, а также множество версий и движков браузерного JavaScript.

Не каждая среда JavaScript изначально поддерживает все возможности JavaScript, но тем не менее вы должны стараться писать код в последних версиях языка. Это можно делать двумя способами.

1. *Транспиляция* (автоматическая конвертация) приложений из последней версии JavaScript в более старую, которая поддерживается целевой платформой. Это нужно для `async` и `await` и циклов `for…of`, которые могут быть автоматически преобразованы в вызовы `.then` и циклы `for` соответственно.
2. *Использование полифилов* (предоставление реализации) для новейших возможностей, отсутствующих в среде выполнения JavaScript, где вы запускаете приложение. Это нужно для поддержки возможностей, предоставляемых стандартной библиотекой JavaScript (вроде `Promise`, `Map` и `Set`), а также для методов прототипов (вроде `Array.prototype.includes` и `Function.prototype.bind`).

В TSC есть встроенная поддержка транспиляции кода в более старые версии JavaScript, но он не может автоматически внедрять полифилы в ваш код. Еще раз повторюсь: TSC будет транспилировать большинство возможностей JavaScript в более старые среды, но он не предоставит реализации для отсутствующих.

В TSC есть три регулятора целевых сред:

- `target` устанавливает версию, в которую нужно транспилировать: es5, es2015 и т. д.;
- `module` устанавливает целевую систему модулей: модули es2015, commonjs, systemjs и т. д.;

- `lib` сообщает TypeScript, какие возможности JavaScript доступны в целевой среде: возможности es5, es2015, dom и т. д. Для реализации возможностей необходимы полифилы, но lib указывает TypeScript на доступные возможности (по умолчанию или посредством полифилов).

Среда, в которой вы планируете запускать приложение, диктует, на какую версию нужно настроить транспиляцию посредством `target` и что указать в `lib`. Если вы не уверены, то используйте es5, которая вполне подходит для обоих пунктов. Что вы выберете в `module`, зависит от вашей цели — это NodeJS или среда браузера. В случае с браузером дополнительно имеет значение, какой загрузчик модулей вы используете.

Чтобы реализовать поддержку необычного набора платформ, поищите поддерживаемые по умолчанию возможности JavaScript в таблицах совместимости Юрия Зайцева (http://kangax.github.io/compat-table/es5/).

Остановимся на `target` и `lib`, оставив `module` для разделов «Запуск TypeScript на сервере» и «Запуск TypeScript в браузере», которые следуют далее на с. 317 и 318.

#### target

Встроенный в TSC транспайлер поддерживает преобразование большинства возможностей JavaScript в более старые версии, поэтому вы можете писать код в последней версии TypeScript и транспилировать его в любую версию JavaScript, поддержка которой требуется. Поскольку TypeScript поддерживает последние возможности JavaScript (вроде `async` и `await`, которые на момент написания книги поддерживаются еще не всеми ведущими платформами JavaScript), вы почти всегда будете пользоваться преимуществом транспайлера для конвертации кода в то, что будет актуально для NodeJS и браузеров.

Рассмотрим, какие конкретные функциональные возможности JavaScript могут быть транспилированы TSC, а какие — нет (табл. 12.2 и 12.3)[1].

[1] Если вы используете такие функции языка, которые TSC не транспилирует, и целевая среда их не поддерживает, то, как правило, можно использовать плагин Babel, который произведет транспиляцию. Самые свежие плагины можно найти в поисковике по тегу `"babel plugin <название функции>"`.

Раньше новые ревизии JavaScript выпускались раз в несколько лет и сопровождались повышением версии языка (ES1, ES3, ES5, ES6). Начиная с 2015 года цикл релизов сократился до одного года и версии языка стали носить имя года выпуска (ES2015, ES2016 и т. д.). Тем не менее некоторые возможности JavaScript получают поддержку TypeScript до их включения в конкретную версию JavaScript. Такие возможности мы обозначаем как ESNext (следующая ревизия).

**Таблица 12.2.** TSC может транспилировать

| Версия | Характеристики |
|---|---|
| ES2015 | `const`, `let`, `for..of` циклов, массив или объект распространения (...), помеченные строки шаблона, классы, генераторы, функции стрелок, параметры функций по умолчанию, параметры остальных функций, деструктуризация, а также декларации, или задания, или параметры |
| ES2016 | Оператор возведения в степень (`**`) |
| ES2017 | `async`-функции и `await`-промисы |
| ES2018 | `async`-итераторы |
| ES2019 | Необязательный параметр в предложении catch |
| ESNext | Числовые разделители (123_456) |

**Таблица 12.3.** TSC не может транспилировать

| Version | Feature |
|---|---|
| ES5 | Объекты getter / setters |
| ES2015 | Флаги регулярных выражений `y` и `u` |
| ES2018 | Флаг регулярных выражений `s` |
| ESNext | BigInt (`123n`) |

Чтобы установить цель транспиляции, откройте tsconfig.json и установите поле `target` как:

- es3 для ECMAScript 3;
- es5 для ECMAScript 5 (хорошее решение, если вы не уверены);
- es6 или es2015 для ECMAScript 2015;

- es2016 для ECMAScript 2016;
- es2017 для ECMAScript 2017;
- es2018 для ECMAScript 2018;
- esnext для любой последней ревизии ECMAScript.

Например, для компиляции в ES5:

```
{
    "compilerOptions": {
        "target": "es5"
    }
}
```

#### Библиотеки

Как я уже писал, при транспиляции кода в более старую версию JavaScript есть одна неувязка: несмотря на то что основная часть функционала языка может быть безопасно транспилирована (`let` в `var`, `class` в `function`), если целевая среда не поддерживает новые возможности библиотеки, вам по-прежнему нужно с помощью полифилов добавлять некоторую функциональность самостоятельно (например, утилиты `Promise` и `Reflect`, а также структуры данных вроде `Map`, `Set` и `Symbol`). Если в роли целевой среды выступает новейший Chrome, Firefox или Edge, то полифилы, как правило, не требуются. Но если вы нацелены на среду, отстающую на несколько версий или несколько сред NodeJS, то потребуется дополнить посредством полифилов недостающий функционал.

Если вы запускаете приложение в браузере, постарайтесь не раздуть размер связки JavaScript, включив все подряд полифилы независимо от того, нужны они в среде назначения или нет. Целевая платформа наверняка уже поддерживает некоторые из нужных вам возможностей. Вместо этого используйте сервис вроде Polyfill.io (https://polyfill.io/v3/), чтобы загрузить только недостающие браузеру полифилы.

К счастью, вам не придется прописывать полифилы самим. Вместо этого вы можете установить их из популярных библиотек вроде core-js (https://www.npmjs.com/package/core-js) или добавить автоматически, запустив про-

шедший проверку типов код через Babel с @babel/polyfill (https://babeljs.io/docs/en/babel-polyfill).

Как только вы добавили полифилы в код, настало время сообщить TSC, что целевая среда гарантированно поддерживает полизаполненные функции, указав это в поле `lib` файла tsconfig.json. Например, если бы вы внесли с помощью полифилов весь функционал ES2015 и дополнили его Array.prototype.includes из ES2016, то могли бы использовать следующую конфигурацию:

```
{
    "compilerOptions": {
        "lib": ["es2015", "es2016.array.includes"]
    }
}
```

Если вы запускаете код в браузере, то также активируйте декларации типов DOM для `window`, `document` и всех других API, появляющихся при запуске JavaScript в браузере:

```
{
    "compilerOptions": {
    "lib": ["es2015", "es2016.array.include", "dom"]
    }
}
```

Для ознакомления с полным списком поддерживаемых библиотек запустите `tsc -help`.

## Активация карт кода

Карты кода — это способ связать транспилированный код с его исходником. Большинство инструментов разработки (вроде Chrome DevTools), платформ регистрации ошибок и журналирования, а также инструментов сборки знакомы с картами кода. Поскольку типичная линия сборки может производить код, весьма отличный от начального (например, компилировать TypeScript в ES5 JavaScript, встряхивать его дерево в Rollup, предварительно вычислять с помощью Prepack, а затем минифицировать посредством Uglify), то использование карт кода по ходу сборки может существенно облегчить отладку итогового кода JavaScript.

В целом, использовать карты кода в разработке и поставлять их в продакшен как в среде браузера, так и для серверов — хорошая идея. Но есть нюанс: если полная безопасность кода для браузера неочевидна, не стоит использовать их в продакшене.

### Проектные ссылки

По мере роста приложения TSC будет требоваться все больше времени для проверки его типов и компиляции. Количество необходимого для этого времени растет линейно в соответствии с ростом базы кода. При локальной разработке медленное время инкрементной компиляции может замедлить разработку в целом, мешая получать удовольствие от работы в TypeScript.

Для разрешения этой проблемы в TSC реализована функция, названная *проектными ссылками*, которая серьезно ускоряет компиляцию, в том числе сокращая время инкрементного компилирования. Для любого проекта, имеющего сотни файлов и более, такие ссылки просто необходимы.

Используйте их так.

1. Разделите проект на несколько проектов. Проект — это просто каталог, содержащий tsconfig.json и код TypeScript. Попробуйте разделить код так, чтобы части, требующие одновременного обновления, находились в одном каталоге.
2. В каждом каталоге проекта создайте tsconfig.json, содержащий по меньшей мере следующее:

```
{
    "compilerOptions": {
        "composite": true,
        "declaration": true,
        "declarationMap": true,
        "rootDir": "."
    },
    "include": [
        "./**/*.ts"
    ],
    "references": [
```

```
        {
            "path": "../myReferencedProject",
            "prepend": true
        }
    ],
}
```

Основное здесь:

- `composite`, сообщающий TSC, что этот каталог является подпроектом более крупного проекта;
- `declaration`, побуждающий TSC создать для этого проекта файлы деклараций `.d.ts`. Принцип работы проектных ссылок подразумевает, что у проектов есть доступ к файлам деклараций типов и сгенерированным JavaScript-кодам друг друга, но не к исходным файлам TypeScript. Это создает границу, за которой TSC не будет перепроверять или перекомпилировать код: если вы обновите строку кода в подпроекте `A`, TSC не будет проверять другой подпроект `B`. Все, что ему нужно проверить на предмет ошибок типов, — декларации типов `B`. Это делает проектные ссылки эффективными при повторной сборке больших проектов;
- `declarationMap`, сообщающий TSC, что нужно создать карты кода для сгенерированных деклараций типов;
- `references` — массив подпроектов, от которых зависит текущий подпроект. Каждый `path` ссылки должен указывать либо на каталог, содержащий `tsconfig.json`, либо непосредственно на файл конфигурации TSC (если только он не назван `tsconfig.json`). `prepend` произведет конкатенацию кода JavaScript и карт кода, сгенерированных подпроектом, на который ведет ссылка, с кодом JavaScript и картами кода, сгенерированными текущим подпроектом. Обратите внимание, что, если вы не используете `outFile`, функция `prepend` вам не потребуется;
- `rootDir` явно определяет, что этот подпроект должен быть скомпилирован относительно корневого проекта (`.`). В качестве альтернативы можно определить каталог `outDir`, являющийся подкаталогом `outDir` корневого проекта.

3. Создайте корневой `tsconfig.json`, ссылающийся на любые подпроекты, на которые еще не ссылается ни один другой подпроект:

```
{
    "files": [],
    "references": [
        {"path": "./myProject"},
        {"path": "./mySecondProject"}
    ]
}
```

4. Теперь при компиляции проекта в TSC используйте флаг `build`, чтобы он учел проектные ссылки:

```
tsc --build # Или краткий вариант tsc -b
```

На момент написания книги проектные ссылки являются новой возможностью TypeScript, имеющей свои недоработки. Поэтому при их использовании учтите следующее:

- Пересоберите весь проект (с помощью `tsc -b`) после его клонирования или повторного получения, чтобы восстановить любые недостающие или устаревшие файлы .d.ts. В качестве альтернативы можете проверить сгенерированные файлы .d.ts.
- Не используйте с проектными ссылками `noEmitOnError: false` — TSC будет всегда устанавливать эту опцию как `true`.
- Самостоятельно убедитесь, что данный подпроект добавлен к началу не более чем одного подпроекта. В противном случае проект, добавленный к двум подпроектам, при компиляции будет появляться дважды. Простая ссылка одного подпроекта на другой не приведет к такому поведению.

## Мониторинг ошибок

TypeScript предупреждает об ошибках во время компиляции, но вам также понадобится способ узнавать об исключениях, которые получают пользователи во время выполнения, чтобы попытаться предотвратить их на уровне компиляции (или по крайней мере исправить баг, вызывающий ошибку среды выполнения). Используйте инструменты для мониторинга ошибок вроде Sentry (https://sentry.io/welcome/) или Bugsnag (https://www.bugsnag.com/) для регистрации и сопоставления ошибок среды выполнения.

**ИСПОЛЬЗОВАНИЕ РАСШИРЕНИЯ ДЛЯ УМЕНЬШЕНИЯ РУТИННОГО КОДА В TSCONFIG.JSON**

Скорее всего, вы захотите, чтобы все ваши подпроекты разделяли одни и те же настройки компиляции, поэтому создайте базовый tsconfig.json в корневом каталоге, который смогут расширить файлы tsconfig.json подпроектов:

```
{
    "compilerOptions": {
          "composite": true,
          "declaration": true,
          "declarationMap": true,
          "lib": ["es2015", "es2016.array.include"],
          "rootDir": ".", "sourceMap": true,
          "strict": true,
          "target": "es5",
     }
}
```

А затем используйте опцию `extends` в этих файлах tsconfig.json подпроектов, чтобы добавить расширение:

```
{
     "extends": "../tsconfig.base",
     "include": [
          "./**/*.ts"
     ],
     "references": [
          {
               "path": "../myReferencedProject",
               "prepend": true
          }
     ],
}
```

# Запуск TypeScript на сервере

Для запуска TypeScript-кода в среде NodeJS нужно просто скомпилировать его в ES2015 JavaScript (или ES5, если вы планируете запуск на устаревшей версии NodeJS) и указать `commonjs` в пункте `module` файла tsconfig.json:

```
{
    "compilerOptions": {
        "target": "es2015",
        "module": "commonjs"
    }
}
```

В ES2015 вызовы `import` и `export` будут скомпилированы в `require` и `module.exports` соответственно и ваш код будет запускаться на NodeJS, не нуждаясь в дополнительных связках.

Если вы используете карты кода (рекомендуется), то вам нужно предоставить их и в NodeJS. Для этого просто возьмите пакет source-map-support (https://www.npmjs.com/package/source-map-support) из NPM и следуйте инструкциям при его установке. Большинство инструментов вроде PM2 (https://www.npmjs.com/package/pm2), Winston (https://www.npmjs.com/package/winston) и Sentry (https://sentry.io/welcome/), предназначенных для мониторинга процессов, журналирования и регистрации ошибок, имеют встроенную поддержку карт кода.

## Запуск TypeScript в браузере

Компиляция TypeScript для запуска в браузере требует больше усилий.

Сначала выберите систему модулей, в которую будете компилировать. Если вы публикуете библиотеку (например, на NPM), то согласно общему правилу стоит придерживаться `umd`, чтобы максимально увеличить ее совместимость с различными бандлерами модулей, которые могут использоваться разработчиками.

Если вы собираетесь просто использовать код сами, то формат для компиляции будет зависеть от вашего бандлера модулей. Проверьте его документацию — например, Webpack и Rollup лучше работают с модулями ES2015, а Browserfy требует модулей CommoJS. Вот несколько рекомендаций:

- Если вы используете загрузчик модулей SystemJS (https://github.com/systemjs/systemjs), установите `module` как `systemjs`.
- Если вы запускаете код через бандлер, работающий с модулями ES2015, вроде Webpack (https://webpack.js.org/) или Rollup (https://github.com/rollup/rollup), установите `module` как es2015 или выше.

- Если вы применяете бандлер для модулей ES2015, а ваш код использует динамический импорт (см. подраздел «Динамический импорт» на с. 270), установите `module` как `esnext`.
- Если вы создаете библиотеку для использования в других проектах и не применяете к коду никаких дополнительных шагов сборки после tsc, то максимально увеличьте совместимость с загрузчиками, которые могут использовать другие разработчики, установив `module` как `umd`.
- Если вы связываете модуль бандлером для CommonJS вроде Browserfy (https://github.com/browserify/browserify), установите `module` как `commonjs`.
- Если вы планируете загружать код с помощью RequireJS (https://requirejs.org/) или другого загрузчика модулей AMD, установите `module` как `amd`.
- Если вы хотите, чтобы экспорты верхнего уровня были глобально доступны в объекте `window`, установите `module` как `none`. Обратите внимание, что если ваш код находится в режиме модулей (см. подраздел «Режим модулей против режима скриптов» на с. 273), то TSC попытается усмирить ваш порыв насолить другим разработчикам и все равно произведет компиляцию в `commonjs`.

Следующим шагом настройте линию сборки так, чтобы компилировать весь код TypeScript в один файл JavaScript (обычно это называется связкой) или в набор JavaScript-файлов. Несмотря на то что в небольших проектах TSC может сделать это за вас с помощью флага `outFile`, этот флаг позволяет генерацию только связок SystemJS и AMD. А поскольку TSC не поддерживает встроенные плагины и интеллектуальное разделение кода тем же способом, каким это делают специализированные инструменты сборки вроде Webpack, то вы наверняка захотите использовать более функциональный бандлер.

Именно поэтому во фронтенд-проектах вам изначально следует использовать более функциональные инструменты сборки. Для любых из них существуют плагины TypeScript. Например:

- ts-loader (https://www.npmjs.com/package/ts-loader) для Webpack (https://webpack.js.org/);
- tsify (https://www.npmjs.com/package/tsify) для Browserify (https://www.npmjs.com/package/browserify);

- @babel/preset-typescript (https://github.com/Microsoft/TypeScript-Babel-Starter) для Babel (https://babeljs.io/);
- gulp-typescript (https://www.npmjs.com/package/gulp-typescript) для Gulp (https://gulpjs.com/);
- grunt-ts (https://www.npmjs.com/package/grunt-ts) для Grunt (https://gruntjs.com/).

Несмотря на то что полноценное рассмотрение оптимизации связки JavaScript кода для ускорения загрузки выходит за рамки этой книги, некоторые отдельные советы, не относящиеся к TypeScript, стоит упомянуть:

- Сохраняйте код модульным и избегайте неявных зависимостей (которые могут случиться при присваивании элементов глобальному window или другим глобальным объектам), чтобы используемый вами инструмент сборки мог более точно анализировать граф зависимостей проекта.
- Используйте динамический импорт для загрузки кода по требованию, который не применяется для загрузки начальной страницы, чтобы не препятствовать отображению страниц без необходимости.
- Воспользуйтесь функциональностью инструмента сборки для автоматического разделения кода, чтобы избежать ненужной загрузки больших частей кода и замедления работы страниц.
- Выработайте подход для замера времени загрузки страницы либо синтетически, либо, что было бы идеально, исходя из реальных данных пользователей. По мере роста вашего приложения начальное время загрузки может постепенно увеличиваться, а оптимизировать вы его сможете, только измерив. В такой ситуации бесценны инструменты вроде New Relic (https://newrelic.com/) и Datadog (https://www.datadoghq.com/).
- Старайтесь сохранять вашу сборку в продакшене максимально приближенной к сборке разработки. Чем больше они будут отличаться, тем сложнее будет устранять баги, появляющиеся только в продакшене.
- Ну и последнее. При отправке кода TypeScript для запуска в браузере выработайте метод для добавления недостающего функционала. Это может быть стандартный набор полифилов, основанный на том, какие функции поддерживает браузер пользователя.

# Публикация TypeScript-кода на NPM

Компилировать код для использования в других TypeScript- и JavaScript-проектах достаточно легко. Нужно учесть несколько рекомендаций.

- Генерируйте карты кода, которые помогут производить его отладку.
- Компилируйте в ES5, чтобы другие пользователи могли легко собрать и запустить ваш код.
- Отнеситесь серьезно к тому, в какой формат модулей компилировать (UMD, CommonJS, ES2015 и т. д.).
- Генерируйте декларации типов, чтобы другие пользователи TypeScript располагали типами для вашего кода.

Начните с компиляции TypeScript-кода в JavaScript с помощью `tsc` и генерации соответствующих деклараций типов. Обязательно настройте tsconfig.json для максимальной совместимости с наиболее популярными средами JavaScript и системами сборки (раздел «Создание проекта в TypeScript», с. 306):

```
{
"compilerOptions": {
    "declaration": true,
    "module": "umd",
    "sourceMaps": true,
    "target": "es5"
    }
}
```

Затем для исключения публикации исходного кода на NPM внесите его в черный список в .npmignore. Таким образом вы избежите увеличения размера пакета. А в .gitignore исключите сгенерированные артефакты из репозитория Git, чтобы избежать его засорения:

```
# .npmignore

*.ts # Игнорировать файлы .ts
!*.d.ts # Допустить .d.ts

# .gitignore

*.d.ts # Игнорировать файлы .d.ts
*.js # Игнорировать .js
```

Если вы придерживались рекомендованной схемы проекта и хранили исходные файлы в src/, а сгенерированные файлы в dist/, то ваши файлы .ignore будут еще проще:

```
# .npmignore

src/ # Игнорировать #исходные файлы

# .gitignore

dist/ # Игнорировать сгенерированные файлы
```

В завершение добавьте поле `"types"` в файл package.json проекта, чтобы указать, что он содержит декларации типов (это не обязательно, но полезно для пользователей TypeScript), а также добавьте скрипт для сборки пакета перед его отправкой, чтобы убедиться, что ваш JavaScript-код пакета, декларации типов и карты кода всегда актуальны и соответствуют коду TypeScript, из которого были скомпилированы:

```
{
    "name": "my-awesome-typescript-project",
    "version": "1.0.0",
    "main": "dist/index.js",
    "types": "dist/index.d.ts",
    "scripts": {
        "prepublishOnly": "tsc -d"
    }
}
```

Вот и все! Теперь, когда вы опубликовали пакет, NPM автоматически скомпилирует ваш код TypeScript в формат, пригодный как для пользователей TypeScript (с безопасностью типов), так и для тех, кто использует JavaScript (с некоторой безопасностью типов, если их редактор это поддерживает).

## Директивы с тремя слешами

В TypeScript реализована малоизвестная, редко используемая и чаще всего необновленная функция, называемая директивами с тремя слешами.

Эти директивы являются особым форматом комментариев TypeScript, служащих в качестве инструкций для TSC.

Они бывают нескольких видов. В этом разделе мы рассмотрим два из них: `types`, используемую для пропуска импорта всего модуля, предполагающего только импорт типов, и `amd-module`, служащую для именования модулей AMD. За полной информацией обратитесь к приложению Д.

## Директива types

В зависимости от того, что вы импортировали из модуля, TypeScript не всегда требуется генерировать вызов `import` или `require` при компиляции кода в JavaScript. Если у вас есть инструкция `import`, чей экспорт используется только в позиции типа в модуле (то есть вы просто импортировали из этого модуля тип), TypeScript не будет генерировать для этого `import` код JavaScript — рассматривайте его как существующий только на уровне типов. Эта функция называется пропуском импорта.

Исключением из этого правила являются импорты, используемые для побочных эффектов: если вы импортируете весь модуль (не импортируя конкретный экспорт или символ подстановки из этого модуля), то этот импорт будет генерировать код JavaScript при компиляции из TypeScript. Вы можете делать это, например, если хотите убедиться, что внешний тип, определенный в режиме скриптов, доступен в вашей программе (как мы делали в разделе «Безопасное расширение прототипа» на с. 193). Например:

```
// global.ts
type MyGlobal = number

// app.ts
import './global'
```

После компиляции app.ts в JavaScript посредством tsc app.ts вы заметите, что импорт ./global не был пропущен:

```
// app.js
import './global'
```

Если вы часто пишете подобные импорты, проверьте, действительно ли импорту нужно использовать побочные эффекты и нет ли другого способа

переписать код, более явно выразив, какое значение или тип вы импортируете (например, `import {MyType} from './global'` который TypeScript пропустит, вместо `import './global'`). Либо посмотрите, можете ли вы включить внешний тип в поле `types`, `files` или `include` файла tsconfig.json и полностью избежать импорта.

Если ни один из этих вариантов не подходит и вы хотите продолжить импортировать целые модули, но избежать JavaScript-вызова `import` или `require` для этого импорта, используйте директиву `types`. Директива с тремя слешами (`///`) сопровождается одним из предусмотренных XML-тегов, каждый из которых имеет свой набор необходимых атрибутов. Для директивы `types` это выглядит так:

- Объявление зависимости во внешней декларации типа:

```
/// <reference types="./global" />
```

- Объявление зависимости в @types/jasmine/index.d.ts:

```
/// <reference types="jasmine" />
```

Чрезмерное использование этой директивы должно побудить вас поискать способ, при котором не придется слишком часто полагаться на внешние типы.

## Директива amd-module

При компиляции TypeScript-кода в формат модулей AMD (что указано в tsconfig.json как `{"module": "amd"}`) TypeScript по умолчанию генерирует анонимные модули AMD. Директиву в данном случае можно использовать, чтобы задать модулям имена.

К примеру, у вас есть следующий код:

```
export let LogService =
    { log() {
        // ...
    }
}
```

При его компиляции в формат модулей AMD, TSC генерирует следующий код JavaScript:

```
define(['require', 'exports'], function(require,
    exports) { exports. esModule = true
    exports.LogService = {
        log() {
            // ...
        }
    }
})
```

Если вы знакомы с форматом модулей AMD, то могли заметить, что это безымянный модуль. Чтобы задать ему имя, используйте директиву с тремя слешами и `amd-module`:

```
/// <amd-module name="LogService" /> ❶
export let LogService = { ❷
    log() {
        // ...
    }
}
```

❶ Используем директиву `amd-module` и устанавливаем в ней атрибут `name`.

❷ Остальной код остается неизменным.

Теперь при повторной компиляции через TSC в формат модуля AMD, мы получаем следующий JavaScript-код:

```
/// <amd-module name='LogService' />
define('LogService', ['require', 'exports'], function(require, exports) {
    exports. esModule = true
    exports.LogService = {
        log() {
            // ...
        }
    }
})
```

Используйте директиву `amd-module`, чтобы облегчить связывание и отладку кода (или по возможности перейдите на более современный формат модулей вроде ES2015).

## Итоги

В этой главе вы рассмотрели все, что вам нужно знать для создания и запуска своего TypeScript-приложения в продакшене браузера или сервера. Вы научились выбирать версию JavaScript для компиляции, отмечать доступные в вашей среде библиотеки (и добавлять недостающий функционал посредством полифилов), а также создавать и отправлять карты кода вместе с приложением, чтобы облегчить его отладку в продакшене и локальную разработку. Затем вы узнали, как применять модули в проекте, чтобы сохранить скорость его компиляции. В завершение вы увидели запуск приложения на сервере и в браузере, публикацию TypeScript-кода на NPM, принцип работы пропуска импорта и научили пользователей AMD именовать модули с помощью директивы с тремя слешами.

ГЛАВА 13

# Итоги

Вот мы и приближаемся к концу нашего совместного путешествия.

В нем мы обсудили, что такое типы и чем они полезны, как работает TSC, какие типы поддерживает TypeScript, как его система типов производит вывод, уточнение, расширение, тотальность и проверяет совместимость. Далее мы перечислили правила контекстной типизации, принципы работы вариативности, а также рассмотрели использование операторов типов. Мы прошлись по темам функций, классов, интерфейсов, итераторов и генераторов, перегрузок, полиморфных типов, примесей, декораторов, а также узнали различные запасные решения, которые можно изредка использовать, жертвуя безопасностью ради сдачи кода в рамках дедлайна. Мы изучили несколько способов безопасной обработки исключений (их плюсы и минусы), а также научились использовать типы, чтобы сделать конкурентные, параллельные и асинхронные программы безопасными. Погрузились в тему использования TypeScript с популярными фреймворками вроде Angular и React, а затем рассмотрели работу пространств имен и модулей. Мы изучили использование, создание и развертывание TypeScript во фронтенд и бэкенд, поговорили о том, как пошагово мигрировать в него код, как использовать декларации типов, публиковать свой код на NPM для других разработчиков, научились безопасно использовать сторонний код и создавать собственные проекты TypeScript.

Надеюсь, мне удалось вдохнуть в вас дух статических типов и теперь время от времени вы будете сначала прописывать программы в типах, а затем реализовывать их. Думаю, что вы обрели глубокое интуитивное понимание использования типов для повышения безопасности ваших приложений. Рад, если у вас хоть немного изменился взгляд на мир и теперь при написании кода вы будете мыслить типами.

На данный момент у вас уже есть достаточно материала, чтобы делиться знаниями о TypeScript с другими. Выступайте в защиту безопасности

и способствуйте улучшению и облегчению написания кода вашими коллегами и друзьями.

Закончу на том, что посоветую продолжать освоение нового. TypeScript, вероятно, не первый и не последний известный вам язык. Продолжайте изучение новых подходов к программированию, рассматривайте типы с новых ракурсов, а также по-новому оценивайте баланс между безопасностью и простотой использования. Кто знает, может именно вы создадите будущую более совершенную аналогию TypeScript, а я однажды напишу об этом книгу...

ПРИЛОЖЕНИЕ А

# Операторы типов

TypeScript поддерживает богатый набор операторов для работы с типами. Используйте табл. А.1 в качестве подручной справки на случай, когда вам понадобится узнать об операторах подробнее.

**Таблица А.1.** Операторы типов

| Оператор типа | Синтаксис | Использование с | Подробнее |
|---|---|---|---|
| Запрос типа | `typeof`, `instanceof` | Любым типом | «Уточнение», с. 160, «Классы объявляют и значения, и типы», с. 127 |
| Ключи | `keyof` | Типом объектов | «Оператор keyof», с. 169 |
| Поиск свойства | `O[K]` | Типом объектов | «Оператор подключения (key in)», с. 167 |
| Отображенный тип | `[K in O]` | Типом объектов | «Отображенные типы», с. 173 |
| Сложение | `+` | Типом объектов | «Отображенные типы», с. 173 |
| Вычитание | `-` | Типом объектов | «Отображенные типы», с. 173 |
| Только чтение | `readonly` | Типом объектов, массивов, кортежей | «Объекты», с. 42, «Классы и наследование», с. 111, «Массивы и кортежи только для чтения», с. 56 |
| Опциональность | `?` | Типом объектов, кортежей, параметров функций | «Объекты», с 42, «Кортежи», с. 55, «Предустановленные и опциональные параметры», с. 69 |
| Условный тип | `?` | Обобщенным типом, псевдонимами типов, типом параметров функций | «Условные типы», с. 180 |
| Утверждение не null | `!` | Типом, допускающим null | «Ненулевые утверждения», с. 187, «Утверждения типов», с. 185 |
| Предустановка параметра обобщенного типа | `=` | Обобщенным типом | «Предустановки обобщенных типов», с. 107 |
| Утверждение типа | `as, <>` | Любым типом | «Утверждения типов», с. 185, «Тип const», с. 156 |
| Защита типа | `is` | Возвращаемым типом функции | «Пользовательские защиты типов», с. 178 |

ПРИЛОЖЕНИЕ Б

# Утилиты типов

Утилиты типов в TypeScript изначально привязаны к его стандартной библиотеке. В табл. Б.1 перечисляются те, что доступны на момент написания книги.

Актуальную информацию можно найти на es5.d.ts (https://github.com/Microsoft/TypeScript/blob/master/src/lib/es5.d.ts).

**Таблица Б.1.** Утилиты типов

| Утилита типов | Используется с | Описание |
|---|---|---|
| `Constructor-Parameters` | Типом конструкторов класса | Кортеж типов параметров конструктора класса |
| `Exclude` | Типом объединения | Исключает тип из другого типа |
| `Extract` | Типом объединения | Выбирает тип, совместимый с другим типом |
| `InstanceType` | Типом конструкторов класса | Тип экземпляра, получаемый посредством `new` конструктора класса |
| `NonNullable` | Типом, допускающим `null` | Исключает `null` и `undefined` из типа |
| `Parameters` | Типом функции | Кортеж типов параметров функции |
| `Partial` | Типом объектов | Делает все свойства объекта опциональными |
| `Pick` | Типом объектов | Подтип типа объекта с подмножеством его ключей |
| `Readonly` | Типом массивов, объектов и кортежей | Делает все свойства объекта, массива или кортежа только для чтения |
| `ReadonlyArray` | Любым типом | Создает неизменяемый массив заданного типа |
| `Record` | Типом объектов | Отображает тип ключа на тип значения |
| `Required` | Типом объектов | Делает все свойства объекта необходимыми |
| `ReturnType` | Типом функций | Возвращаемый тип функции |

ПРИЛОЖЕНИЕ B

# Область действия деклараций

Декларации в TypeScript имеют различные варианты поведения, необходимого для моделирования типов и значений. Так же как и в JavaScript, они могут быть перегружены несколькими способами. Это приложение рассматривает два варианта поведения, разделяя декларации, генерирующие типы, значения и допускающие слияние.

## Генерирует ли декларация тип

Одни декларации в TypeScript создают тип, другие — значение, а некоторые — и то и другое. Таблица B.1 представляет их перечень.

**Таблица B.1.** Генерирует ли декларация тип

| Ключевое слово | Генерирует тип | Генерирует значение |
|---|---|---|
| `class` | Да | Да |
| `const, let, var` | Нет | Да |
| `enum` | Да | Да |
| `function` | Нет | Да |
| `interface` | Да | Нет |
| `namespace` | Нет | Да |
| `type` | Да | Нет |

## Допускает ли декларация слияние

Слияние деклараций — это ключевая особенность TypeScript. Воспользуйтесь ею для создания более богатых API, улучшения модульной системы кода и повышения его безопасности.

Таблица В.2 взята из раздела «Слияние деклараций» на с. 279. Она наглядно демонстрирует, для каких деклараций TypeScript будет производить слияние.

**Таблица В.2.** Допускает ли декларация слияние

| | | В | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Знч. | Класс | Перчисл. | Функция | Псвд. типа | Интерфейс | Пр-во имен | Модуль |
| Из | Значение | Нет | Нет | Нет | Нет | Да | Да | Нет | — |
| | Класс | — | Нет | Нет | Нет | Нет | Да | Да | — |
| | Перечисл. | — | — | Да | Нет | Нет | Нет | Да | — |
| | Функция | — | — | — | Нет | Да | Да | Да | — |
| | Псв. типа | — | — | — | — | Нет | Нет | Да | — |
| | Интерфейс | — | — | — | — | — | Да | Да | — |
| | Пр. имен | — | — | — | — | — | — | Да | — |
| | Модуль | — | — | — | — | — | — | — | Да |

ПРИЛОЖЕНИЕ Г

# Правила написания файлов деклараций для сторонних модулей JavaScript

Это приложение рассматривает несколько ключевых элементов структуры и паттернов, регулярно встречающихся при типизации сторонних модулей. Более подробно ознакомиться с типизацией стороннего кода вы можете в разделе «JavaScript, не имеющий деклараций типов на DefinitelyTyped» на с. 303.

Поскольку файлы деклараций модулей должны находиться в файлах .d.ts, они не могут содержать значения и при объявлении типов модулей потребуется использовать ключевое слово declare для подтверждения, что значения данных типов действительно экспортируются вашим модулем. Таблица Г.1 представляет краткую сводку стандартных деклараций и эквивалентных им деклараций типов.

**Таблица Г.1.** Декларации TypeScript и их эквиваленты, имеющие только типы

| .ts | .d.ts |
|---|---|
| `var a = 1` | `declare var a: number` |
| `let a = 1` | `declare let a: number` |
| `const a = 1` | `declare const a: 1` |
| `function a(b) { return b.toFixed() }` | `declare function a(b: number): string` |
| `class A { b() { return 3 } }` | `declare class A { b(): number }` |
| `namespace A {}` | `declare namespace A {}` |
| `type A = number` | `type A = number` |
| `interface A { b?: string }` | `interface A { b?: string }` |

# Типы экспорта

От того, какой экспорт использует ваш модуль (ES2015, CommonJS или глобальный объект), будет зависеть написание файлов деклараций.

## Глобальные объекты

```
// Глобальная переменная
declare let someGlobal: GlobalType

// Глобальный класс
declare class GlobalClass {}

// Глобальная функция
declare function globalFunction(): string

// Глобальное перечисление
enum GlobalEnum {A, B, C}

// Глобальное пространство имен
namespace GlobalNamespace {}

// Глобальный псевдоним типа
type GlobalType = number

// Глобальный интерфейс
interface GlobalInterface {}
```

Каждая из этих деклараций будет доступна глобально для каждого файла проекта без явного импорта. Вы могли бы использовать someGlobal в любом файле проекта без необходимости его импорта, но при выполнении потребовалось бы присвоить `someGlobal` глобальному пространству имен (`window` в браузерах или `global` в NodeJS).

Старайтесь избегать `import` и `export` в файлах деклараций, чтобы сохранить файл в режиме скриптов.

### Экспорт ES2015

Если ваш модуль использует экспорты ES2015 (ключевое слово export), то просто замените `declare` (подтверждающее, что глобальная переменная определена) на `export` (подтверждающее, что экспортируется привязка ES2015):

```
// Экспорт по умолчанию
declare let defaultExport: SomeType
export default defaultExport

// Проименованный экспорт
export class SomeExport {
    a: SomeOtherType
}

// Экспорт класса
export class ExportedClass {}

// Экспорт функции
export function exportedFunction(): string

// Экспорт перечисления
enum ExportedEnum {A, B, C}

// Экспорт пространства имен
export namespace SomeNamespace {
    let someNamespacedExport: number
}

// Экспорт типа
export type SomeType = {
    a: number
}

// Экспорт интерфейса
export interface SomeOtherType {
    b: string
}
```

### Экспорт CommonJS

CommonJS де-факто был стандартом модулей до появления ES2015 и на момент написания книги все еще является стандартом для NodeJS, но его синтаксис несколько отличается от синтаксиса ES2015:

```
declare let defaultExport: SomeType
export = defaultExport
```

Обратите внимание, что мы присвоили экспорты к `export`, а не использовали export в роли модификатора (как делали в ES2015).

Декларация типов для стороннего модуля CommonJS может содержать только один экспорт. Для экспорта нескольких элементов мы используем слияние деклараций (приложение В).

Например, чтобы типизировать несколько экспортов, а не экспорт по умолчанию, мы экспортируем одно `namespace`:

```
declare namespace MyNamedExports {
    export let someExport: SomeType
    export type SomeType = number
    export class OtherExport {
        otherType: string
    }
}
export = MyNamedExports
```

А что насчет модуля CommonJS, который имеет и экспорт по умолчанию, и именованный экспорт? Для него мы используем слияние деклараций:

```
declare namespace MyExports {
    export let someExport: SomeType
    export type SomeType = number
}
declare function MyExports(a: number): string
export = MyExports
```

### Экспорт UMD

Типизация модуля UMD почти идентична типизации модуля ES2015. Единственное различие в том, что, если вы хотите сделать модуль доступным глобально для файлов с режимом скриптов (см. подраздел «Режим

модулей против режима скриптов» на с. 273), то используете специальный синтаксис `export as namespsce`. Например:

```
// Экспорт по умолчанию
declare let defaultExport: SomeType
export default defaultExport

// Именованный экспорт
export class SomeExport {
    a: SomeType
}

// Экспорт типа
export type SomeType =
    { a: number
}

export as namespace MyModule
```

Обратите внимание на последнюю строку: если у вас в проекте есть файл с режимом скриптов, то вы можете использовать модуль напрямую (не импортируя его) в глобальном пространстве имен `MyModule`:

```
let a = new MyModule.SomeExport
```

# Расширение модуля

Расширение декларации типов модуля менее распространено, чем его типизация, но вы можете с ним столкнуться при написании плагина jQuery или примеси Lodash. Старайтесь по возможности его избегать и лучше рассмотрите использование отдельного модуля. То есть вместо примеси Lodash используйте обычную функцию, а вместо плагина jQuery... стоп, а почему вы до сих пор используете jQuery?

## Глобальные объекты

Если вы хотите расширить глобальное пространство имен или интерфейс другого модуля, просто создайте файл с режимом скриптов (см. подраздел «Режим модулей против режима скриптов» на с. 273) и дополните его. Обратите внимание, что это работает только для интерфейсов и пространств имен, ведь TypeScript заботится об их слиянии сам.

Например, добавим в jQuery новый замечательный метод marquee. Начнем с установки самого jquery:

```
npm install jquery --save
npm install @types/jquery --save-dev
```

Затем создадим в проекте новый файл, скажем jquery-extensions.d.ts, и добавим marquee в глобальный интерфейс jQuery (я узнал, что jQuery определяет свои методы в интерфейсе jQuery, разыскивая для него декларации типов):

```
interface JQuery {
    marquee(speed: number): JQuery<HTMLElement>
}
```

Теперь в любом файле, применяющем jQuery, мы можем использовать marquee (но понадобится добавить для `marquee` реализацию среды выполнения):

```
import $ from 'jquery'
$(myElement).marquee(3)
```

Заметьте, что эту же технику мы использовали для расширения встроенных глобальных объектов в разделе «Безопасное расширение прототипа» на с. 193.

## Модули

Расширение экспортов модулей имеет больше подвохов: необходимо корректно типизировать это расширение, загружать модули в правильном порядке в среде выполнения и обязательно обновлять типы расширений, когда структура деклараций типов для расширяемого модуля изменится.

В качестве примера типизируем новый экспорт для React. Начнем с установки React и его деклараций типов:

```
npm install react --save
npm install @types/react --save-dev
```

Затем воспользуемся слиянием модулей (см. раздел «Слияние деклараций» на с. 279) и просто объявим модуль с таким же именем, что и у нашего модуля React:

```
import {ReactNode} from 'react'

declare module 'react' {
    export function inspect(element: ReactNode): void
}
```

Обратите внимание, что, в отличие от предыдущего примера о расширении глобальных объектов, неважно, где находится файл расширения: в режиме скриптов или в режиме модулей.

А что насчет расширения конкретного экспорта из модуля? В духе ReasonReact (https://reasonml.github.io/reason-react/) предположим, что мы хотим добавить встроенный редуктор (reducer) для компонентов React (редуктор — это способ объявления явного набора переходов состояний для компонента React). На момент написания книги декларации типов React объявляют тип `React.Component` как интерфейс и класс, которые сливаются в один экспорт UMD:

```
export = React
export as namespace React

declare namespace React {
    interface Component<P = {}, S = {}, SS = any>
        extends ComponentLifecycle<P, S, SS> {}
class Component<P, S> {
        constructor(props: Readonly<P>)
        // ...
    }
    // ...
}
```

Расширим `Component` методом `reducer`. Для этого в файле react-extensions.d.ts в корне проекта укажем следующее:

```
import 'react' ❶

declare module 'react' { ❷
    interface Component<P, S> { ❸
        reducer(action: object, state: S): S ❹
    }
}
```

❶ Импортируем `'react'`, переключая тем самым наш файл расширения в режим скриптов для потребления модуля `React`. Обратите внимание, что есть и другие способы переключения в режим скриптов. Например, импорт или экспорт чего-либо или экспорт пустого объекта (`export {}`).

❷ Объявляем модуль `'react'`, указывая TypeScript, что мы хотим объявить типы именно для этого пути `import`. Мы уже установили `@types/react` (и определили экспорт для точно такого же пути `'react'`), поэтому TypeScript произведет слияние декларации модуля с декларацией, предоставленной `@types/react`.

❸ Расширяем интерфейс `Component`, предоставленный `React`, объявляя наш интерфейс `Component`. Согласно правилам слияния интерфейсов (см. раздел «Слияние деклараций» на с. 279) наша сигнатура декларации должна в точности совпадать с сигнатурой в `@types/react`.

❹ В завершение объявляем метод `reducer`.

После объявления этих типов (и реализации всего поведения среды выполнения для поддержки этого обновления где-либо еще) мы можем объявить компоненты React со встроенными методами `reducer` типобезопасным способом:

```
import * as React from 'react'

type Props = {
    // ...
}

type State = {
    count: number
    item: string
}

type Action =
    | {type: 'SET_ITEM', value: string}
    | {type: 'INCREMENT_COUNT'}
    | {type: 'DECREMENT_COUNT'}

class ShoppingBasket extends React.Component<Props, State> {
    reducer(action: Action, state: State): State {
```

```
        switch (action.type) {
            case 'SET_ITEM':
                return {...state, item: action.value}
            case 'INCREMENT_COUNT':
                return {...state, count: state.count + 1}
            case 'DECREMENT_COUNT':
                return {...state, count: state.count - 1}
        }
    }
}
```

Как было отмечено в начале раздела, лучше избегать этого паттерна (хоть он и крутой), потому что он может сделать модули нестабильными и зависящими от порядка загрузки. Вместо него попробуйте использовать объединение, чтобы расширения модуля потребляли его и экспортировали обертку без изменения самого модуля.

ПРИЛОЖЕНИЕ Д

# Директивы с тремя слешами

Директивы с тремя слешами — это обычные комментарии JavaScript, к которым TypeScript обращается, чтобы, например, установить настройки компилятора для конкретного файла или определить, что файл зависит от другого файла. Размещайте директивы в верхней части файла перед кодом. Они выглядят так (начинаются с `///` и сопровождаются XML-тегом):

```
/// <директива attr="value" />
```

TypeScript поддерживает несколько таких директив. Таблица Д.1 содержит список самых полезных из них:

`amd-module`

Подробности — в подразделе «Директива amd-module» на с. 324.

`Lib`

Директива `lib` — это способ указать TypeScript, от каких библиотек зависит ваш модуль. Это может понадобиться, если в проекте нет tsconfig.json. Объявление `lib`, от которых вы зависите, в tsconfig.json практически всегда будет лучшим решением.

`path`

При использовании TSC-опции `outFile` директива `path` позволяет объявить зависимые файлы, чтобы при компиляции отодвинуть их назад. Эта директива не нужна, если проект использует `import` и `export`.

`type`

Подробности — в подразделе «Директива types» на с. 323.

**Таблица Д.1.** Директивы с тремя слешами

| Директива | Синтаксис | Используйте, чтобы … |
|---|---|---|
| `amd-module` | `<amd-module name="MyComponent" />` | Объявить имена экспорта при компиляции в модули AMD |
| `lib` | `<reference lib="dom" />` | Объявить, от каких встроенных в TS lib зависят ваши декларации типов |
| `path` | `<reference path="./path.ts" />` | Объявить, от каких файлов TS зависит ваш модуль |
| `type` | `<reference types="./path.d.ts" />` | Объявить, от каких файлов деклараций типов зависит ваш модуль |

# Внутренние директивы

Вы наверняка никогда не станете использовать в своем коде директиву no-default-lib (табл. Д.2).

**Таблица Д.2.** Внутренние директивы с тремя слешами

| Директива | Синтаксис | Используйте, чтобы… |
|---|---|---|
| `no-default-lib` | `<reference no-default-lib="true" />` | Указать TypeScript не использовать никакие `lib` для этого файла |

# Нежелательные директивы

Директиву `amd-dependency` (табл. Д.3) вообще не стоит использовать, а вместо нее нужно придерживаться обычного `import`.

**Таблица. Д.3.** Нежелательные директивы с тремя слешами

| Директива | Синтаксис | Альтернатива |
|---|---|---|
| `amd-dependency` | `<amd-dependency path="./a.ts" name="MyComponent" />` | `import` |

ПРИЛОЖЕНИЕ E

# Флаги безопасности компилятора TSC

Полный список доступных флагов можно посмотреть на справочном сайте TypeScript (https://www.typescriptlang.org/docs/handbook/compiler-options.html).

Каждый выпуск TypeScript предоставляет все новые виды проверок, которые вы можете использовать, чтобы добиться еще большей безопасности кода. Некоторые из флагов, имеющих префикс `strict`, задействуются как часть основного флага `strict`. Вы также можете применять их по одному. Таблица E.1 содержит список флагов безопасности, актуальных на момент написания книги.

**Таблица E.1.** Флаги безопасности TSC

| Флаг | Опция |
|---|---|
| `alwaysStrict` | Генерирует `'use strict'` |
| `noEmitOnError` | Не генерирует JavaScript, если код содержит ошибки |
| `noFallthroughCasesInSwitch` | Убеждается, что каждый случай `switch` возвращает значение или проваливается |
| `noImplicitAny` | Выдает ошибку, когда тип переменной выведен как `any`. |
| `noImplicitReturns` | Убеждается, что каждый путь каждой функции делает возврат явно (см. раздел «Тотальность», с. 165) |
| `noImplicitThis` | Выдает ошибку при использовании this в функции без явного аннотирования типа `this` (см. подраздел «Типизация this», с. 351) |
| `noUnusedLocals` | Предупреждает о неиспользованных локальных переменных |
| `noUnusedParameters` | Предупреждает о неиспользуемых параметрах функции. Для игнорирования этой функции добавьте к параметрам префикс _ |

| Флаг | Опция |
|---|---|
| `strictBindCallApply` | Убеждается в безопасности `bind`, `call`, и `apply` (см. подраздел «Методы call, apply и bind», с. 72) |
| `strictFunctionTypes` | Убеждается, что функции контрвариантны в их параметрах и типах `this` (см. подраздел «Вариантность функций», с. 151) |
| `strictNullChecks` | Проверяет тип на `null` (см. подраздел «null, undefined, void и never», с. 57) |
| `strictPropertyInitialization` | Убеждается, что свойства класса либо допускают `null`, либо инициализированы (см. главу 5) |

ПРИЛОЖЕНИЕ Ж

# TSX

Внутренняя структура TypeScript имеет несколько обработчиков для типизации TSX подключаемым способом. Ими являются специальные типы в глобальном пространстве имен global.JSX, используемом TypeScript как источник данных для типов TSX по всей программе.

Если вы используете только React, то вам не нужно знать об этих низкоуровневых обработчиках, но если вы пишете TypeScript-библиотеку, использующую TSX без React, это приложение предоставит вам о них полезную информацию.

TSX поддерживает два вида элементов: встроенные (неотъемлемые элементы) и пользовательские (основанные на значениях). Неотъемлемые элементы имеют имена в нижнем регистре и относятся к встроенным вроде `<li>`, `<hl>` и `<div>`. Элементы, основанные на значениях, имеют имена в PascalCase (все слова имени начинаются с прописных букв) и относятся к тем, которые вы создаете посредством React (или другого фронтенд-фреймворка, с которым используете TSX). Они могут быть определены как функции или классы (рис. Ж.1).


**Рис. Ж.1.** Виды элементов TSX

На примере деклараций типов React (https://github.com/DefinitelyTyped/DefinitelyTyped/blob/48cb1f5/types/react/index.d.ts) мы рассмотрим обработчики, используемые TypeScript для безопасной типизации TSX. Вот как React подключается в TSX для безопасной типизации JSX:

```
declare global {
    namespace JSX {
        interface Element extends React.ReactElement<any> {} ❶
        interface ElementClass extends React.Component<any> { ❷
            render(): React.ReactNode
        }
        interface ElementAttributesProperty { ❸
            props: {}
        }
        interface ElementChildrenAttribute { ❹
            children: {}
        }

        type LibraryManagedAttributes<C, P> = // ... ❺

        interface IntrinsicAttributes extends React.Attributes {} ❻
        interface IntrinsicClassAttributes<T>
                    extends React.ClassAttributes<T> {} ❼

        interface IntrinsicElements { ❽
            a: React.DetailedHTMLProps<
                React.AnchorHTMLAttributes<HTMLAnchorElement>,
                HTMLAnchorElement
            >
            abbr: React.DetailedHTMLProps<
                React.HTMLAttributes<HTMLElement>,
                HTMLElement
            >
            address: React.DetailedHTMLProps<
                React.HTMLAttributes<HTMLElement>,
                HTMLElement
            >
            // ...
        }
    }
}
```

1. JSX.Element — это тип TSX-элемента, основанного на значении.
2. JSX.ElementClass — это тип экземпляра компонента класса, основанного на значении. Когда бы вы ни объявляли компонент класса, который собираетесь инстанцировать с помощью TSX-синтаксиса `<MyComponent />`, его класс должен соответствовать этому интерфейсу.
3. JSX.ElementAttributesProperty — это имя свойства, по которому TypeScript выясняет, какие атрибуты поддерживает компонент. В случае с React подразумевается свойство `props`. TypeScript ищет его значение в экземпляре класса.
4. JSX.ElementChildrenAttribute — это имя свойства, по которому TypeScript определяет, какие типы потомков поддерживает компонент. Для React это свойство `children`.
5. JSX.IntrinsicAttributes — это набор атрибутов, которые поддерживают все неотъемлемые элементы. Для React это атрибут `key`.
6. JSX.IntrinsicClassAttributes — это набор атрибутов, поддерживаемых всеми компонентами класса (и неотъемлемыми, и основанными на значениях). Для React это `ref`.
7. JSX.LibraryManagedAttributes определяет другие места, где элементы JSX могут объявлять и инициализировать типы свойств. Для React это propTypes (другое место для объявления типов свойств) и defaultProps (место для объявления значений по умолчанию для свойств).
8. JSX.IntrinsicElements перечисляет каждый тип HTML-элемента, который вы можете использовать в TSX, отображая имя тега каждого элемента в типы его атрибутов и потомков. JSX не относится к HTML, поэтому декларации типов React должны сообщать TypeScript, какие конкретно элементы кто-либо может использовать в TSX-выражении. Поскольку вы можете использовать любой стандартный HTML-элемент из TSX, декларации должны сами перечислять каждый элемент вместе с типами его атрибутов (к примеру, для тега `<a>` допустимые атрибуты включают `href: string` и `rel: string`, но не `value`), а также какие типы потомков он может иметь.

Объявляя любой из этих типов в глобальном пространстве JSX, вы можете воспользоваться проверкой типов TypeScript для TSX и кастомизировать ее нужным вам способом. Вы, скорее всего, станете использовать описанные обработчики, только если столкнетесь с написанием библиотеки, использующей TSX, но не React.

# Об авторе

**Борис Черный** — главный инженер и Product Leader в Facebook. Ранее работал в VC, AdTech и во множестве стартапов, большинство из которых ныне не существует. Интересуется языками программирования, синтезом кода и статическим анализом, а также стремится делиться с пользователями своим опытом работы с цифровыми продуктами. В свободное время организует клубные встречи TypeScript в Сан-Франциско и ведет личный блог — performancejs.com. Вы можете найти аккаунт Бориса на GitHub по ссылке https://github.com/bcherny.

# Об обложке

На обложке книги «Профессиональный TypeScript. Разработка масштабируемых JavaScript-приложений» изображены гуанако (ламы гуанако). Эти животные являются дикими предками лам и также относятся к верблюдам. До того как на континенте появились овцы, гуанако можно было встретить в Южной Америке повсеместно. Они обитают в сухих горных регионах на высоте около 4 км над уровнем моря. На сегодняшний день бóльшая часть их численности, составляющей около шестисот тысяч особей, проживает в Аргентине, в регионе Патагонии.

Выживать на скалистых косогорьях гуанако помогают по два мягких пальца на каждой ноге, а также низко расположенный центр тяжести. Средний рост этого животного составляет чуть более метра. Его толстая шерсть может быть короткой или средней, имеет красновато-коричневый оттенок, на животе практически белая. Длинные ресницы защищают глаза от сильных ветров, а большие заостренные уши помогают услышать угрозу на большом расстоянии.

Гуанако перемещаются стадами, состоящими из нескольких самок, молодых особей, не достигших одного года, и племенного самца. При опасности гуанако сигнализируют стаду о необходимости спасаться бегством, издавая пронзительное блеяние, похожее на лающий смех. На них охотятся пумы и лисицы, поэтому ламы вынуждены поочередно дежурить на возвышенностях. Средняя скорость бегущего гуанако — примерно 60 км/час. Если хищник погонится за стадом, самец побежит назад, чтобы защитить его.

Период вынашивания детенышей гуанако длится примерно год. С момента рождения так называемым чуленго требуется всего пять минут, чтобы начать ходить. Спустя год жизни рядом с родителями они должны искать собственные стада. Раннее изгнание сказывается на выживаемости молодых особей — взрослого возраста достигает всего около 30 % гуанако. При благоприятных условиях гуанако может прожить 15–20 лет.

В их среде обитания жесткие и толстые травы являются жизненно важным источником влаги. Гуанако имеют трехкамерные животы, которые

помогают им переваривать растения и долго сохранять жидкость. Их верхние губы разделены надвое для облегчения захвата пищи.

Многие животные, изображенные на обложках книг издательства O'Reilly, находятся под угрозой вымирания; все они очень важны для биосферы. Чтобы узнать, чем вы можете им помочь, посетите сайт animals.oreilly.com.

Изображение на обложке выполнено Карен Монтгомери на основе черно-белой гравюры из Музея изобразительных искусств.

*Борис Черный*

# Профессиональный TypeScript. Разработка масштабируемых JavaScript-приложений

*Перевел с английского Д. Акуратер*


| | |
|---|---|
| Заведующая редакцией | *Ю. Сергиенко* |
| Ведущий редактор | *К. Тульцева* |
| Технический редактор | *М. Кольцов* |
| Литературный редактор | *А. Руденко* |
| Обложка | *В. Мостипан* |
| Корректоры | *М. Одинокова, Е. Павлович* |
| Верстка | *Е. Неволайнен* |

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: 08.2020. 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 21.08.20. Формат 70x100/16. Бумага офсетная. Усл. п. л. 28,380. Тираж 1000. Заказ