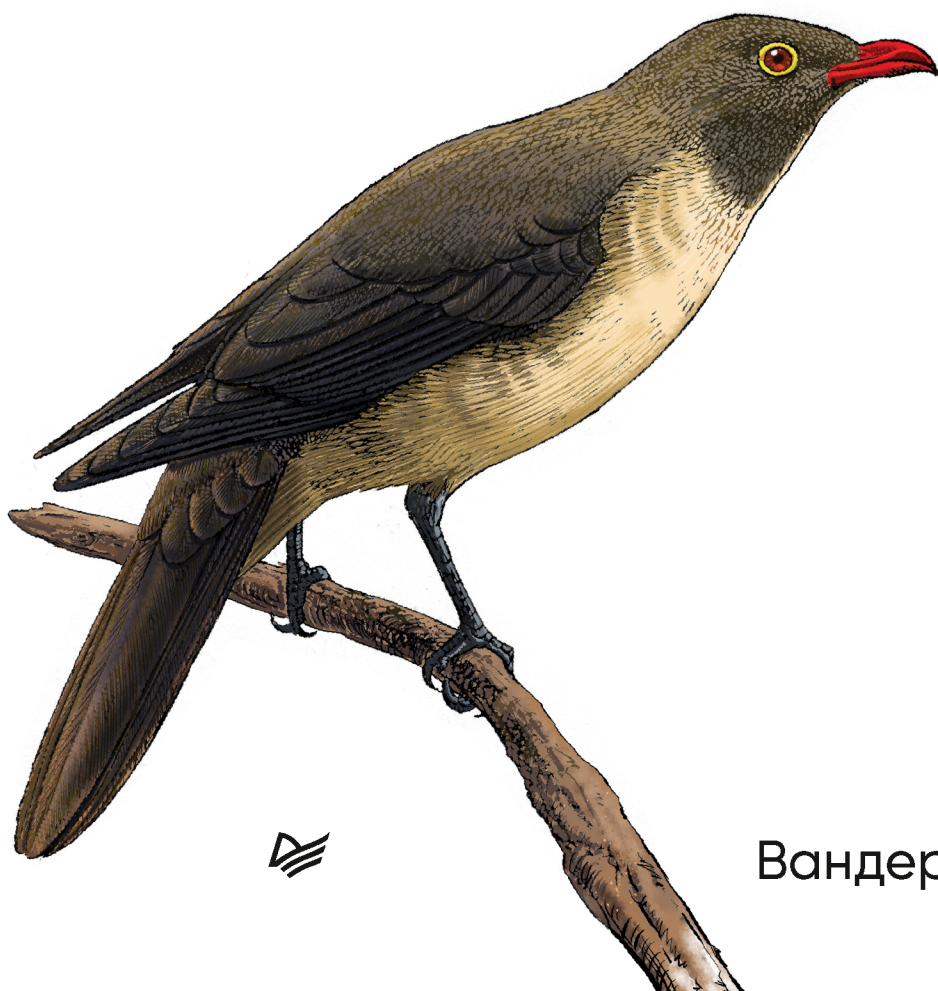


O'REILLY®

Эффективный TypeScript

62 способа улучшить код



Дэн
Вандеркам

Effective TypeScript

62 Specific Ways to Improve Your TypeScript

Dan Vanderkam

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Эффективный TypeScript

62 способа улучшить код

Дэн Вандеркам



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

2020

ББК 32.988.02-018
УДК 004.738.5
В17

Вандеркам Дэн

В17 Эффективный TypeScript: 62 способа улучшить код. — СПб.: Питер, 2020. — 288 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1623-2

«Эффективный TypeScript» необходим тем, кто уже имеет опыт работы с JavaScript. Цель этой книги — не научить пользоваться инструментами, а помочь повысить профессиональный уровень.

TypeScript представляет собой не просто систему типов, а набор служб языка, удобных в использовании. Он повышает безопасность разработки в JavaScript, делает работу увлекательнее и проще.

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

ББК 32.988.02-018
УДК 004.738.5

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

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

ISBN 978-1492053743 англ.

Authorized Russian translation of the English edition of Effective TypeScript
ISBN 9781492053743 © 2020 Dan Vanderkam

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 978-5-4461-1623-2

© Перевод на русский язык ООО Издательство «Питер», 2020

© Издание на русском языке, оформление ООО Издательство «Питер», 2020

© Серия «Бестселлеры O'Reilly», 2020

Краткое содержание

ОТЗЫВЫ	10
ВВЕДЕНИЕ	12
ГЛАВА 1. ЗНАКОМСТВО С TYPESCRIPT.....	18
ГЛАВА 2. СИСТЕМА ТИПОВ В TYPESCRIPT	44
ГЛАВА 3. ВЫВОД ТИПОВ	109
ГЛАВА 4. ПРОЕКТИРОВАНИЕ ТИПОВ	149
ГЛАВА 5. ЭФФЕКТИВНОЕ ПРИМЕНЕНИЕ ANY	190
ГЛАВА 6. ДЕКЛАРАЦИИ ТИПОВ И @TYPES.....	211
ГЛАВА 7. НАПИСАНИЕ И ЗАПУСК КОДА.....	237
ГЛАВА 8. ПЕРЕНОС ДАННЫХ В TYPESCRIPT	259
ОБ АВТОРЕ	284
ОБ ОБЛОЖКЕ	285

Оглавление

ОТЗЫВЫ	10
ВВЕДЕНИЕ	12
От издательства.....	17
ГЛАВА 1. ЗНАКОМСТВО С TYPESCRIPT.....	18
Правило 1. TypeScript и JavaScript взаимосвязаны	18
Правило 2. Выбирайте нужные опции в TypeScript	25
Правило 3. Генерация кода не зависит от типов	28
Правило 4. Привыкайте к структурной типизации.....	35
Правило 5. Ограничьте применение типов any	40
ГЛАВА 2. СИСТЕМА ТИПОВ В TYPESCRIPT	44
Правило 6. Используйте редактор для работы с системой типов	44
Правило 7. Воспринимайте типы как наборы значений	48
Правило 8. Правильно выражайте отношение символа к пространству типов или пространству значений	56
Правило 9. Объявление типа лучше его утверждения	62
Правило 10. Избегайте оберточных типов (String, Number, Boolean, Symbol, BigInt).....	66
Правило 11. Проверяйте пределы исключительных свойств типа	69
Правило 12. По возможности применяйте типы ко всему выражению функции	73

Правило 13. Знайте разницу между <code>type</code> и <code>interface</code>	76
Правило 14. Операции типов и обобщения сокращают повторы.....	81
Правило 15. Используйте сигнатуры индексов для динамических данных.....	90
Правило 16. В качестве сигнатур индексов используйте массивы, кортежи и <code>ArrayLike</code> , но не <code>number</code>	94
Правило 17. Используйте <code>readonly</code> против ошибок, связанных с изменяемостью	98
Правило 18. Используйте отображенные типы для синхронизации значений	105
ГЛАВА 3. ВЫВОД ТИПОВ	109
Правило 19. Не засоряйте код ненужными аннотациями типов	109
Правило 20. Для разных типов — разные переменные.....	117
Правило 21. Контролируйте расширение типов.....	119
Правило 22. Старайтесь сужать типы.....	123
Правило 23. Создавайте объекты целиком.....	127
Правило 24. Применяйте псевдонимы согласованно	130
Правило 25. Для асинхронного кода используйте функции <code>async</code> вместо обратных вызовов.....	134
Правило 26. Используйте контекст при выводе типов	139
Правило 27. Используйте функциональные конструкции и библиотеки для содействия движению типов	144
ГЛАВА 4. ПРОЕКТИРОВАНИЕ ТИПОВ	149
Правило 28. Используйте типы, имеющие допустимые состояния.....	149
Правило 29. Будьте либеральны в том, что берете, но консервативны в том, что даете.....	155
Правило 30. Не повторяйте информацию типа в документации.....	159
Правило 31. Смещайте нулевые значения на периферию типов.....	161
Правило 32. Предпочитайте объединения интерфейсов интерфейсам объединений	165

Правило 33. Используйте более точные альтернативы типов <code>string</code>	169
Правило 34. Лучше сделать тип незавершенным, чем ошибочным	174
Правило 35. Генерируйте типы на основе API и спецификаций, а не данных	178
Правило 36. Именуйте типы согласно области их применения	183
Правило 37. Рассмотрите использование маркировок для номинального типизирования	186
ГЛАВА 5. ЭФФЕКТИВНОЕ ПРИМЕНЕНИЕ ANY	190
Правило 38. Используйте максимально узкий диапазон для типов <code>any</code>	190
Правило 39. Используйте более точные варианты <code>any</code>	193
Правило 40. Скрывайте небезопасные утверждения типов в грамотно типизированных функциях	195
Правило 41. Распознавайте изменяющиеся <code>any</code>	197
Правило 42. Используйте <code>unknown</code> вместо <code>any</code> для значений с неизвестным типом	201
Правило 43. Используйте типобезопасные подходы вместо обезьяньего патча	205
Правило 44. Отслеживайте зону охвата типов для сохранения типобезопасности	207
ГЛАВА 6. ДЕКЛАРАЦИИ ТИПОВ И @TYPES	211
Правило 45. Размещайте TypeScript и <code>@types</code> в <code>devDependencies</code>	211
Правило 46. Проверяйте совместимость трех версий, задействованных в декларациях типов	213
Правило 47. Экспортируйте все типы, появляющиеся в публичных API	218
Правило 48. Используйте TSDoc для комментариев в API	219
Правило 49. Определяйте тип <code>this</code> в обратных вызовах	222
Правило 50. Лучше условные типы, чем перегруженные декларации	226
Правило 51. Зеркалируйте типы для разрыва зависимостей	229
Правило 52. Тестируйте типы с осторожностью	231

ГЛАВА 7. НАПИСАНИЕ И ЗАПУСК КОДА.....	237
Правило 53. Используйте возможности ECMAScript, а не TypeScript	237
Правило 54. Проводите итерацию по объектам	243
Правило 55. Иерархия DOM — это важно.....	246
Правило 56. Не полагайтесь на private при скрытии информации.....	251
Правило 57. Используйте карты кода для отладки	254
ГЛАВА 8. ПЕРЕНОС ДАННЫХ В TYPESCRIPT	259
Правило 58. Пишите современный JavaScript	260
Правило 59. Используйте // @ts-check и JSDoc для экспериментов в TypeScript	269
Правило 60. Используйте allowJs для совмещения TypeScript и JavaScript	274
Правило 61. Конвертируйте модуль за модулем, восходя по графу зависимостей	276
Правило 62. Не считайте миграцию завершенной, пока не включите noImplicitAny.....	281
ОБ АВТОРЕ	284
ОБ ОБЛОЖКЕ	285

Отзывы

«“Эффективный TypeScript” рассматривает наиболее распространенные проблемы, с которыми мы сталкиваемся при работе с TypeScript, и дает практические, ориентированные на результаты советы. Книга будет полезна и опытным, и начинающим разработчикам».

Райан Кавано, ведущий инженер по TypeScript в Microsoft

«“Эффективный TypeScript” содержит практические рецепты и должна быть под рукой у каждого профессионального разработчика. Даже если вы думаете, что знаете TypeScript, купите эту книгу — не пожалеете».

Яков Файн, Java-чемпион

«TypeScript захватывает мир разработки... Глубокое понимание TypeScript поможет разработчикам проявить себя, используя мощные функции языка».

*Джейсон Куллиан, соучредитель TypeScript NYC
и бывший специалист по обслуживанию TSLint*

«Эта книга не только о том, что может делать TypeScript, но и о том, почему полезна отдельно взятая функция и где применять шаблоны для достижения наибольшего эффекта. Книга фокусируется на практических советах, которые будут полезны в повседневной работе. При этом в ней достаточно теории, чтобы читатель на глубинном уровне понял, как все работает. Я считаю себя опытным пользователем TypeScript, но узнал много нового из этой книги».

Джесси Халлетт, старший инженер-программист, Originate, Inc.

Посвящается Алекс.

Ты — мой тип.

Введение

Весной 2016 года я навел на своего бывшего коллегу Эвана Мартина, работавшего в офисе Google в Сан-Франциско, и поинтересовался, чем он сейчас занимается. Этот вопрос я задавал ему неоднократно в разные годы и постоянно получал новые и весьма интересные ответы. Он рассказывал об инструментах построения C++, аудиодрайверах Linux, плагинах для Emacs и онлайн-кроссвордах. На этот раз Эван занялся TypeScript и Visual Studio Code.

Он меня удивил. Я слышал о TypeScript, но знал, что это разработка Microsoft, ошибочно думая, что она работает с .NET. Казалось шуткой, что Эван, который всю жизнь пользовался Linux, вдруг оценил Microsoft.

Затем он продемонстрировал мне VS Code и TypeScript в деле: все работало очень быстро, а интеллект кода легко выстраивал модель системы типов. Я много лет аннотировал типы в комментариях JSDoc для Closure Compiler и воспринял TS как реально работающий типизированный JavaScript. Неужели Microsoft разработала кроссплатформенный текстовый редактор на платформе Chromium? Возможно, этот язык с его набором инструментов действительно достоин изучения.

Я не так давно присоединился к Sidewalk Labs и приступил к написанию первого JavaScript-проекта. Основа кода все еще была достаточно мала, и мы с Эваном смогли конвертировать его в TypeScript всего за несколько дней.

С тех пор TypeScript меня зацепил. Он представляет собой не просто систему типов, а целый набор служб языка, удобных в использовании. Поэтому он не только повышает безопасность разработки в JavaScript, но и делает работу увлекательнее.

Кому адресована эта книга

Обычно книги, в названии которых есть слово «эффективный», рассматриваются в качестве второй основной книги по теме. Поэтому «Эффективный TypeScript» окажется максимально полезен тем, кто уже имеет опыт работы с JavaScript и TypeScript. Цель этой книги — не обучать читателей пользоваться инструментами, а помочь им повысить свой профессиональный уровень. Прочитав ее, вы сформируете лучшее представление о работе компонентов TypeScript, сможете избежать многих ловушек и ошибок и развить свои навыки. В то время как справочное руководство покажет пять разных путей применения языка для реализации одной задачи, «эффективная» книга объяснит, какой из этих путей лучше и почему.

В течение последних лет TypeScript развивался очень быстро, но я надеюсь, что сейчас он достаточно стабилен и моя книга еще долго будет актуальной. Все основное внимание в ней сконцентрировано на самом языке, а не на различных фреймворках и прочих инструментах. Вы не встретите здесь примеров использования React или Angular, равно как и пояснений возможного конфигурирования TypeScript для работы с webpack, Babel или rollup, зато обнаружите много универсальных советов.

Почему я написал эту книгу

Когда я начинал работать в Google, мне вручили экземпляр третьего издания «Эффективный C++», и книга показалась мне своеобразной. Ее нельзя было назвать доступной для начинающих или полноценным руководством по языку. Вместо пояснения назначения тех или иных инструментов C++ она обучала эффективно использовать их и включала множество коротких специфичных правил с примерами.

Эффект от ее чтения вместе с ежедневным использованием языка оказался ошеломительным. Я уже работал с C++, но только после знакомства с книгой стал уверенно пользоваться его компонентами. Позднее у меня были похожие опыты с чтением «Эффективный Java» и «Эффективный JavaScript».

Если вам комфортно работать с несколькими языками, разберитесь, чем именно они отличаются, и примите вызов от TypeScript. В процессе написания книги я сам узнал много нового, чего искренне желаю и вам.

Структура книги

Книга представляет собой сборник кратких эссе (правил). Правила объединены в тематические разделы (главы), к которым можно обращаться автономно в зависимости от интересующего вопроса.

Заголовок каждого правила содержит совет, поэтому ознакомьтесь с оглавлением. Если, например, вы пишете документацию и сомневаетесь, надо ли писать информацию типов, обратитесь к оглавлению и правилу 30 («Не повторяйте информацию типа в документации»).

Практически все выводы в книге продемонстрированы на примерах кода. Думаю, вы, как и я, склонны читать технические книги, глядя в примеры и лишь вскользь просматривая текстовую часть. Конечно, я надеюсь, что вы внимательно прочтаете объяснения, но основные моменты я отразил в примерах.

Прочитав каждый совет, вы сможете понять, как именно и почему он поможет вам использовать TypeScript более эффективно. Вы также поймете, если он окажется непригодным в каком-то случае. Мне запомнился пример, приведенный Скоттом Майерсом, автором книги «Эффективный C++»: разработчики ПО для ракет могли пренебречь советом о предупреждении утечки ресурсов, потому что их программы уничтожались при попадании ракеты в цель. Мне неизвестно о существовании ракет с системой управления, написанной на JavaScript, но такое ПО есть на телескопе James Webb. Поэтому будьте осторожны.

Каждое правило заканчивается блоком «Следует запомнить». Бегло просмотрев его, вы сможете составить общее представление о материале и выделить главное. Но я настоятельно рекомендую читать правило полностью.

Условные обозначения в примерах кода

Большинство примеров кода относятся к TypeScript, кроме случаев, где контекст прямо указывает на пример с JSOB, GraphQL и т. д. Большая часть работы с TypeScript подразумевает взаимодействие с редактором и вывод, поэтому я использовал некоторые сокращения.

Большинство редакторов выделяют ошибки волнистым подчеркиванием. Чтобы увидеть полное сообщение об ошибке, нужно навести курсор на подчеркнутый текст. Для отображения ошибок в примерах кода

я делал пометки в виде волнистой линии в комментариях под строкой с ошибкой:

```
let str = 'not a number';
let num: number = str;
// ~~~ Тип 'string' не может быть назначен для типа 'number'.
```

Иногда я редактировал сообщение об ошибке для придания ему ясности и краткости, но никогда не удалял саму ошибку. Если вы скопируете любой из образцов кода в свой редактор, то получите такую же ошибку.

Для привлечения внимания к отсутствию ошибки я использовал такое обозначение: `// ok`:

```
let str = 'not a number';
let num: number = str as any; // ok
```

Наведя курсор на отдельный символ в редакторе, вы должны увидеть, к какому типу он принадлежит. Дополнительно я использовал комментарий, начинающийся со слова «тип».

```
let v = {str: 'hello', num: 42}; // тип { str: string; num: number; }
```

Тип указан для первого символа строки (в данном случае `v`) или для результата вызова функции:

```
'four score'.split(' '); // строковый тип[]
```

Вы увидите такие же типы в своем редакторе. В случае с вызовом функции может потребоваться назначить временную переменную для отображения типа. В некоторых случаях я буду использовать фиктивные значения для отображения типа переменной в отдельной ветви кода:

```
function foo(x: string|string[]) {
  if (Array.isArray(x)) {
    x; // строковый тип []
  } else {
    x; // строковый тип
  }
}
```

Для `x`; слешы добавлены лишь в целях указания типа в каждой части условного выражения. Вам не нужно включать подобные выражения в код. Если из контекста не следует иное, то примеры кода должны проверяться флагом `-strict`. Все примеры были получены в TypeScript 3.5.3.

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, баз данных, типов данных, переменных окружения, инструкций и ключевых слов.

Моноширинный жирный

Обозначает команды или другой текст, который должен вводиться пользователем. Также 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/oreillymedia/Effective_TypeScript.

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 разрабатываете 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

Эта книга появилась благодаря усилиям многих людей. Спасибо Эвану Мартину за подробное знакомство с TypeScript. Доуи Осинге — за установку связи с O'Reilly и поддержку в проекте. Бретту Слаткину — за советы по структуре и подсказку, что среди моих знакомых есть автор «эффективной» книги. Скотту Мейерсу — за внедрение этого формата и за его полезный пост «Эффективные книги» в блоге. Моим научным редакторам: Рiku Баттлайну, Райану Кавано, Борису Черному, Якову Файну, Джейсону Киллиану и Джесси Халлету. Спасибо Джэкобу Баскину и Кэт Буш за замечания по отдельным правилам, а также всем моим коллегам, посвятившим годы работе с TypeScript. Команде TypeScript NYC: Джейсону, Орте и Кириллу, а также всем спикерам. Многие правила появились благодаря обсуждениям на семинаре, а именно правило 37 (Джейсон Киллиан) и правило 51 (Стив Фолкнер). Было еще большое количество постов и обсуждений, многие из которых собраны в ветке [r/typescript](#) на Reddit. Отдельную благодарность хочу выразить разработчикам, предоставившим образцы кода, которые оказались очень полезны для понимания базовых особенностей TypeScript: Андерсу, Дэниэлу, Райану и всей команде TypeScript в Microsoft — спасибо за общение и отзывчивость в решении возникавших проблем. Даже в случае простых недоразумений было очень приятно сообщить баг и видеть, как сам Андерс Хейлсберг моментально его исправляет. В завершение хочу сказать спасибо Алекс за постоянную поддержку в течение всего проекта и понимание моей потребности в работе по утрам, вечерам и выходным.

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.

Знакомство с TypeScript

Эта глава поможет составить общее впечатление о TypeScript, прежде чем мы перейдем к деталям. Что это такое и как его воспринимать? Каким образом он связан с JavaScript? Как он работает с нулевыми типами, функциями `any` и утиной типизацией?

TypeScript — необычный язык. Он не использует интерпретатор (как Ruby или Python) и не проводит компиляцию кода в низкоуровневый язык (как Java или C). Вместо этого он компилирует программу в другой высокоуровневый язык — JavaScript. Приложение запускает именно JavaScript, а не TypeScript. Поэтому их взаимосвязь весьма существенна, но может вызвать путаницу.

Система типов в TypeScript имеет свои особенности, которые будут подробно разобраны. В настоящей главе мы предупредим об основных сюрпризах.

ПРАВИЛО 1. TypeScript и JavaScript взаимосвязаны

Наверняка вы встречали выражение: «TypeScript — это надмножество JavaScript» или «TypeScript — это типизированное расширение JavaScript». Но какая на самом деле между ними взаимосвязь?

TypeScript действительно является надмножеством JavaScript в синтаксическом смысле. Любая программа JavaScript, не имеющая синтаксических ошибок, является и программой TypeScript. Модуль контроля типов в TypeScript отмечает некоторые проблемные места в коде, но все равно его обрабатывает и выдает в виде JavaScript. Эту часть отношений между ними мы ближе рассмотрим в правиле 3.

Файлы TypeScript имеют расширения `.ts` или `.tsx` вместо расширений `.js` и `.jsx`, характерных для JavaScript. Однако переименование файла `main.js` в `main.ts` не изменит код.

Это очень удобно, ведь можно просто продолжить работать с кодом, пользуясь дополнительными преимуществами TypeScript. Это бы не сработало, если бы вы решили переписать код на язык вроде Java. Плавный процесс переноса кода является одной из отличительных особенностей TypeScript (глава 8).

Все программы JS совместимы с TS, но не всегда наоборот. Существуют программы TS, не подходящие к JS. Это обусловлено тем, что TypeScript отличается дополнительным синтаксисом в спецификации типов (правило 53).

К примеру, вот рабочая программа TypeScript:

```
function greet(who: string) {  
  console.log('Hello', who);  
}
```

Если вы ее запустите через программу вроде Node, которая работает с JavaScript, то получите ошибку:

```
function greet(who: string) {  
  ^
```

SyntaxError: Unexpected token :

Содержимое: `string` является аннотацией типа, характерной для TypeScript. Используя ее хоть раз, вы выходите за рамки JavaScript (рис. 1.1).

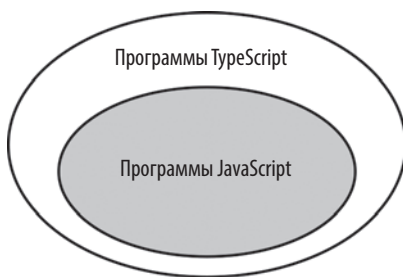


Рис. 1.1. Все программы JavaScript поддерживаются в TypeScript, но не наоборот (диаграмма Венна)

Это не говорит о том, что TS бывает вреден для JS-программ. Рассмотрим следующий код JS:

```
let city = 'new york city';
console.log(city.toUppercase());
```

Программа выдаст ошибку при запуске:

```
TypeError: city.toUppercase is not a function
```

В ней нет аннотаций типов, но модуль проверки типов TS все равно обнаруживает проблему:

```
let city = 'new york city';
console.log(city.toUppercase());
// ~~~~~ Свойство 'toUppercase' не существует в типе
// 'string'. Может, вы имели в виду 'toUpperCase'?
```

В этой ситуации не пришлось сообщать TypeScript, что тип `city` был `string`. Он вывел его, опираясь на исходное значение. Вывод типов является основным занятием TS (глава 3).

Одна из целей системы типов в TypeScript — это поиск кода, который выдаст исключение при выполнении, без его запуска. Поэтому TS называют «статичной» системой типов. Тем не менее модуль проверки типов не всегда способен обнаружить код, приводящий к выводу исключений.

Даже если код не выдает исключение, возможно, он все еще делает не то, что вам нужно. TypeScript старается уловить подобные проблемы.

Например, следующий код JavaScript:

```
const states = [
  {name: 'Alabama', capital: 'Montgomery'},
  {name: 'Alaska', capital: 'Juneau'},
  {name: 'Arizona', capital: 'Phoenix'},
  // ...
];
for (const state of states) {
  console.log(state.capitol);
}
```

Выведет:

```
undefined
undefined
undefined
```

Упс! Что здесь пошло не так? Это рабочая JavaScript-программа (а значит, и TypeScript), и она запустилась без сообщений об ошибках, но выполнила совсем не то, что от нее ожидали. Даже без добавления аннотаций типов модуль их проверки способен обнаружить место ошибки (а также предложить ее решение):

```
for (const state of states) {  
  console.log(state.capitol);  
    // ~~~~~ Свойство 'capitol' не существует в типе  
    //       '{ name: string; capital: string; }'.  
    //       Вы имели в виду 'capital'?  
}
```

Мало того что TypeScript способен отлавливать ошибки даже при отсутствии аннотирования типов, он делает это еще эффективнее, если вы аннотирование проведете. Все потому, что прописанные типы конкретно сообщают TS о ваших *намерениях*, что позволяет ему легче обнаруживать места кода, им не соответствующие. Например, обратная ситуация:

```
const states = [  
  {name: 'Alabama', capitol: 'Montgomery'},  
  {name: 'Alaska', capitol: 'Juneau'},  
  {name: 'Arizona', capitol: 'Phoenix'},  
  // ...  
];  
for (const state of states) {  
  console.log(state.capital);  
    // ~~~~~ Свойство 'capital' не существует в типе  
    //       '{ name: string; capitol: string; }'.  
    //       Вы имели в виду 'capitol'?  
}
```

Та подсказка, которая оказалась столь полезна в первом случае, теперь совершенно неверна. Проблема в том, что вы указали одно свойство в двух разных вариантах, и TypeScript не знает, какой из них верный. Он может угадывать, но не всегда правильно. Решением станет прояснение ваших намерений однозначным объявлением типов `states`:

```
interface State {  
  name: string;  
  capital: string;  
}  
const states: State[] = [  
  {name: 'Alabama', capitol: 'Montgomery'},  
    // ~~~~~
```

```

{name: 'Alaska', capitol: 'Juneau'},
    // ~~~~~
{name: 'Arizona', capitol: 'Phoenix'},
    // ~~~~~Объектный литерал может
    //         определять только известные свойства,
    // но `capitol` не существует в типе `State`.
    // Возможно, вы хотели написать `capital`?
// ...
];
for (const state of states) {
    console.log(state.capitol);
}

```

Теперь ошибка соответствует проблеме, а предложенное решение корректно. Прописывая свое намерение, вы помогаете TypeScript обнаружить и другие потенциальные проблемы. Например, если написать `capitol` только в одной части массива, то в первом примере ошибки бы не возникло. Но уже при аннотированном типе она обнаружится:

```

const states: State[] = [
    {name: 'Alabama', capital: 'Montgomery'},
    {name: 'Alaska', capitol: 'Juneau'},
        // ~~~~~ Вы имели в виду 'capital'?
    {name: 'Arizona', capital: 'Phoenix'},
    // ...
];

```

Таким образом, мы можем добавить на диаграмму Венна еще одну группу: программы TypeScript, прошедшие проверку типов (рис. 1.2).

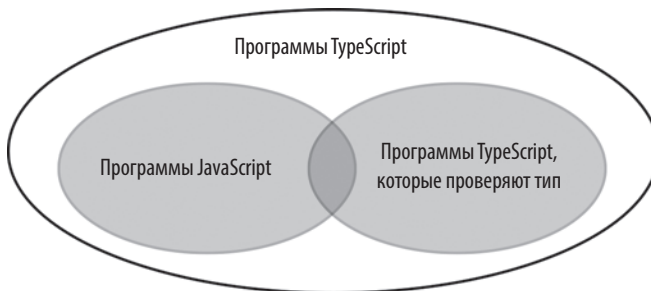


Рис. 1.2. Все программы JavaScript являются TypeScript-программами, но лишь некоторые программы JavaScript (и TypeScript) проходят проверку типов

Да, TypeScript — это надмножество JavaScript, и конечно, вы захотите, чтобы код прошел все проверки типов.

Система типов TypeScript *моделирует* процесс выполнения JavaScript. Вы удивитесь такому моделированию, если привыкли к языку с более строгими проверками выполнения. Например:

```
const x = 2 + '3'; // ok, строковый тип
const y = '2' + 3; // ok, строковый тип
```

Оба этих выражения пройдут проверку типов, даже несмотря на то что они сомнительны и вызвали бы ошибки выполнения во многих других языках. Но в данном случае мы просто определяем выполнение JavaScript и на выходе получим результат обоих выражений в виде строки "23".

Тем не менее TypeScript имеет границы дозволенного. Модуль проверки типов указывает на проблемы в приведенных выражениях, даже несмотря на то что они не выдают исключения во время выполнения:

```
const a = null + 7; // вычисляется как 7 в JS
    // ~~~~~ Оператор '+' не может быть применен к типам ...
const b = [] + 12; // вычисляется как '12' в JS
    // ~~~~~ Оператор '+' не может быть применен к типам ...
alert('Hello', 'TypeScript'); // alerts "Hello".
    // ~~~~~ Ожидается аргумент 0-1, но получен 2.
```

Основная задача системы типов в TypeScript заключается в моделировании поведения JavaScript при выполнении. Причем TypeScript склонен воспринимать непонятные элементы как ошибки, а не как замысел разработчика, то есть идет дальше простого моделирования выполнения. В случае с `capitol` программа не выдавала ошибку (возвращая `undefined`), но модуль проверки типов все же ее отмечал.

Как же TypeScript принимает решения? Доверьтесь команде его разработчиков. Вам нравится складывать `null` и 7 или `[]` и 12? А может быть, вызывать функции с лишними аргументами? TypeScript это вряд ли поддержит.

Может ли быть, что программа, прошедшая проверку типов, все равно выдаст ошибку при выполнении? Да, и вот пример:

```
const names = ['Alice', 'Bob'];
console.log(names[2].toUpperCase());
```

При запуске она выдаст:

```
TypeError: Cannot read property 'toUpperCase' of undefined
```

Доступ к массиву оказался за пределами, ожидаемыми TypeScript. Результатом стало исключение.

Ошибки также ускользают от обнаружения, когда вы используете тип `any`, который мы подробно рассмотрим в правиле 5 и еще более углубленно изучим в главе 5.

Основная причина исключений кроется в том, что понятный TypeScript тип значения расходится с реальным. Система типов TypeScript не гарантирует точности своих статичных типов. Если таковая точность для вас важна, обратите внимание на другие языки вроде Reason или Elm, но за их повышенный уровень безопасного выполнения придется заплатить повышенной сложностью перехода с JavaScript.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ TypeScript — это надмножество JavaScript. Проще говоря, все программы JavaScript по умолчанию являются программами TypeScript. Однако в TS имеется некоторый специфичный синтаксис, не позволяющий некоторым его программам быть совместимыми с JavaScript.
- ✓ TypeScript имеет систему типов, моделирующую выполнение JavaScript, а также обнаруживающую код, который при выполнении выдаст исключение. Гарантий полного представления исключений нет: возможны случаи, когда код проходит проверку типов, но при выполнении выдает ошибку.
- ✓ Несмотря на то что TypeScript моделирует поведение JavaScript, существуют конструкции, которые JS допускает, а TS нет. К их числу относятся вызовы функций с неверными числами или аргументами. Их использование, в большинстве случаев, — дело вкуса.

ПРАВИЛО 2. Выбирайте нужные опции в TypeScript

Пройдет ли этот код проверку типов?

```
function add(a, b) {  
    return a + b;  
}  
add(10, null);
```

Все будет зависеть от того, какие именно опции вы используете. На момент написания книги их в компиляторе TypeScript насчитывается около ста.

Настраиваются они через командную строку:

```
$ tsc --noImplicitAny program.ts
```

А также через файл конфигурации `tsconfig.json`:

```
{  
  "compilerOptions": {  
    "noImplicitAny": true  
  }  
}
```

Рекомендую использовать файл конфигурации, чтобы коллеги, а также вспомогательные программы понимали ваши намерения относительно TypeScript. Для создания файла запустите `tsc - - init`.

Многие из опций конфигурации определяют, откуда брать исходные файлы и какой результат нужно генерировать на выходе. Несколько отдельных элементов контролируют основные аспекты самого языка. Они касаются высокоуровневой архитектуры, настройка которой в большинстве языков не доверяется пользователям. В зависимости от опций TypeScript может работать и восприниматься разработчиком совершенно по-разному. Наиболее важные из них: **noImplicitAny** и **strictNullChecks**.

noImplicitAny определяет, в каких случаях переменные должны иметь явные типы. Следующий код будет работать, если значение `noImplicitAny` будет `off`:

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

Если вы наведете курсор на элемент `add`, то увидите тип, который TypeScript вывел для этой функции:

```
function add(a: any, b: any): any
```

Типы `any` отключают проверку типов для кода, содержащего эти параметры. Они являются полезным инструментом, но использовать их стоит осторожно (правило 5, глава 5).

Назовем их *неявными any*, потому что вы не писали слово «`any`», но все равно работаете с ними. В следующем варианте возникнут ошибки, если включить опцию `noImplicitAny`:

```
function add(a, b) {  
    // ~ Параметр 'a' неявно имеет тип 'any'.  
    // ~ Параметр 'b' неявно имеет тип 'any'.  
    return a + b;  
}
```

Ошибки можно устранить неявно, указав тип `any` или более определенный тип:

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

TypeScript максимально эффективен, когда располагает информацией о типах, поэтому стоит использовать опцию `noImplicitAny` везде, где только возможно. Как только вы привыкнете к тому, что все переменные имеют типы, TypeScript с выключенной `noImplicitAny` покажется вам совершенно другим языком.

Каждый новый проект следует начинать, включив эту опцию, чтобы прописывать типы в процессе написания кода. Это поможет TypeScript обнаруживать проблемы, улучшит читаемость кода, а также повысит ваш уровень как разработчика (правило 6). Отключение такой опции уместно только при переносе проекта из JavaScript (глава 8).

`strictNullChecks` определяет, когда `null` и `undefined` являются допустимыми значениями для каждого типа.

Следующий код окажется рабочим, если `strictNullChecks` определить как `off`:

```
const x: number = null; // ok, null является допустимым параметром
```

Этот же код выдаст ошибку, если `strictNullChecks` будет `on`:

```
const x: number = null;
// ~ Тип 'null' не может быть назначен для типа 'number'.
```

Такая же ошибка будет появляться при использовании значения `undefined` вместо `null`.

Если вы намерены использовать `null`, то исправить ошибку можно, выразив ваш замысел более конкретно:

```
const x: number | null = null;
```

Если вы не хотите разрешать `null`, отследите, где он обнаружен, и добавьте проверку или утверждение:

```
const el = document.getElementById('status');
el.textContent = 'Ready';
// ~~ Возможно, объект является 'null'.

if (el) {
  el.textContent = 'Ready'; // ok, null был исключен.
}
el!.textContent = 'Ready'; // ok, мы утвердили, что el не является null.
```

Опция `strictNullChecks` чрезвычайно полезна для обнаружения ошибок, связанных со значениями `null` и `undefined`, но она существенно усложняет язык. Рекомендую активировать ее в начале работы над новым проектом. Однако если вы новичок или переносите код из JavaScript, то, возможно, предпочтете оставить ее выключенной. Помните, что нужно включить `noImplicitAny`, прежде чем включать `strictNullChecks`.

Если вы не включите `strictNullChecks`, следите за возможностью ошибки вроде `undefined is not an object`. Активация проверки несет все больше сложностей по мере роста вашего проекта, поэтому не затягивайте с решением о ее включении.

Существует также много других опций, влияющих на семантику языка. К ним относятся `noImplicitThis` и `strictFunctionTypes`. Однако они менее значимы по сравнению с упомянутыми `noImplicitAny` и `strictNullChecks`. Для активации всех проверок нужно включить опцию `strict`, и TypeScript поймает больше ошибок.

Продумывайте использование опций. Если ваш коллега предоставляет вам свой пример TypeScript-кода, а вы не можете обнаружить в нем

те же ошибки, то убедитесь, что ваши настройки компиляторов совпадают.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Компилятор TypeScript имеет опции, влияющие на основные аспекты функционирования языка.
- ✓ Лучше настраивать конфигурацию, используя `tsconfig.json`, а не командную строку.
- ✓ Включите `noImplicitAny`, если только вы не переносите проект с JavaScript.
- ✓ Используйте `strictNullChecks` для избежания ошибок выполнения наподобие `undefined is not an object`.
- ✓ Активированная опция `strict` позволит TypeScript обнаружить больше ошибок.

ПРАВИЛО 3. Генерация кода не зависит от типов

На высоком уровне `tsc` (компилятор) делает две вещи:

- Конвертирует новейшую версию TypeScript / JavaScript в более старую версию JavaScript, способную работать в браузерах (транспилиция).
- Проверяет код на наличие ошибок типов.

Удивительно то, что эти действия не связаны и проверка типов не влияет на работу кода JavaScript.

Это позволяет сделать неожиданные выводы и понять, что именно может или чего не может TypeScript.

Код с ошибками типов может выполнять вывод

Это возможно благодаря тому, что вывод кода независим от проверки типов.

```
$ cat test.ts
let x = 'hello';
x = 1234;
```

```
$ tsc test.ts
test.ts:2:1 - error TS2322: Type '1234' is not assignable to type
'string'.

2 x = 1234;
  ~

$ cat test.js
var x = 'hello';
x = 1234;
```

В языках типа C или Java вывод и проверка типов идут рука об руку. В TypeScript вы можете воспринимать ошибки как предупреждения в упомянутых языках, поскольку выполнение они не останавливают.

КОМПИЛЯЦИЯ И ПРОВЕРКА ТИПОВ

Иногда говорят, что код TypeScript «не компилируется», имея в виду, что в нем есть ошибки. Но это заявление неверно. Компиляция относится лишь к кодогенерации. До тех пор пока ваш TypeScript-код является рабочим (а чаще не является) JavaScript-кодом, компилятор TypeScript будет проводить вывод. В таких случаях лучше просто говорить, что в коде есть ошибки или что он не проходит проверку типов.

Выполнение кода, содержащего ошибки, позволяет лучше в нем ориентироваться. Например, вы разрабатываете веб-приложение и знаете, что в определенных его частях есть проблемы. Благодаря тому что TypeScript будет генерировать код даже при наличии ошибок, вы сможете сперва проверить остальные части приложения, а ошибки исправить после.

Лучше, чтобы в коде полностью отсутствовали ошибки, иначе вам придется запоминать, какие ошибки ожидаемые, а какие неожиданные. Если вы хотите отключить возможность выполнения при наличии ошибок, активируйте опцию `noEmitOnError` в `tsconfig.json` или через вашу систему сборки.

При выполнении проверка типов TypeScript невозможна

Допустим, у вас возникло желание написать код в таком виде:

```
interface Square {
  width: number;
```

```

}
interface Rectangle extends Square {
  height: number;
}
type Shape = Square | Rectangle;

function calculateArea(shape: Shape) {
  if (shape instanceof Rectangle) {
    // ~~~~~ 'Rectangle' относится только к типу,
    //             но здесь использована в качестве значения.
    return shape.width * shape.height;
    // ~~~~~ Свойство 'height' не существует
    //             в типе 'Shape'.
  } else {
    return shape.width * shape.width;
  }
}

```

Проверка `instanceof` происходит при выполнении кода, но тип `Rectangle` на него не влияет. Так как типы TypeScript являются «стираемыми», то при компиляции кода в JavaScript из него удалятся все `interfaces`, `types` и аннотации типов.

Для уточнения типа `shape` потребуется перестроить этот тип в процессе выполнения, например, проверить присутствие свойства `height`:

```

function calculateArea(shape: Shape) {
  if ('height' in shape) {
    shape; // Тип Rectangle
    return shape.width * shape.height;
  } else {
    shape; // Тип Square
    return shape.width * shape.width;
  }
}

```

Проверка свойств относится только к значениям, доступным при выполнении, но в то же время позволяет модулю проверки типов уточнить тип `shape` для `Rectangle`.

Еще можно добавить фрагмент кода, приводящий тип в доступную при выполнении форму:

```

interface Square {
  kind: 'square';
  width: number;
}

```

```

interface Rectangle {
  kind: 'rectangle';
  height: number;
  width: number;
}
type Shape = Square | Rectangle;

function calculateArea(shape: Shape) {
  if (shape.kind === 'rectangle') {
    shape; // Тип Rectangle
    return shape.width * shape.height;
  } else {
    shape; // Тип Square
    return shape.width * shape.width;
  }
}

```

Тип `Shape` здесь является примером размеченного объединения, которое широко применяется в TypeScript, поскольку позволяет легко получать информацию о типе при выполнении.

Некоторые построения используют и типы (недоступные при выполнении), и значения (доступные). Например, создание `Square` и `Rectangle` в качестве классов исправит ошибку:

```

class Square {
  constructor(public width: number) {}
}
class Rectangle extends Square {
  constructor(public width: number, public height: number) {
    super(width);
  }
}
type Shape = Square | Rectangle;

function calculateArea(shape: Shape) {
  if (shape instanceof Rectangle) {
    shape; // Тип Rectangle
    return shape.width * shape.height;
  } else {
    shape; // тип Square
    return shape.width * shape.width; // ok
  }
}

```

Это работает, потому что `class Rectangle` относится и к *типу* (в `type Square | Rectangle`), и к *значению* (`shape instanceof Rectangle`), в то время как

`interface` представляет только тип. Лучше разобраться в этих тонкостях поможет правило 8.

Операции типов не влияют на значения при выполнении

Предположим, у вас есть значение, которое может быть строкой или числом, и вы хотите его нормализовать, чтобы оно всегда было только числом. Вот такой неверный пример пройдет проверку типов:

```
function asNumber(val: number | string): number {  
    return val as number;  
}
```

Взглянув на сгенерированный код JavaScript, становится ясно, что эта функция делает:

```
function asNumber(val) {  
    return val;  
}
```

Здесь не происходит никакого преобразования. Элемент `as number` является операцией и, следовательно, не может повлиять на процесс выполнения кода. Для нормализации значения вам потребуется проверить его тип при выполнении и провести преобразование посредством конструкций JavaScript:

```
function asNumber(val: number | string): number {  
    return typeof(val) === 'string' ? Number(val) : val;  
}
```

Элемент `as number` является *утверждением типа*. Для понимания, где лучше его использовать, смотрите правило 9.

При выполнении типы могут отличаться от объявленных

Может ли приведенная функция провести финальное выполнение `console.log`?

```
function setLightSwitch(value: boolean) {  
    switch (value) {  
        case true:  
            turnLightOn();  
            break;  
    }
```

```

    case false:
        turnLightOff();
        break;
    default:
        console.log(`I'm afraid I can't do that.`);
}
}

```

Обычно TypeScript указывает на мертвый код, но в этом случае он не жалуется даже с включенным режимом `strict`. Как же мы получили такую ветвь?

Необходимо помнить, что `boolean` является *объявленным* типом. Это тип TypeScript, а следовательно, при выполнении удаляется. В коде JavaScript пользователь может по незнанию вызвать `setLightSwitch` со значением наподобие `ON`.

Существуют также способы спровоцировать появление такой ветви и в оригинальном варианте TypeScript. Возможно, функция была вызвана значением, поступившим через сетевой вызов:

```

interface LightApiResponse {
    lightSwitchValue: boolean;
}
async function setLight() {
    const response = await fetch('/light');
    const result: LightApiResponse = await response.json();
    setLightSwitch(result.lightSwitchValue);
}

```

Вы объявили, что результатом запроса `/light` является `LightApiResponse`, но ничто этому не способствует. Если вы неверно поняли API и `lightSwitchValue` на деле является *строкой*, тогда эта строка и будет передана в `setLightSwitch` при выполнении. А возможно, API изменился уже после выполнения вами развертывания.

TypeScript может несколько вводить в заблуждение, когда типы при выполнении не совпадают с объявленными. Таких ситуаций следует избегать. Значения тоже могут получить типы, отличные от объявленных.

Функции, основанные на типах TypeScript, не поддерживают перегрузку

Языки, подобные C++, позволяют определять разные версии функции, отличающиеся только типами их параметров. Это называется перегрузкой функций. Однако раз выполнение кода в TypeScript происходит

без влияния типов, то, вероятно, подобные конструкции в нем недопустимы:

```
function add(a: number, b: number) { return a + b; }  
    // ~~~ Повторение выполнения функции.  
function add(a: string, b: string) { return a + b; }  
    // ~~~ Повторение выполнения функции.
```

На самом же деле TypeScript предоставляет возможность перегрузки функций, но она реализуется только на уровне типов. Вы можете сделать много объявлений функции, но при этом указать лишь один вариант выполнения:

```
function add(a: number, b: number): number;  
function add(a: string, b: string): string;  
  
function add(a, b) {  
    return a + b;  
}  
  
const three = add(1, 2); // Тип является числом  
const twelve = add('1', '2'); // Тип является строкой
```

Два первых объявления `add` только сообщают информацию типа. Когда TypeScript произведет вывод JavaScript-кода, они будут удалены и останется только выполнение. Если вы пользуетесь подобным способом перегрузки, то сперва обратитесь к правилу 50, где описаны несколько тонкостей по этой теме.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Генерация кода происходит независимо от системы типов. Это означает, что типы TypeScript не могут влиять на выполнение кода.
- ✓ Программа, содержащая ошибки типов, все равно может проводить вывод кода (компиляцию).
- ✓ Типы TypeScript недоступны при выполнении, и для обращения к ним необходимо найти способ их перестроить. Обычно это делается с помощью тип-суммы или проверки свойств. Некоторые конструкции, например `class`, представляют одновременно и тип TypeScript, и доступное при выполнении значение.

ПРАВИЛО 4. Привыкайте к структурной типизации

JavaScript имеет неумышленную утиную типизацию: если вы передадите функции значение с верными свойствами, то ее не будет волновать, как вы получили это значение. Она просто его использует. TypeScript моделирует это поведение, что иногда приводит к неожиданным результатам, так как понимание типов модулем проверки может оказаться шире привычного вам. Развитие навыка структурной типизации позволит лучше чувствовать, где действительно есть ошибки, и писать более надежный код.

К примеру, вы работаете с библиотекой физических характеристик и у вас есть тип вектора 2D:

```
interface Vector2D {  
  x: number;  
  y: number;  
}
```

Вы пишете функцию для вычисления его длины:

```
function calculateLength(v: Vector2D) {  
  return Math.sqrt(v.x * v.x + v.y * v.y);  
}
```

и вводите определение вектора `named`:

```
interface NamedVector {  
  name: string;  
  x: number;  
  y: number;  
}
```

Функция `calculateLength` будет работать с `NamedVector`, так как в нем присутствуют свойства `x` и `y`, являющиеся `number`. TypeScript это понимает:

```
const v: NamedVector = { x: 3, y: 4, name: 'Zee' };  
calculateLength(v); // ок, результат равен 5.
```

Интересно то, что вы не объявляли связь между `Vector2D` и `NamedVector`. Вам также не пришлось прописывать альтернативное выполнение `calculateLength` для `NamedVector`. Система типов TypeScript моделирует поведение JavaScript при выполнении (правило 1), что позволило `NamedVector`

вызвать `calculateLength` на основании того, что его *структура* сопоставима с `Vector2D`. Отсюда выражение «структурная типизация».

Но это также может привести и к проблемам. Допустим, вы добавите тип вектора 3D:

```
interface Vector3D {  
  x: number;  
  y: number;  
  z: number;  
}
```

и напишете функцию, чтобы нормализовать векторы (сделать их `length` равной 1):

```
function normalize(v: Vector3D) {  
  const length = calculateLength(v);  
  return {  
    x: v.x / length,  
    y: v.y / length,  
    z: v.z / length,  
  };  
}
```

Если вы вызовете эту функцию, то, вероятнее всего, получите больше, чем единичную длину:

```
> normalize({x: 3, y: 4, z: 5})  
{ x: 0.6, y: 0.8, z: 1 }
```

Что же пошло не так и почему TypeScript не сообщил об ошибке?

Баг заключается в том, что `calculateLength` работает с векторами 2D, а `normalize` — с 3D. Поэтому компонент `z` игнорируется при нормализации.

Может показаться странным, что модуль проверки типов не уловил этого. Почему допускается вызов `calculateLength` 3D-вектором, несмотря на то что ее тип работает с 2D-векторами?

То, что работало хорошо с `named`, здесь привело к обратному результату. Вызов `calculateLength` объектом `{x, y, z}` не выдает ошибку. Следовательно, модуль проверки типов не жалуется, что в итоге приводит к появлению бага. Если вы захотите, чтобы в подобном случае ошибка все же обнаруживалась, обратитесь к правилу 37.

Прописывая функции, легко представить, что они будут вызываться свойствами, которые вы объявили, *и никакими другими*. Это называется «запечатанный», или «точный», тип и не может быть применено в системе типов TypeScript. Нравится вам это или нет, но здесь типы открыты.

Иногда это приводит к сюрпризам:

```
function calculateLengthL1(v: Vector3D) {
  let length = 0;
  for (const axis of Object.keys(v)) {
    const coord = v[axis];
    // ~~~~~ Элемент неявно имеет тип "any", потому что
    // тип "string" не может быть использован
    // для обозначения типа "Vector3D"
    length += Math.abs(coord);
  }
  return length;
}
```

Почему это ошибка? Поскольку `axis` является одним из ключей `v` из `Vector3D`, то он должен быть `x`, `y` или `z`. А согласно изначальному объявлению `Vector3D`, они все являются `numbers`. Следовательно, не должен ли тип `coord` также быть `number`?

Это не ложная ошибка. Мы знаем, что `Vector3D` строго определен и не имеет иных свойств. Хотя мог бы:

```
const vec3D = {x: 3, y: 4, z: 1, address: '123 Broadway'};
calculateLengthL1(vec3D); // ok, возвращает NaN
```

Поскольку `v`, вероятно, мог иметь любые свойства, то тип `axis` является `string`. У TypeScript нет причин считать `v[axis]` только числом. При итерации объектов может быть сложно добиться корректной типизации. Мы вернемся к этой теме в правиле 54, а сейчас обойдемся без циклов:

```
function calculateLengthL1(v: Vector3D) {
  return Math.abs(v.x) + Math.abs(v.y) + Math.abs(v.z);
}
```

Структурная типизация может также служить причиной сюрпризов в классах, которые сравниваются на предмет возможного назначения свойств:

```
class C {
  foo: string;
```

```

    constructor(foo: string) {
        this.foo = foo;
    }
}

const c = new C('instance of C');
const d: C = { foo: 'object literal' }; // ok!

```

Почему `d` может быть назначен для `C`? У него есть свойство `foo`, являющееся `string`. Еще у него есть `constructor` (из `Object.prototype`), который может быть вызван аргументом (хотя обычно он вызывается без него). Итак, структуры совпадают. Это может привести к неожиданностям, если у вас присутствует логика в конструкторе `C` и вы напишете функцию, подразумевающую его запуск. В этом существенное отличие от языков вроде `C++` или `Java`, где объявления параметра типа `C` гарантирует, что он будет принадлежать именно `C` либо его подклассу.

Структурная типизация хорошо помогает при написании тестов. Допустим у вас есть функция, которая выполняет запрос в базу данных и обрабатывает результат.

```

interface Author {
    first: string;
    last: string;
}

function getAuthors(database: PostgresDB): Author[] {
    const authorRows = database.runQuery(`SELECT FIRST, LAST FROM
        AUTHORS`);
    return authorRows.map(row => ({first: row[0], last: row[1]}));
}

```

Чтобы ее протестировать, вы могли бы создать имитацию `PostgresDB`. Однако лучшим решением будет использование структурной типизации и определение более узкого интерфейса:

```

interface DB {
    runQuery: (sql: string) => any[];
}

function getAuthors(database: DB): Author[] {
    const authorRows = database.runQuery(`SELECT FIRST, LAST FROM
        AUTHORS`);
    return authorRows.map(row => ({first: row[0], last: row[1]}));
}

```

Вы по-прежнему можете передать `postgresDB` функции `getAuthors` в вывод, поскольку в ней есть метод `runQuery`. Структурная типизация не обязывает `PostgresDB` сообщать, что она выполняет `DB`. `TypeScript` сам определит это.

При написании тестов вы можете передавать и более простой объект:

```
test('getAuthors', () => {
  const authors = getAuthors({
    runQuery(sql: string) {
      return [['Toni', 'Morrison'], ['Maya', 'Angelou']];
    }
  });
  expect(authors).toEqual([
    {first: 'Toni', last: 'Morrison'},
    {first: 'Maya', last: 'Angelou'}
  ]);
});
```

`TypeScript` определит, что тестовый `DB` соответствует интерфейсу. В то же время ваши тесты совершенно не нуждаются в информации о базе данных вывода: не требуется никаких имитированных библиотек. Введя абстракцию (`DB`), мы освободили логику от деталей выполнения (`PostgresDB`).

Еще одним преимуществом структурной типизации является то, что она способна четко обрывать зависимости между библиотеками. Больше информации по этой теме вы найдете в правиле 51.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ JavaScript применяет утиную типизацию, а `TypeScript` ее моделирует при помощи структурной типизации. В связи с этим значения, присваиваемые вашим интерфейсам, могут иметь свойства, не указанные в объявленных типах. Типы в `TypeScript` не бывают запечатанными.
- ✓ Имейте в виду, что классы также подчиняются правилам структурной типизации. Поэтому вы можете получить не тот образец класса, какой ожидали.
- ✓ Используйте структурную типизацию для упрощения тестирования элементов.

ПРАВИЛО 5. Ограничьте применение типов `any`

Система типов в TypeScript является *постепенной* и *выборочной*. Постепенность проявлена в возможности добавлять типы в код шаг за шагом, а выборочность — в возможности отключения модуля проверки типов, когда вам это нужно. Ключом к управлению в данном случае выступает тип `any`:

```
let age: number;
age = '12';
// ~~~ Тип '"12"' не может быть назначен для типа 'number'.
age = '12' as any; // ok
```

Модуль проверки прав, указывая на ошибку, но ее обнаружение можно исключить, просто добавив `as any`. Когда вы начинаете работать с TypeScript, становится заманчивым использование типов `any` или утверждений `as any` при непонимании ошибки, недоверии к модулю проверки или нежелании тратить время на прописывание типов. Но помните, что `any` нивелирует многие преимущества TypeScript, а именно:

Снижает безопасность кода

В примере выше, согласно объявленному типу, `age` является `number`. Но `any` позволил назначить для него строку. Модуль проверки будет считать, что это число (ведь именно так вы объявили), что приведет к появлению путаницы.

```
age += 1; // ok. При выполнении получится age "121".
```

Позволяет нарушать условия

Создавая функцию, вы ставите условие, что, получив от вызова определенный тип данных, она произведет соответствующий тип при выводе, которое нарушается так:

```
function calculateAge(birthDate: Date): number {
  // ...
}

let birthDate: any = '1990-01-19';
calculateAge(birthDate); // ok
```

Параметр даты рождения `birthDate` должен быть `Date`, а не `string`. Тип `any` позволил нарушить условие, относящееся к `calculateAge`. Это может оказаться особенно проблематичным, так как JavaScript имеет склонность к неявной конвертации типов. Из-за этого `string` в некоторых случаях сработает там, где предполагается `number`, но неизбежно даст сбой в другом месте.

Исключает поддержку языковой службы

Если символу присвоен тип, языковые службы TypeScript способны предоставить соответствующую автоподстановку и контекстную документацию (рис. 1.3).

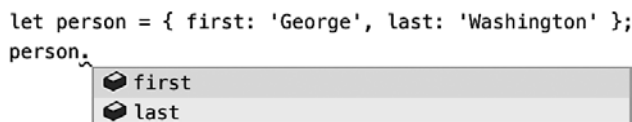


Рис. 1.3. Языковая служба TypeScript способна обеспечить контекстное автозаполнение для символьных типов

Однако, присваивая символам тип `any`, вам придется все делать самостоятельно (рис. 1.4).

```
let person: any = { first: 'George', last: 'Washington' };
person.
```

Рис. 1.4. Нет автозаполнения для свойств символов любых типов

И переименование тоже. Если у вас есть тип `Person` и функции для форматирования имени:

```
interface Person {
  first: string;
  last: string;
}

const formatName = (p: Person) => `${p.first} ${p.last}`;
const formatNameAny = (p: any) => `${p.first} ${p.last}`;
```

тогда вы можете выделить `first` в редакторе и выбрать `Rename Symbol` для его переименования в `firstName` (рис. 1.5 и рис. 1.6).

Это изменит функцию `formatName`, но не в случае с `any`:

```
interface Person {
  first: string;
  last: string;
}

const formatName = (p: Person) => `${p.firstName} ${p.last}`;
const formatNameAny = (p: any) => `${p.first} ${p.last}`;
```

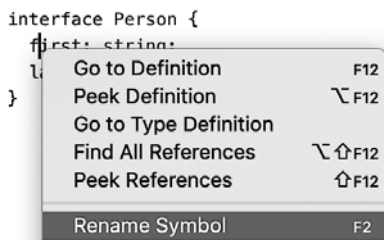


Рис. 1.5. Переименование символа в vscode

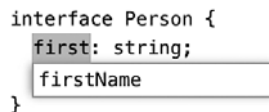


Рис. 1.6. Выбираем новое имя. Служба языка TypeScript гарантирует, что все символы в проекте также будут переименованы

Слоган TypeScript: «JavaScript, который масштабируется». Ключевой составляющей масштабирования выступают языковые службы (правило 6). Утрата возможности их применения ведет к падению производительности, причем не только для вас, но и для тех, кто будет дальше работать с кодом.

Маскирует возможности рефакторинга кода

Предположим, вы создаете веб-приложение, в котором пользователи могут выбирать некий предмет. Один из компонентов должен иметь обратный вызов `onSelectItem`.

Прописывание типа для `Item` кажется проблематичным, поэтому вы решаете использовать `any`:

```
interface ComponentProps {
  onSelectItem: (item: any) => void;
}
```

Вот код, который управляет этим компонентом:

```
function renderSelector(props: ComponentProps) { /* ... */ }

let selectedId: number = 0;
function handleSelectItem(item: any) {
  selectedId = item.id;
}

renderSelector({onSelectItem: handleSelectItem});
```

Позже вы перестраиваете оператор ветвления так, что становится сложнее передать весь объект `item` в `onSelectItem`. Но здесь нет ничего сложного, поскольку вам всего лишь нужен ID. Вы меняете сигнатуру `ComponentProps`:

```
interface ComponentProps {  
  onSelectItem: (id: number) => void;  
}
```

Затем вы обновляете компонент, и вся структура проходит проверку типов. Победа!

...или нет? `handleSelectItem` принимает параметр `any`, и теперь ей подходит `Item` в качестве ID. Это порождает исключение при выполнении, несмотря на успешную проверку типов. Если бы вы использовали определенный тип, то эта ошибка была бы обнаружена модулем проверки.

Скрывает структуру типов

Определение типов для сложных объектов вроде состояний приложений может оказаться действительно долгим. Вместо прописывания типов для десятков свойств состояния страницы вы можете соблазниться использованием типов `any` для скорейшего решения задачи.

И сделаете структуру типов неявной. Как далее поясняется в главе 4, правильная структура типов очень важна для написания чистого, корректного и понятного впоследствии кода.

Подрывает уверенность в системе типов

Когда ошибку ловит модуль проверки, ваша уверенность в системе типов возрастает. Однако обнаружение ошибки типов при выполнении кода наносит по вашей уверенности удар. Типы `any` зачастую оказываются причиной появления непонятных ошибок.

Частое применение типов `any` заставит вас удерживать в уме реальные типы, тогда как TypeScript нацелен проследить за ними вместо вас.

В случаях, когда применение `any` неизбежно, воспользуйтесь советами из главы 5.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Типы `any` отключают реакцию модуля проверки и языковых служб TypeScript. Они могут маскировать реальные проблемы, вредить опыту разработчика и подрывать его уверенность в системе типов. По возможности избегайте их применения.

Система типов в TypeScript

TypeScript генерирует код (правило 3), но основная польза этого языка заключена в системе типов.

Эта глава познакомит вас со всеми основными элементами системы типов и пояснит, как ее использовать, на примере правильных и нежелательных решений. Мощность системы типов в TypeScript превзойдет ваши ожидания. Пункты этой главы дадут вам прочный фундамент для работы с этой книгой.

ПРАВИЛО 6. Используйте редактор для работы с системой типов

После установки TypeScript вы получите два исполняемых файла:

- `tsc` (компилятор);
- `tsserver` (изолированный сервер).

Скорее всего, вы будете запускать непосредственно компилятор, но значение сервера не менее велико, так как он предоставляет *языковые службы*. Они включают в себя автоподстановку, инспекцию, навигацию и рефакторинг. Воспользоваться ими можно через редактор. Если вы не настроите его на предоставление служб, то много потеряете. Службы, такие как автоподстановка, превращают работу с TypeScript в сплошное удовольствие. Настроенный редактор станет хорошей площадкой для расширения и проверки вашего понимания системы типов. Работая со службами, вы начнете ориентироваться, где TypeScript может вводить типы самостоятельно, и сможете писать компактные и идиоматичные коды (правило 19).

Детали в разных редакторах отличаются, но в каждом из них можно увидеть, к какому типу TypeScript относит определенный символ (рис. 2.1).

```
let num: number  
num = 10;
```

Рис. 2.1. Редактор (vscode), показывающий, что предполагаемый тип `num` — число

Вы написали `number`, но TypeScript определил тип по значению `10`.

Вы также можете инспектировать функции (рис. 2.2).

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

Рис. 2.2. Применение редактора для выяснения типа, определенного для функции

Здесь следует обратить внимание на значение, присвоенное возвращаемому типу, — `number`. Если оно не совпадает с предполагаемым вами значением, то следует сделать объявление типа и проследить причину несоответствия (правило 9).

Видение того, как TypeScript понимает типы переменных в любой точке кода, очень важно для расширения (правило 21) и сужения (правило 22) типов. Отслеживая изменение типа переменной в ответвлении условного выражения, вы запомните модели поведения типов — они станут предсказуемыми функциями (рис. 2.3).

Рассмотрите отдельные свойства более крупного объекта, чтобы увидеть, какие выводы для них сделал TypeScript (рис. 2.4).

```
function logMessage(message: string | null) {  
  if (message) {  
    (parameter) message: string  
    message  
  }  
}
```

Рис. 2.3. Тип сообщения (`message`) `string | null` находится вне ветви, но `string` находится внутри

```
const foo = {
  (property) x: number[]
  x: [1, 2, 3],
  bar: {
    name: 'Fred'
  }
};
```

Рис. 2.4. Инспектирование вывода типов в объекте

Если вы хотите, чтобы `x` имел тип `tuple ([number, number, number])`, сделайте аннотацию типа.

Чтобы увидеть выведенные обобщенные типы в середине цепочки операций, инспектируйте имя метода функции (рис. 2.5).

```
function restOfPath(path: string) {
  (method) Array<string>.slice(start?: number, end?: number): string[]
  Returns a section of an array.
  @param start — The beginning of the specified portion of the array.
  @param end — The end of the specified portion of the array.
  return path.split('/').slice(1).join('/');
}
```

Рис. 2.5. Выяснение выведенных обобщенных типов в цепочке вызовов метода

Графа `Array<string>` показывает, что TypeScript понял создание массива строк функцией `split`. Несмотря на некоторую неоднозначность в приведенном примере, эта информация может оказаться очень важной при написании и отладке длинных цепочек вызовов функций.

Обращая внимание на ошибки типов в редакторе, вы изучите тонкости системы типов. Например, эта функция пытается получить `HTMLElement` по его ID или вернуть заданный тип по умолчанию. TypeScript фиксирует две ошибки:

```
function getElement(elOrId: string|HTMLElement|null): HTMLElement {
  if (typeof elOrId === 'object') {
    return elOrId;
  }
  // ~~~~~ 'HTMLElement | null' не может быть назначен
  // для 'HTMLElement'.
```

```

    } else if (elOrId === null) {
        return document.body;
    } else {
        const el = document.getElementById(elOrId);
        return el;
    }
    // ~~~~~ 'HTMLElement | null' не может быть назначен
    //      для 'HTMLElement'.
}

```

В первой ветви оператора `if` было намерение отфильтровать объекты (`object`), а именно `HTMLElement`. Но, как это ни странно, в JavaScript `typeof null` является `object`, поэтому `elOrId` смог иметь значение `null` в этой ветви. Вы можете это исправить, добавив изначальную проверку `null`. Вторая ошибка следует из того, что `document.getElementById` может вернуть `null`, и вам нужно отреагировать на это, например, сгенерировав исключение.

Языковые службы помогают ориентироваться среди библиотек и объявленных типов. Предположим, вы видите вызов функции `fetch` в коде и хотите узнать о нем больше. Редактор должен предоставить для этого опцию «Go to definition» (следовать к определению). В моем редакторе это выглядит так (рис. 2.6).

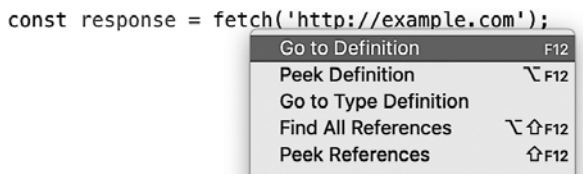


Рис. 2.6. Языковая служба TypeScript предоставляет функцию Go to definition, которая должна отображаться в редакторе

Выбор этой опции направляет в библиотеку деклараций типов `lib.dom.d.ts`, которую TypeScript содержит для DOM (объектной модели документа).

```

declare function fetch
  (input: RequestInfo, init?: RequestInit
): Promise<Response>;

```

Как видите, `fetch` возвращает `Promise` и берет два аргумента. Аналогичный переход в `RequestInfo` перенесет вас сюда:

```

type RequestInfo = Request | string;

```

Отсюда вы уже можете попасть в `Request`:

```
declare var Request: {  
  prototype: Request;  
  new(input: RequestInfo, init?: RequestInit): Request;  
};
```

Здесь тип и значение `Request` моделируются отдельно (правило 8). Теперь обратитесь к `RequestInit`, и вы увидите все, что может быть использовано для построения `Request`:

```
interface RequestInit {  
  body?: BodyInit | null;  
  cache?: RequestCache;  
  credentials?: RequestCredentials;  
  headers?: HeadersInit;  
  // ...  
}
```

Есть и другие типы, которые приведут вас сюда. Поначалу декларации типов могут показаться сложными, но они хорошо показывают возможности TypeScript, принцип устройства библиотек и способы отладки ошибок (глава 6).

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Используйте возможности языковых служб TypeScript посредством подходящего редактора.
- ✓ Редактор поможет вам развить интуитивное понимание системы типов и принципов вывода типов TypeScript.
- ✓ Освойте работу с файлами деклараций, чтобы видеть, каким образом моделируется поведение кода.

ПРАВИЛО 7. Воспринимайте типы как наборы значений

При выполнении каждая переменная имеет одно значение из ограниченного множества значений. Вот лишь несколько примеров:

- ☐ 42
- ☐ null
- ☐ undefined
- ☐ 'Canada'
- ☐ {animal: 'Whale', weight_lbs: 40_000}
- ☐ /regex/
- ☐ new HTMLButtonElement
- ☐ (x, y) => x + y

Перед запуском кода TypeScript проверяет наличие ошибок — в это время переменные имеют *тип*, похожий на *набор возможных значений*. Такие наборы называют *областью* типов. К примеру, можно воспринимать тип `number` как набор числовых значений. В него будут входить 42, -37,5, но никак не 'Canada'. В зависимости же от настройки опции `strictNullChecks`, `null` и `undefined` могут являться или не являться частью области.

Пустой набор, который не содержит значений, меньше остальных. В TypeScript он соответствует типу `never`. Так как его область пуста, то переменной типа `never` не могут быть присвоены никакие значения:

```
const x: never = 12;  
// ~ Тип '12' не может быть назначен для типа 'never'.
```

Следующими по размеру являются наборы, содержащие единичные значения. Это касается литеральных типов, известных также как типы единиц:

```
type A = 'A';  
type B = 'B';  
type Twelve = 12;
```

Для формирования типов с двумя, тремя или более значениями используется объединение:

```
type AB = 'A' | 'B';  
type AB12 = 'A' | 'B' | 12;
```

Типы объединения соответствуют объединениям наборов значений.

Во многих сообщениях об ошибках в TypeScript фигурирует слово `assignable` (может быть назначен). В контексте наборов значений это

означает, что тип является либо частью набора (при связи между типом и значением), либо его подмножеством (при связи между типами):

```
const a: AB = 'A'; // ok, значение 'A' является частью набора {'A', 'B'}
const c: AB = 'C';
// ~ Тип "'C'" не может быть назначен для типа 'AB'.
```

Тип C является типом единицы, и его область состоит из одного значения C. Он не является подмножеством области AB, состоящей из значений A и B, поэтому возникнет ошибка. В общем, почти вся работа модуля проверки заключается в выяснении того, не является ли каждый отдельный набор подмножеством другого:

```
// Ok, {"A", "B"} является подмножеством {"A", "B"}:
const ab: AB = Math.random() < 0.5 ? 'A' : 'B';
const ab12: AB12 = ab; // ok, {"A", "B"} является подмножеством {"A",
// "B", 12}

declare let twelve: AB12;
const back: AB = twelve;
// ~~~~ Тип 'AB12' не может быть назначен для типа 'AB'.
//      Тип '12' не может быть назначен для типа 'AB'.
```

Наборы, относящиеся к этим типам, вполне понятны, так как являются конечными. Однако многие типы, с которыми вам предстоит работать, имеют бесконечные области значений. Их можно воспринимать как структурные построения:

```
type Int = 1 | 2 | 3 | 4 | 5 // | ...
```

или описать их составляющие:

```
interface Identified {
  id: string;
}
```

Считайте этот интерфейс описанием значений области его типа. Если интересующее значение имеет свойство `id`, чье значение может быть назначено для `string`, тогда оно будет `identifiable` (идентифицируемым).

Это *все*, о чем оно говорит. Правила структурной типизации в TypeScript позволяют одному значению иметь несколько свойств, включая даже возможность вызова (правило 4). Этот факт иногда скрывается за дополнительной проверкой свойств (правило 11).

Рассуждение о типах как о наборах значений помогает легче ими оперировать. Например:

```
interface Person {  
    name: string;  
}  
interface Lifespan {  
    birth: Date;  
    death?: Date;  
}  
type PersonSpan = Person & Lifespan;
```

Оператор `&` просчитывает пересечение двух типов. Какого рода значения в этом случае принадлежат типу `PersonSpan`? На первый взгляд интерфейсы `Person` и `Lifespan` не имеют общих свойств, поэтому вы можете ожидать получения пустого набора. Но операции типов применяются к наборам значений, а не к свойствам интерфейса. Запомните, что значения с дополнительными свойствами относятся к типу. Поэтому значение, имеющее свойства `Person` и `Lifespan` *одновременно*, будет принадлежать типу пересечения:

```
const ps: PersonSpan = {  
    name: 'Alan Turing',  
    birth: new Date('1912/06/23'),  
    death: new Date('1954/06/07'),  
}; // ok
```

Конечно, значение может иметь больше трех свойств, сохраняя тот же тип. Главное, что значения в типе пересечения содержат объединенные свойства в каждом своем элементе.

Пересечение свойств является корректным для *объединения* двух интерфейсов, а не для их пересечения:

```
type K = keyof (Person | Lifespan); // тип never
```

Не существует ключей, гарантированно принадлежащих значению типа объединения, поэтому `keyof` для этого объединения будет пустым набором (`never`):

```
keyof (A&B) = (keyof A) | (keyof B)  
keyof (A|B) = (keyof A) & (keyof B)
```

Если вы понимаете, почему выполняются эти уравнения, значит, вы далеко продвинулись в понимании системы типов TypeScript.

Тип `PersonSpan` можно написать проще — при помощи `extends`:

```
interface Person {  
    name: string;  
}  
interface PersonSpan extends Person {  
    birth: Date;  
    death?: Date;  
}
```

Но что же означает `extends` (расширение) в контексте восприятия типов в виде набора значений? Здесь оно выступает в роли подмножества, так как каждое значение, принадлежащее `PersonSpan`, должно иметь и свойство `name`, являющееся `string`, и свойство `birth`.

Вы также можете встретить термин «подтип» в значении «подмножество». В контексте 1-, 2- и 3-х мерных векторов он выглядит так:

```
interface Vector1D { x: number; }  
interface Vector2D extends Vector1D { y: number; }  
interface Vector3D extends Vector2D { z: number; }
```

Можно говорить, что `Vector3D` является подтипом `Vector2D`, который, в свою очередь, является подтипом `Vector1D` (в случае с классами вы бы говорили «подкласс»). Эту взаимосвязь обычно представляют в виде иерархии, но для иллюстрации наборов значений лучше подойдет диаграмма Венна (рис. 2.7).

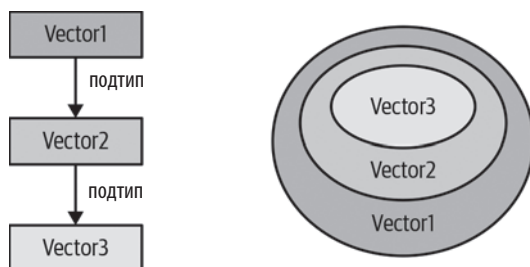


Рис. 2.7. Два способа представления связей типов: иерархия или наложение наборов

Диаграмма Венна показывает, что взаимосвязи подмножества / подтипа / возможности назначения не изменятся, если мы перепишем интерфейс без `extends`:

```
interface Vector1D { x: number; }
interface Vector2D { x: number; y: number; }
interface Vector3D { x: number; y: number; z: number; }
```

Наборы не изменились, следовательно, и диаграмма тоже.

`extends` также может оказаться ограничителем в обобщенных типах, сохраняя при этом значение подмножества (правило 14):

```
function getKey<K extends string>(val: any, key: K) {
  // ...
}
```

Что значит расширить `string`? Будет сложно представить это в рамках наследования объекта или пытаться определить подкласс оберточного объекта типа `String` (правило 10).

В контексте наборов значений все просто: расширением станет любой тип, чья область является подмножеством `string`: и литеральные типы, и объединения строковых литеральных типов, и сами `string`:

```
getKey({}, 'x'); // ok, 'x' расширяет string
getKey({}, Math.random() < 0.5 ? 'a' : 'b'); // ok, 'a'|'b' расширяет
// string
getKey({}, document.title); // ok, string расширяет string
getKey({}, 12);
// ~~ Тип '12' не может быть назначен параметру типа 'string'
```

В последней ошибке произошла замена `extends` (расширяет) на `assignable` (может быть назначен), но это не должно сбить вас с толку, так как вы знаете, что и то и другое обозначает подмножество. Этот подход также хорошо помогает при работе с конечными наборами значений, подобными получаемым при помощи конструкции `keyof T`, которая возвращает тип только для ключей типа объекта:

```
interface Point {
  x: number;
  y: number;
}
type PointKeys = keyof Point; // Тип "x" | "y"

function sortBy<K extends keyof T, T>(vals: T[], key: K): T[] {
  // ...
}
const pts: Point[] = [{x: 1, y: 1}, {x: 2, y: 0}];
```

```

sortBy(pts, 'x'); // ok, 'x' расширяет 'x'|'y' (aka keyof T)
sortBy(pts, 'y'); // ok, 'y' расширяет 'x'|'y'
sortBy(pts, Math.random() < 0.5 ? 'x' : 'y'); // ok, 'x'|'y' расширяет
                                                // 'x'|'y'
sortBy(pts, 'z');
    // ~~~ Тип '"z"' не может быть назначен параметру типа
    //      '"x" | "y"'

```

Восприятие через призму набора значений также помогает проще разобраться в случаях, когда между типами не определены строгие иерархические связи. К примеру, какова связь между `string|number` и `string|Date`? Они имеют общее пересечение (`string`), но в то же время ни один из них не является подмножеством другого. Взаимосвязь между их областями значений видна, даже несмотря на отсутствие строгой иерархии (рис. 2.8):

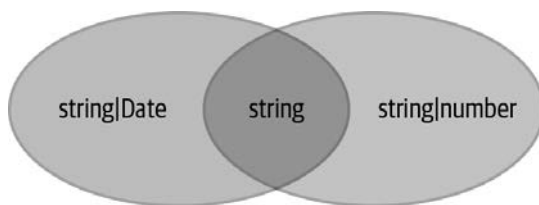


Рис. 2.8. Типы объединения могут не иметь взаимной иерархии, но при этом легко понимаются в контексте наборов значений

Становятся понятными взаимосвязи между массивами и кортежами:

```

const list = [1, 2]; // Тип number[]
const tuple: [number, number] = list;
    // ~~~~ Тип 'number[]' не имеет свойств типа '[number, number]': 0, 1

```

Здесь мы видим списки чисел, которые не являются парами чисел: пустой список и список `[1]`. Очевидно, что `number[]` не может быть назначен для `[number, number]`, поскольку не является его подмножеством (хотя обратное назначение работает).

Может ли тройка быть назначена для пары? Думая в категориях структурной типизации, вы можете ожидать, что это возможно. В паре есть ключи `0` и `1`, так не может ли она иметь и другие тоже? Например, `2`?

```

const triple: [number, number, number] = [1, 2, 3];
const double: [number, number] = triple;

```

```
// ~~~~~ '[number, number, number]' не может быть назначен
//      для '[number, number]'.
//      Типы свойства 'length' являются несовместимыми.
//      Тип '3' не может быть назначен для типа '2'.
```

Нет, и по весьма интересной причине. Вместо того чтобы моделировать пару чисел как `{0: number, 1: number}`, TypeScript моделирует ее как `{0: number, 1: number, length: 2}`. Это полезно — вы можете проверить длину кортежа, и она исключит лишнее назначение.

Если учитывать, что типы лучше всего рассматривать как наборы значений, то получится, что типы, содержащие одинаковые значения, тоже одинаковые. Это действительно так, когда два типа семантически разные, но, по совпадению, имеют одинаковые области значений.

В заключение стоит отметить, что не все наборы значений соответствуют типам TypeScript. В нем нет типов для всех целых чисел (*integers*) или для всех объектов, имеющих только свойства *x* и *y* и никакие другие. Вы можете исключать типы при помощи функции `Exclude`, но только если получите приемлемый тип TypeScript:

```
type T = Exclude<string|Date, string|number>; // Тип Date
type NonZeroNums = Exclude<number, 0>; // Тип все еще просто number
```

В табл. 2.1 показано соответствие понятий TypeScript и понятий из теории множеств.

Таблица 2.1. Выражения TypeScript и наборов значений

TypeScript	Наборы значений
<code>never</code>	$\emptyset$ (пустое множество)
литерал	Множество из одного элемента
назначение значения T	Значение $\in$ T (член множества)
T1 назначается T2	$T1 \subseteq T2$ (подмножество)
T1 расширяет T2	$T1 \subseteq T2$ (подмножество)
<code>T1 T2</code>	$T1 \cup T2$ (объединение)
<code>T1 & T2</code>	$T1 \cap T2$ (пересечение)
<code>unknown</code>	Универсальное множество

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Воспринимайте типы как наборы значений (*области* типа). Эти наборы могут быть как конечными (`boolean` либо литеральные типы), так и бесконечными (`number`, `string`).
- ✓ Типы TypeScript формируют пересекающиеся наборы (диаграмма Венна) вместо строгой иерархии. Два типа могут пересекаться, не являясь подтипами друг друга.
- ✓ Помните, что объект может по-прежнему принадлежать к определенному типу, несмотря на дополнительные свойства, которые не были отмечены в декларации типа.
- ✓ Операции с типами также применяются и к области набора их значений. Поэтому пересечение A и B также является и пересечением области A с областью B. Для типов объектов это означает, что значения A и B приобретают свойства как A, так и B.
- ✓ Воспринимайте `extends` (расширяет), `assignable to` (может быть назначен) и `subtype of` (подтип) как `subset of` (подмножество).

ПРАВИЛО 8. Правильно выражайте отношение символа к пространству типов или пространству значений

В TypeScript символ может существовать в одном из двух пространств:

- пространство типов;
- пространство значений.

Иногда возникает путаница, так как одно и то же имя может называть разные вещи в зависимости от того, в каком пространстве оно находится:

```
interface Cylinder {  
  radius: number;  
  height: number;  
}
```

```
const Cylinder = (radius: number, height: number) => ({radius, height});
```

`interface Cylinder` представляет символ в пространстве типов, а `const Cylinder` — символ в пространстве значений. Друг к другу они не имеют отношения. В зависимости от контекста, печатая `Cylinder`, вы будете обращаться к пространству либо типов, либо значений.

```
function calculateVolume(shape: unknown) {
  if (shape instanceof Cylinder) {
    shape.radius
    // ~~~~~ Свойство 'radius' не существует в типе '{}'.
  }
}
```

Почему возникают ошибки? Вы надеетесь, что `instanceof` проверит отношение `shape` (формы) к типу `Cylinder`. Но `instanceof` является оператором процедуры выполнения в JavaScript и работает со значениями. Поэтому `instanceof Cylinder` относится к функции, а не к типу.

Иногда сложно определить, к какому пространству символ относится. На помощь должен прийти контекст появления символа, однако многие конструкции пространств типов выглядят совершенно так, как конструкции пространств значений.

К примеру, так выглядят литералы:

```
type T1 = 'string literal';
type T2 = 123;
const v1 = 'string literal';
const v2 = 123;
```

Можно считать, что чаще всего символы, следующие за `type` или `interface`, находятся в пространстве типов, в то время как представленные в декларациях `const` либо `let` — являются значениями.

Один из лучших способов выработать интуицию в отношении двух пространств — это игровая площадка¹ TypeScript. Она покажет исходный TypeScript-код в виде JavaScript. В процессе компиляции типы будут удалены (правило 3), поэтому в случае исчезновения символа станет понятно, что он относился к пространству типов (рис. 2.9).

Инструкции в TypeScript могут колебаться между пространствами типов и значений. Символ, идущий после объявления типа (`:`) или утверждения (`as`), относится к пространству типов, в то время как все, что следует после `=`, является частью пространства значений.

¹ <https://www.typescriptlang.org/play/>

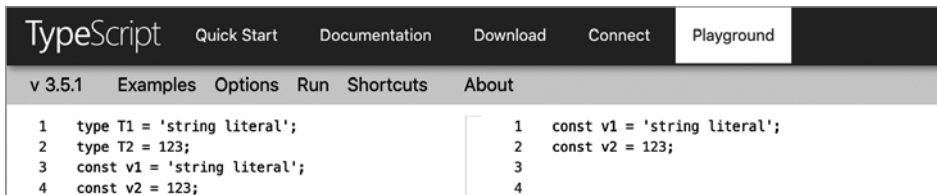


Рис. 2.9. Песочница TypeScript демонстрирует сгенерированный JavaScript-код. Символы на первых двух строках удаляются, следовательно, они относились к пространству типов

Например:

```
interface Person {
  first: string;
  last: string;
}
const p: Person = { first: 'Jane', last: 'Jacobs' };
//      -      ----- значения
//      ----- тип
```

Особенно это касается инструкций функций, которые колеблются между пространствами:

```
function email(p: Person, subject: string, body: string): Response {
  //      -----      -----      ---- значения
  //      -----      -----      -----      ----- типы
  // ...
}
```

Конструкции `class` и `enum` представляют одновременно и типы, и значения. В первом примере `Cylinder` должен был быть `class`:

```
class Cylinder {
  radius=1;
  height=1;
}

function calculateVolume(shape: unknown) {
  if (shape instanceof Cylinder) {
    shape // ok, тип Cylinder shape
    radius // ok, тип number
  }
}
```

В TypeScript тип, представленный в виде класса, основывается на своей форме (свойствах и методах), в то время как значение является конструкцией.

Существует много операторов и ключевых слов, имеющих разные значения в зависимости от контекста пространства. Например, `typeof`:

```
type T1 = typeof p; // Тип Person
type T2 = typeof email;
    // Тип (p: Person, субъект: string, тело: string) => Response

const v1 = typeof p; // значение object
const v2 = typeof email; // значение function
```

В контексте типа `typeof` принимает значение и возвращает его тип TypeScript. Можно использовать его как часть более крупного выражения типа либо применить инструкцию `type`, чтобы присвоить ему имя.

В контексте значения `typeof` является оператором процесса выполнения JavaScript. Он возвращает строку, содержащую тип, соответствующий символу при выполнении. И не является типом TypeScript! Система типов процедуры выполнения JavaScript существенно проще, чем статическая система типов TypeScript. В противовес разнообразию типов TypeScript за всю историю JavaScript в ее процессе выполнения существовало всего шесть типов: `string`, `number`, `boolean`, `undefined`, `object` и `function`.

Оператор `typeof` всегда работает со значениями. Его нельзя применять к типам. Ключевое слово `class` представляет и значение, и тип — так каков же `typeof` класса? Зависит от контекста:

```
const v = typeof Cylinder; // значение function
type T = typeof Cylinder; // тип typeof Cylinder
```

Значение является `function`, потому что так реализуются классы в JavaScript. Тип явно не выделяется. Важно то, что он *не* `Cylinder` (тип экземпляра). На деле это функция-конструктор, которую вы можете увидеть при помощи `new`:

```
declare let fn: T;
const c = new fn(); // Тип Cylinder
```

Чтобы перемещаться между типом конструктора и типом экземпляра, используйте обобщение `Instance Type`:

```
type C = InstanceType<typeof Cylinder>; // Тип Cylinder
```

Метод доступа к свойствам `[]` также имеет идентичный эквивалент в пространстве типов. Но имейте в виду, что, в то время как `obj['field']` и `obj.field` являются эквивалентными в пространстве значений, они не являются таковыми для пространства типов. Получить тип свойства другого типа можно с помощью формировщика:

```
const first: Person['first'] = p['first']; // либо p.first
// ----- значения
// ----- типы
```

Здесь `Person['first']` — *тип*, так как появляется в контексте типа (после `:`). Вы можете вставить любой тип в ячейку индекса, включая типы объединения и примитивные типы:

```
type PersonEl = Person['first' | 'last']; // Тип string
type Tuple = [string, number, Date];
type TupleEl = Tuple[number]; // Тип string | number | Date
```

Подробнее эта тема раскрыта в правиле 14.

Существуют и другие конструкции, которые имеют разное значение в зависимости от пространства:

- Метод `this` в пространстве значений является ключевым словом JavaScript (правило 49). В пространстве типов `this` выступает как TypeScript-тип `this` (а-ля «полиморфный `this`»). Он полезен при выполнении цепочек методов для подклассов.
- В качестве префикса `!` выступает как оператор единичного отрицания в JavaScript (а-ля «не»). Будучи суффиксом, он означает ненулевое утверждение типа (правило 9).
- В пространстве значений `&` и `|` являются побитовыми операциями «и» и «или». В пространстве типов они выступают как операторы объединения и пересечения.
- Ключевое слово `const` вводит новую переменную, но `as const` изменяет выведенный тип на постоянный (правило 21).
- `extends` может либо определять подкласс (класс `A` расширяет класс `B`) или подтип (интерфейс `A` расширяет интерфейс `B`), либо ставить ограничение на обобщенный тип (`Generic<T extends number>`).
- `in` может являться либо частью цикла `for...in`, либо отображаемым типом (правило 14).

Если TypeScript никак не желает понимать ваш код, то это может быть связано с запутанностью вокруг пространств типов и значений. Например, вы изменили функцию `email`, чтобы передать ее аргументы в единичный параметр объекта:

```
function email(options: {person: Person, subject: string, body: string}) {  
    // ...  
}
```

В JavaScript можно использовать деструктурирующее присваивание, чтобы создать локальные переменные для каждого свойства объекта:

```
function email({person, subject, body}) {  
    // ...  
}
```

Если вы попытаетесь повторить это в TypeScript, то получите сбивающие с толку ошибки:

```
function email({  
    person: Person,  
        // ~~~~~ Присваиваемый элемент 'Person' неявно имеет тип 'any'.  
    subject: string,  
        // ~~~~~ Дублирование идентификатора 'string'.  
        // Присваиваемый элемент 'string' неявно имеет тип 'any'.  
    body: string  
        // ~~~~~ Дублирование идентификатора 'string'.  
        // Присваиваемый элемент 'string' неявно имеет тип 'any'.  
) { /* ... */ }
```

Проблема в том, что `Person` и `string` воспринимаются в контексте значений. Вы пытаетесь создать переменную с именем `Person` и две переменные с именем `string`. Вместо этого вам следует отделить типы от значений:

```
function email(  
    {person, subject, body}: {person: Person, subject: string, body: string}  
) {  
    // ...  
}
```

Выглядит сложно, но на деле достаточно иметь либо названный тип для параметров, либо возможность вывести его исходя из контекста (правило 26).

Несмотря на то что сперва схожие конструкции типов и значений могут быть несколько запутывающими, впоследствии они оказываются весьма полезными в качестве мнемосхемы. Надо только их освоить.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ В процессе чтения выражений TypeScript умеете правильно выразить их отношение к пространствам типов и значений. Используйте игровую площадку TypeScript для тренировки интуиции в этом процессе.
- ✓ Каждое значение имеет тип, но типы значений не имеют. Конструкции наподобие `type` и `interface` существуют только в пространстве типов.
- ✓ `foo` может быть как строковым литералом, так и типом строкового литерала. Имейте в виду отличие между ними и умеете правильно его выразить.
- ✓ `typeof`, `this` и многие другие операторы, а также ключевые слова имеют различные значения для пространства типов и пространства значений.
- ✓ Некоторые конструкции вроде `class` или `enum` представляют одновременно и тип, и значение.

ПРАВИЛО 9. Объявление типа лучше его утверждения

В TypeScript существует два способа присваивания значения переменной и определения ее типа:

```
interface Person { name: string };

const alice: Person = { name: 'Alice' }; // Тип Person
const bob = { name: 'Bob' } as Person; // Тип Person
```

Несмотря на то что оба варианта приходят к одному итогу, они имеют отличия. `alice: Person` добавляет переменной *декларацию типа* и гарантирует, что значение ему соответствует. Следующее выражение `as Person` выполняет *утверждение типа*. Тип сообщает TypeScript, но вы настаиваете на своем варианте.

В большинстве случаев стоит предпочесть объявление типа, а не его утверждение, и вот почему:

```
const alice: Person = {};
// ~~~~ Свойство 'name' отсутствует в типе '{}'
//      но необходимо для типа 'Person'.
const bob = {} as Person; // нет ошибки
```

Декларация типа определяет, что значение должно согласовываться с `interface`, но поскольку этого не происходит, фиксируется ошибка. При утверждении типа эта ошибка замалчивается, так как вы определили, что при любых обстоятельствах ваш вариант верный.

То же самое происходит, если вы определяете дополнительное свойство:

```
const alice: Person = {
  name: 'Alice',
  occupation: 'TypeScript developer'
// ~~~~~ Объектный литерал может определять только известные свойства,
//       а 'occupation' не существует в типе 'Person'.
};
const bob = {
  name: 'Bob',
  occupation: 'JavaScript developer'
} as Person; // нет ошибки
```

Расширенная проверка дополнительных свойств (правило 11) не срабатывает, если использовать утверждение.

Объявление типа гарантирует дополнительную проверку безопасности кода и является предпочтительным, если нет серьезных причин для использования утверждения.



Вы можете встретить такой вид кода: `const bob = <Person>{}`. Это оригинальный вариант синтаксиса, соответствующий утверждению, и он является эквивалентом `{ } as Person`. Сейчас этот вариант не столь актуален, так как `<Person> is` понимается как начальный тег в файлах `.tsx` (TypeScript + React).

Недостаточно ясно, как использовать объявление типа со стрелочными функциями. Например, что произойдет, если вы используете в приведенном коде именованный интерфейс `Person`?

```
const people = ['alice', 'bob', 'jan'].map(name => ({name}));
// { name: string; }[]... но нам нужен Person[]
```

Возникает искушение использовать утверждение типа, чтобы решить проблему:

```
const people = ['alice', 'bob', 'jan'].map(
  name => ({name} as Person)
); // Тип Person[]
```

Но это приведет к сложностям, как и прямое утверждение типа. Например:

```
const people = ['alice', 'bob', 'jan'].map(name => ({ } as Person));  
// нет ошибки
```

Как же в таком контексте применить объявление типа? Наиболее прямой путь — это объявление переменной в стрелочной функции:

```
const people = ['alice', 'bob', 'jan'].map(name => {  
  const person: Person = {name};  
  return person  
}); // Тип Person[]
```

К сожалению, код стал более громоздким по сравнению с исходным. Существует и краткий путь — объявить возвращаемый тип стрелочной функции:

```
const people = ['alice', 'bob', 'jan'].map(  
  (name): Person => ({name})  
); // Тип Person[]
```

В этом случае будут выполнены все те же проверки значения, что и в предыдущей версии. Круглые скобки — решающие. Команда `(name) : Person` делает вывод типа `name` и указывает, что возвращаемый тип должен быть `Person`. Написание же `(name: Person)` определило бы тип `name` как `Person` и позволило сделать вывод возвращаемого типа, что привело бы к ошибке.

Еще один вариант — прописать итоговый желаемый тип и позволить TypeScript произвести проверку работоспособности такого назначения:

```
const people: Person[] = ['alice', 'bob', 'jan'].map(  
  (name): Person => ({name})  
);
```

Но в контексте более длинной цепочки вызова функции может потребоваться ранее названный тип. Это также поможет обнаружить ошибки именно в местах их появления.

Когда же вам *стоит* использовать утверждение типов? Например, в контексте, недоступном модулю проверки типов, где вы знаете тип элемента DOM лучше TypeScript:

```
document.querySelector('#myButton').addEventListener('click',  
  e => { e.currentTarget // Тип EventTarget  
    const button = e.currentTarget as HTMLButtonElement;  
    button // Тип HTMLButtonElement  
  });
```

Так как у TypeScript нет доступа к DOM вашей страницы, он не имеет представления, что `#myButton` является кнопочным элементом. Ему также неизвестно, что `currentTarget` события должна совпадать с этой кнопкой. Подробнее о DOM ищите в правиле 55.

Вы также можете встретить `non-null` (ненулевое) утверждение, которое является настолько обыденным, что даже получило свой синтаксис:

```
const elNull = document.getElementById('foo'); // Тип HTMLElement | null
const el = document.getElementById('foo')!; // Тип HTMLElement
```

Использованный в качестве префикса, `!` является логическим отрицанием. Но в роли суффикса `!` он утверждает, что значение является ненулевым. Как и любое другое утверждение, он удаляется в процессе компиляции, поэтому используйте его только тогда, когда можете убедиться, что значение является ненулевым. Если такой возможности нет, используйте для проверки условное выражение.

Утверждение типов не позволяет преобразовывать произвольные типы, которые дают возможность совершать преобразование между A и B, при условии что один из них является подмножеством другого. `HTMLElement` является подтипом `HTMLElement | null`, поэтому здесь уместно утверждение типа, как и в случаях с `HTMLButtonElement` — подтипом `EventTarget` и `Person` — подтипом `{}`.

Но вы не можете произвести преобразование между `Person` и `HTMLElement`, поскольку ни один из них не является подтипом другого:

```
interface Person { name: string; }
const body = document.body;
const el = body as Person;
    // ~~~~~ Преобразование типа 'HTMLElement' в тип 'Person'
    // может обозначаться ошибкой, потому что ни один
    // из этих типов не имеет достаточно общего
    // с другим. Если это было допущено умышленно,
    // то сперва произведите преобразование
    // в 'unknown'.
```

Такая ошибка имеет решение — использование типа `unknown` (правило 42). Любой тип является подтипом `unknown`, поэтому утверждения с ним всегда срабатывают. Он позволит произвести преобразование между произвольными типами и потом вспомнить, где и почему вы это сделали.

```
const el = document.body as unknown as Person; // ok
```

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Предпочитайте объявление типов (`: Type`) утверждению типов (`as Type`).
- ✓ Умейте правильно аннотировать возвращаемый тип стрелочной функции.
- ✓ Используйте утверждение типов и ненулевые утверждения, только когда знаете о типах больше, чем TypeScript.

ПРАВИЛО 10. Избегайте оберточных типов (String, Number, Boolean, Symbol, BigInt)

Помимо объектов, в JavaScript присутствуют семь типов примитивных значений: `strings`, `numbers`, `boolean`, `null`, `undefined`, `symbol` и `bigint`. Первые пять из них были в языке изначально. `symbol` же был добавлен в ES2015, а `bigint` находится на последней стадии внедрения.

Примитивы отличаются от объектов отсутствием методов и неизменностью. Хотя вы можете сказать, что `string` *имеют* методы:

```
> 'primitive'.charAt(3)
"m"
```

JavaScript определяет тип *объекта* `String`, который имеет методы. При обращении к методу JS оборачивает *примитив* `string` в объект `String`, вызывает метод и затем выбрасывает объект.

Вы можете пронаблюдать этот процесс, если используете обезьяний патч для `String.prototype` (правило 43):

```
// Не делайте этого!
const originalCharAt = String.prototype.charAt; String.prototype.charAt =
    function(pos) {
        console.log(this, typeof this, pos);
        return originalCharAt.call(this, pos);
    };
console.log('primitive'.charAt(3));
```

Так вы получите следующий вывод:

```
[String: primitive] object 3
m
```

Значение `this` в методе является оберткой объекта `String`, но не примитивом `string`. Вы можете инстанцировать объект `String` напрямую, и иногда он будет вести себя как примитив. Но так будет не всегда. Например, объект `String` может быть всегда равен только себе:

```
> "hello" === new String("hello")
false
> new String("hello") === new String("hello")
false
```

Неявное преобразование в оберточные типы объектов объясняет странный феномен JavaScript: если вы назначаете свойство примитиву, то оно исчезает.

```
> x = "hello"
> x.language = 'English'
'English'
> x.language
undefined
```

Теперь вам известно, что `x` преобразовывается в экземпляр `String`, к нему добавляется свойство `language`, и затем объект (со свойством `language`) выбрасывается.

Существуют оберточные типы объектов и для других примитивов: `Number` для `number`, `Boolean` для `boolean`, `Symbol` для `symbol` и `BigInt` для `bigint`. А для `null` и `undefined` оберточных типов нет.

В целом обертки используются для применения методов к значениям примитивов и для использования статических методов (`String`, `fromCharCode` и т. п.). Хотя обычно не возникает необходимости инстанцировать их напрямую.

TypeScript моделирует это отличие посредством различия типов примитивов и их оберточных объектов:

- `string` и `String`
- `number` и `Number`
- `boolean` и `Boolean`
- `symbol` и `Symbol`
- `bigint` и `BigInt`

Достаточно легко по ошибке напечатать `String` (особенно если вы привыкли к Java или C#) и увидеть, что все работает. По крайней мере вначале:

```
function getStringLen(foo: String) {
    return foo.length;
}

getStringLen("hello"); // ok
getStringLen(new String("hello")); // ok
```

Но неполадки возникнут, как только вы попыботе передать объект `String` в метод, который ожидает `string`:

```
function isGreeting(phrase: String) {
    return [
        'hello',
        'good day'
    ].includes(phrase);
    // ~~~~~
    // Аргумент типа 'String' не может быть назначен в качестве
    // параметра типа 'string'.
    // 'string' является примитивом, но 'String' является
    // оберточным объектом.
    // Старайтесь использовать 'string' везде, где это возможно.
}
```

Итак, `string` может быть назначен для `String`, но не наоборот. Чтобы не запутаться, придерживайтесь использования `string`. Все декларации типов, которые идут с TypeScript, используют его так же, как и иные типизации для почти всех других библиотек.

Еще один способ избежать сложности с оберточными типами — сделать явную аннотацию типа с заглавной буквы:

```
const s: String = "primitive";
const n: Number = 12;
const b: Boolean = true;
```

Естественно, значения при выполнении по-прежнему будут примитивами, а не объектами. Но TypeScript позволяет подобное объявление, так как типы примитивов могут быть назначены для оберточных типов. Подобные аннотации являются и обманчивыми, и излишними одновременно (правило 19). Лучше просто использовать примитивные типы.

В заключение скажу, что вполне нормально вызывать `BigInt` и `Symbol`, не используя `new`, поскольку они создают примитивы:

```
> typeof BigInt(1234)
"bigint"
> typeof Symbol('sym')
"symbol"
```

Здесь представлены *значения* `BigInt` и `Symbol`, но не типы TypeScript (правило 8). Их вызов в итоге приводит к значениям типа `bigint` или `symbol`.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Существуют оберточные типы для применения методов к значениям примитивов. Избегайте инстанцировать их или использовать напрямую.
- ✓ Лучше использовать примитивные, а не оберточные типы: `string` вместо `String`, `number` вместо `Number`, `Boolean` вместо `boolean`, `symbol` вместо `Symbol` и `bigint` вместо `BigInt`.

ПРАВИЛО 11. Проверяйте пределы исключительных свойств типа

Когда вы назначаете объектный литерал к переменной с объявленным типом, TypeScript дополнительно убеждается, что она обладает исключительно свойствами этого типа и *никакими другими*:

```
interface Room {
  numDoors: number;
  ceilingHeightFt: number;
}
const r: Room = {
  numDoors: 1,
  ceilingHeightFt: 10,
  elephant: 'present',
// ~~~~~ Объектный литерал может определять только известные
//          свойства, а 'elephant' не существует в типе 'Room'.
};
```

Несмотря на присутствие неуместного свойства `elephant`, с точки зрения структурной типизации такая ошибка не имеет большого значения (правило 4). Эта константа может быть назначена для типа `Room`, в чем вы можете убедиться, введя промежуточную переменную.

```
const obj = {
  numDoors: 1,
  ceilingHeightFt: 10,
  elephant: 'present',
};
const r: Room = obj; // ok
```

Тип `obj` выводится как `{numDoors: number; ceilingHeightFt: number; elephant: string; }`. Так как этот тип содержит в себе подмножество значений в типе `Room`, он может быть назначен для `Room`, и код пройдет проверку типов (правило 7).

Так в чем же разница между этими двумя примерами? В первом вы спровоцировали запуск *проверки лишних свойств*, благодаря чему нашли важные ошибки. Однако ее совмещение с регулярной проверкой возможности назначения может вас совсем расслабить. Распознавание проверки лишних свойств как отдельного процесса поможет выстроить более ясную сознательную модель системы типов TypeScript.

TypeScript старается указать на код, который будет выдавать исключения при выполнении (правило 1), а также стремится найти сегменты кода, не делающие то, что вы хотите. Например:

```
interface Options {
  title: string;
  darkMode?: boolean;
}
function createWindow(options: Options) {
  if (options.darkMode) {
    setDarkMode();
  }
  // ...
}
createWindow({
  title: 'Spider Solitaire',
  darkmode: true
  // ~~~~~ Объектный литерал может определять только известные
  // свойства, но 'darkmode' не существует в типе 'Options'.
  // Возможно, вы хотели написать 'darkMode'?
});
```

Этот код не выдаст ошибку при выполнении. Однако он и не сделает того, что вы хотели по конкретной причине: не указано `darkMode` (заглавная M).

Структурный модуль проверки типов не смог бы определить такой вид ошибки, так как область типа `Options` невероятно велика: она включает все объекты со свойством `title`, являющимся `string`, а также *любые другие свойства*, не содержащие свойство `darkMode` со значением, отличным от `true` или `false`.

Бывает трудно осознать, насколько широки границы типов в TypeScript. Вот еще ряд значений, которые могут быть назначены для `Options`:

```
const o1: Options = document; // ok
const o2: Options = new HTMLAnchorElement; // ok
```

И `document`, и экземпляры `HTMLAnchorElement` имеют свойства `title`, являющиеся `string`. Поэтому указанные назначения подходят.

Проверка лишних свойств стремится обуздать это разнообразие, не компрометируя фундаментальную структурную основу системы типов. Она запрещает назначения неизвестных свойств для объектных литералов, в связи с чем иногда ее называют строгой проверкой объектных литералов. Ни `document`, ни `new HTMLAnchorElement` не являются таковыми, поэтому не вызвали проверку. А объект `{title, darkmode}` спровоцировал такую реакцию:

```
const o: Options = { darkmode: true, title: 'Ski Free' };
// ~~~~~ 'darkmode' не существует в типе 'Options'...
```

Это объясняет, почему использование промежуточной переменной без аннотации типа устраняет ошибку:

```
const intermediate = { darkmode: true, title: 'Ski Free' };
const o: Options = intermediate; // ok
```

В то время как правая сторона первой строки является объектным литералом, правая сторона второй (`intermediate`) — нет, поэтому проверка лишних свойств не происходит, а ошибка устраняется.

Эта проверка также не произойдет, когда вы используете утверждение типа:

```
const o = { darkmode: true, title: 'Ski Free' } as Options; // ok
```

Помните, что лучше предпочесть объявления типов их утверждениям (правило 9).

Если же вы хотите избежать проверки, то следует применить сигнатуру индекса (правило 15), чтобы уведомить TypeScript о необходимости ожидать дополнительные свойства:

```
interface Options {
  darkMode?: boolean;
  [otherOptions: string]: unknown;
}
const o: Options = { darkmode: true }; // ok
```

В правиле 15 обсуждается, когда это является и не является подходящим способом для моделирования данных.

Схожая проверка происходит для «слабых» типов, которые имеют только опциональные свойства:

```
interface LineChartOptions {  
  logscale?: boolean;  
  invertedYAxis?: boolean;  
  areaChart?: boolean;  
}  
const opts = { logScale: true };  
const o: LineChartOptions = opts;  
  // ~ Тип '{ logScale: boolean; }' не имеет общих свойств  
  // с типом 'LineChartOptions'.
```

Со структурной точки зрения тип `LineChartOptions` должен включать почти все объекты. Для слабых типов TypeScript добавляет другую проверку, которая следит, чтобы тип значения и объявленный тип имели хотя бы одно общее свойство, и находит опечатки. Она не является строго структурной и проводится только совместно с проверками возможности назначения. Исключение промежуточной переменной не поможет пройти проверку.

Проверка лишних свойств находит ошибки, которые были бы пропущены структурной системой типов. Особенно это оказывается полезным с типами наподобие `Options`, которые содержат опциональные поля. Но нужно помнить, что она применяется только к объектным литералам, и осознавать ее отличие от обычной проверки типов.

Исключение константы устранило ошибку, но оно может привести к появлению ошибок в других контекстах (правило 26).

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Когда вы присваиваете объектный литерал переменной или передаете его в качестве аргумента в функцию, он подвергается проверке лишних свойств.
- ✓ Проверка лишних свойств — это эффективный способ обнаружения ошибок, но она отличается от обычной структурной проверки возможности назначения, проводимой модулем проверки типов TypeScript. Объединение этих процессов усложнит для вас построение ментальной модели возможностей назначения.
- ✓ Будьте внимательны к границам диапазона проверок лишних свойств: введение промежуточной переменной отключит такую проверку.

ПРАВИЛО 12. По возможности применяйте типы ко всему выражению функции

JavaScript (и TypeScript) различают *инструкции* и *выражения функций*:

```
function rollDice1(numSides: number): number { /* ... */ } // инструкция
const rollDice2 = function(numSides: number): number { /* ... */ }; // выражение
const rollDice3 = (numSides: number): number => { /* ... */ }; // также
// выражение
```

Преимущество выражений функций в TypeScript заключается в том, что они позволяют одновременно применить объявление типа ко всей функции вместо указания типов параметров и возвращаемых типов в отдельности:

```
type DiceRollFn = (numSides: number) => number;
const rollDice: DiceRollFn = numSides => { /* ... */ };
```

Если вы наведете курсор на `numSides` в редакторе, то увидите, что TypeScript знает его тип `number`. Тип функции не представляет особой ценности в этом простом примере, но сама техника раскрывает ряд возможностей.

Одна из них — борьба с повторами. Написать несколько функций для выполнения арифметических операций с числами, к примеру, можно так:

```
function add(a: number, b: number) { return a + b; }
function sub(a: number, b: number) { return a - b; }
function mul(a: number, b: number) { return a * b; }
function div(a: number, b: number) { return a / b; }
```

Либо можно объединить повторяемые сигнатуры функций посредством единичного типа функции:

```
type BinaryFn = (a: number, b: number) => number;
const add: BinaryFn = (a, b) => a + b;
const sub: BinaryFn = (a, b) => a - b;
const mul: BinaryFn = (a, b) => a * b;
const div: BinaryFn = (a, b) => a / b;
```

Здесь меньше аннотаций, и они отделены от выполнений функции. Это делает логику более наглядной. Вы также получите проверку, имеют ли все выражения функции возвращаемый тип `number`.

Библиотеки часто предоставляют типы для распространенных сигнатур функций. Например, ReactJS предоставляет тип `MouseEventHandler`, который

можно применить ко всей функции, вместо назначения `MouseEvent` в роли типа параметра этой функции. Если вы являетесь автором библиотеки, рассмотрите возможность предоставления деклараций типов для распространенных обратных вызовов.

Чтобы применить тип к выражению функции для сопоставления с другой функцией в браузере, воспользуйтесь функцией `fetch`, которая производит HTTP-запрос к определенному ресурсу:

```
const responseP = fetch('/quote?by=Mark+Twain'); // Тип Promise<Response>
```

Из ответа можно извлечь данные через `response.json()` либо `response.text()`:

```
async function getQuote() {
  const response = await fetch('/quote?by=Mark+Twain');
  const quote = await response.json();
  return quote;
}

// {
//   "цитата": "Если вы говорите правду, то вам не потребуется ничего
//   запоминать.",
//   "источник": "ноутбук",
//   "дата": "1894"
// }
```

(См. правило 25, поясняющее *промисы*, `async` и `await`.)

Здесь присутствует баг: если запрос `/quote` не пройдет, тело ответа должно содержать пояснение вроде «404 Not Found». Это не формат JSON, поэтому `response.json()` вернет отвергнутый промис с сообщением о некорректном JSON. Это скроет реальную ошибку 404.

Легко забыть, что ответ об ошибке с `fetch` не завершается отвергнутым промисом. Давайте напишем функцию `checkedFetch` для проверки статуса. Декларации типов для `fetch` в `lib.dom.d.ts` выглядят так:

```
declare function fetch(
  input: RequestInfo, init?: RequestInit
): Promise<Response>;
```

Значит, вы можете написать `checkedFetch` так:

```
async function checkedFetch(input: RequestInfo, init?: RequestInit) {
  const response = await fetch(input, init);
  if (!response.ok) {
```

```

    // преобразован в отвергнутый промис в асинхронной функции.
    throw new Error('Request failed: ' + response.status);
  }
  return response;
}

```

Это сработает, но можно написать более лаконично:

```

const checkedFetch: typeof fetch = async (input, init) => {
  const response = await fetch(input, init);
  if (!response.ok) {
    throw new Error('Request failed: ' + response.status);
  }
  return response;
}

```

Мы изменили инструкцию функции на ее выражение и применили тип (`typeof fetch`) ко всей функции. Это позволило TypeScript сделать вывод типов параметров `input` и `init`.

Аннотация типа также гарантирует, что возвращаемый тип `checkedFetch` будет тем же, что и у `fetch`. Например, если вы написали `return` вместо `throw`, TypeScript поймает эту ошибку:

```

const checkedFetch: typeof fetch = async (input, init) => {
  // ~~~~~ Тип 'Promise<Response | HTTPError>'
  //       не может быть назначен для типа 'Promise<Response>'.
  //       Тип 'Response | HTTPError' не может быть назначен
  //       для типа 'Response'.
  const response = await fetch(input, init);
  if (!response.ok) {
    return new Error('Request failed: ' + response.status);
  }
  return response;
}

```

Такая же ошибка в первом примере проявилась бы в коде, который вызвал `checkedFetch`, а не в реализации.

Помимо краткости, типизация всего выражения функции более безопасна. Когда вы пишете функцию, которая имеет сигнатуру типа, аналогичную другой функции, либо пишете множество функций с одинаковыми сигнатурами типов, либо рассмотрите, где можно применить декларацию типа для всей функции вместо повторного указания типов параметров и возвращаемых значений.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Рассмотрите применение аннотаций типов к целым выражениям функций вместо прописывания их для каждого параметра и возвращаемого типа.
- ✓ Если вы пишете одну и ту же сигнатуру типа снова и снова, выведите общий тип функции либо ищите существующий. Если вы разработчик библиотеки, включите в нее типы для распространенных обратных вызовов.
- ✓ Используйте `type of fn` для соответствия сигнатуры одной функции сигнатуре другой.

ПРАВИЛО 13. Знайте разницу между `type` и `interface`

Если вы хотите определить именованный тип в TypeScript, то у вас есть два варианта:

```
type TState = {  
  name: string;  
  capital: string;  
}
```

Или интерфейс:

```
interface IState {  
  name: string;  
  capital: string;  
}
```

(Также можно использовать `class`, но он относится к понятиям среды выполнения JavaScript и тоже вводит значение (правило 8).)

Что же вам следует использовать, `type` или `interface`? Граница между этими вариантами с течением времени размылась настолько, что в некоторых ситуациях можно использовать и то и другое. Обратите внимание на оставшиеся различия между ними и следуйте постоянству применения того или другого в конкретных ситуациях. Еще вам нужно знать, как прописать один и тот же тип при помощи любого из этих способов. Это поможет легче воспринимать код TypeScript, использующий их.



Имена типов в примерах этого правила содержат префиксы **I** (**interface**) или **T** (**type**) для лучшей наглядности их происхождения. Вам не следует представлять их так в своем коде. Использование префиксов для интерфейсных типов является обыденным в C# и даже было ранее внедрено в TypeScript. Сегодня этот вариант считается плохим стилем, так как не применяется на постоянной основе в стандартных библиотеках.

Сперва рассмотрим сходства: типы `State`, указанные выше, являются практически неотличимыми друг от друга. Если вы определите значение `IState` либо `TState` с дополнительным свойством, то ошибки в результате будут идентичными:

```
const wyoming: TState = {
  name: 'Wyoming',
  capital: 'Cheyenne',
  population: 500_000
// ~~~~~ Тип ... не может быть назначен для типа 'TState'.
//      Объектный литерал может определять только известные
//      свойства, а 'population' не существует в типе
//      'TState'.
};
```

Вы можете использовать сигнатуру индекса и для `interface`, и для `type`:

```
type TDict = { [key: string]: string };
interface IDict {
  [key: string]: string;
}
```

Также можно определить типы функций в обоих случаях:

```
type TFn = (x: number) => string;
interface IFn {
  (x: number): string;
}

const toStrT: TFn = x => '' + x; // ok
const toStrI: IFn = x => '' + x; // ok
```

Псевдоним типа выглядит более естественно для этого простейшего типа функции, но если тип будет иметь еще и свойства, тогда декларация будет выглядеть иначе:

```
type TFnWithProperties = {
  (x: number): number;
  prop: string;
```

```

}
interface IFnWithProperties {
  (x: number): number;
  prop: string;
}

```

Этот синтаксис легче запомнить, зная, что в JavaScript функции являются вызываемыми объектами.

И псевдонимы типов, и интерфейсы могут быть обобщенными:

```

type TPair<T> = {
  first: T;
  second: T;
}
interface IPair<T> {
  first: T;
  second: T;
}

```

`interface` может расширять `type` (с некоторыми оговорками, упомянутыми ниже) и наоборот:

```

interface IStateWithPop extends TState {
  population: number;
}
type TStateWithPop = IState & { population: number; };

```

Снова типы оказались идентичными. Оговорка заключается в том, что `interface` не может расширять сложные типы вроде типов объединений. Если вам нужно именно это, придется использовать `type` и `&`.

Класс может реализовывать как `interface`, так и простой тип:

```

class StateT implements TState {
  name: string = '';
  capital: string = '';
}
class StateI implements IState {
  name: string = '';
  capital: string = '';
}

```

Так в чем же отличия? Одно вы уже видели: существуют *типы* объединения, но не существует интерфейсов объединения:

```

type AorB = 'a' | 'b';

```

Расширение типов объединений может быть весьма полезно. Если у вас есть отдельные типы для переменных `Input` и `Output` и отображения из имени в переменную:

```
type Input = { /* ... */ };
type Output = { /* ... */ };
interface VariableMap {
  [name: string]: Input | Output;
}
```

тогда может понадобиться тип, приписывающий переменной имя. Например:

```
type NamedVariable = (Input | Output) & { name: string };
```

Этот тип не может быть выражен через `interface`. В целом `type` имеет больше возможностей, чем `interface`. Он может выступать в качестве объединения, а также пользоваться более продвинутыми возможностями вроде отображения или условных типов.

Помимо этого, он может более ясно выражать кортежи и типы массивов:

```
type Pair = [number, number];
type StringList = string[];
type NamedNums = [string, ...number[]];
```

Вы можете выразить что-либо *подобное* кортежу, используя `interface`:

```
interface Tuple {
  0: number;
  1: number;
  length: 2;
}
const t: Tuple = [10, 20]; // ok
```

Но это неудобно и отбрасывает методы кортежа, такие как `concat`. Лучше использовать `type`. Подробную информацию по числовым индексам см. в правиле 16.

Тем не менее у `interface` есть некоторые возможности, отсутствующие у `type`. Одна из них — это то, что `interface` может быть *дополнен*. Возвращаясь к примеру со `State`, вы могли бы добавить поле `population` другим способом:

```
interface IState {
  name: string;
  capital: string;
```

```

}
interface IState {
  population: number;
}
const wyoming: IState = {
  name: 'Wyoming',
  capital: 'Cheyenne',
  population: 500_000
}; // ok

```

Это называется объединением деклараций. Вы наверняка встречали его ранее. Главным образом оно используется с файлами деклараций типов (глава 6). Для его поддержки уместно использовать `interface`, поскольку в декларациях типов могут быть пропуски, которые должны заполнять пользователи.

TypeScript использует слияние, чтобы получать разные типы для разных версий стандартной библиотеки JavaScript. Интерфейс `Array`, к примеру, определен в `lib.es5.d.ts`. По умолчанию — это все, что вы получаете. Однако если вы добавите ES2015 к записи `lib` вашего `tsconfig.json`, то TypeScript также включит `lib.es2015.d.ts`. Это добавляет (посредством слияния) еще один интерфейс `Array` с дополнительными методами вроде `find`, характерными для ES2015. В итоге вы получаете общий тип `Array`, богатый методами.

Слияние поддерживается в обычном коде так же, как декларации. Если же вам важно, чтобы никто не мог в дальнейшем дополнять тип, то используйте `type`.

Вернемся к вопросу выбора между `type` и `interface`. Для сложных типов выбора у вас нет: необходимо использовать псевдоним типа. Но что насчет более простых типов объектов, которые могут быть представлены и другим путем? Чтобы ответить на этот вопрос, вам следует учитывать согласованность и возможное дополнение. Если вы работаете с кодом, который постоянно использует `interface`, то придерживайтесь `interface`. Если же в нем применяется `type`, используйте `type`.

В проектах, не имеющих установленного стиля, вам следует поразмыслить над возможностью дополнения. Если вы опубликуете декларации типов для API, пользователям будет удобно вставлять новые поля через `interface` при изменении API. Однако для типа, который используется внутри проекта, слияние деклараций может оказаться ошибкой, поэтому отдайте предпочтение `type`.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Постарайтесь понять различия и сходства между `type` и `interface`.
- ✓ Знайте, как прописать одни и те же типы с помощью обоих видов синтаксиса.
- ✓ При выборе предпочтительного способа для вашего проекта учтите установленный стиль и то, насколько полезной окажется возможность дополнения.

ПРАВИЛО 14. Операции типов и обобщения сокращают повторы

Этот скрипт выводит измерения площади поверхностей и объемы нескольких цилиндров:

```
console.log('Cylinder 1 x 1 ',
  'Surface area:', 6.283185 * 1 * 1 + 6.283185 * 1 * 1,
  'Volume:', 3.14159 * 1 * 1 * 1);
console.log('Cylinder 1 x 2 ',
  'Surface area:', 6.283185 * 1 * 1 + 6.283185 * 2 * 1,
  'Volume:', 3.14159 * 1 * 2 * 1);
console.log('Cylinder 2 x 1 ',
  'Surface area:', 6.283185 * 2 * 1 + 6.283185 * 2 * 1,
  'Volume:', 3.14159 * 2 * 2 * 1);
```

Выглядит ли этот код неудобным? Скорее всего, да, так как он состоит из повторов, которые спровоцировали ошибку (вы ее заметили?). Гораздо лучше внести в него общие функции, константу и цикл:

```
const surfaceArea = (r, h) => 2 * Math.PI * r * (r + h);
const volume = (r, h) => Math.PI * r * r * h;
for (const [r, h] of [[1, 1], [1, 2], [2, 1]]) {
  console.log(
    `Cylinder ${r} x ${h}`,
    `Surface area: ${surfaceArea(r, h)}`,
    `Volume: ${volume(r, h)}`);
}
```

Следуйте принципу DRY: Don't Repeat Yourself (не повторяйтесь) — универсальному совету разработчикам. Те, кто привык избегать повторов в коде, легко смогут отказаться от них при работе с типами:

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

interface PersonWithBirthDate {
    firstName: string;
    lastName: string;
    birth: Date;
}
```

Повторение в типах влечет за собой множество проблем. Например, вы решили добавить опциональное поле `middleName` к `Person`. Теперь `Person` и `PersonWithBirthDate` разделились.

Повторение распространено в типах, например, потому что механизмы вынесения общих разделяемых шаблонов не так распространены. Каков эквивалент вынесения вспомогательной функции в системе типов? Освоив отображение между типами, вы сможете привнести принцип DRY в процесс определения типов.

Простейший способ уменьшить повторяемость — именовать типы. Вместо того чтобы прописывать метрическую функцию, подобную следующей:

```
function distance(a: {x: number, y: number}, b: {x: number, y: number}) {
    return Math.sqrt(Math.pow(a.x - b.x, 2) + Math.pow(a.y - b.y, 2));
}
```

присвойте имя типу и используйте его:

```
interface Point2D {
    x: number;
    y: number;
}
function distance(a: Point2D, b: Point2D) { /* ... */ }
```

Так же работает введение константы. Дублированные типы не всегда легко поддаются обнаружению. Иногда они скрыты в синтаксисе. К примеру, когда несколько функций разделяют одну сигнатуру типа:

```
function get(url: string, options: RequestOptions):
    Promise<Response> { /* ... */ }
function post(url: string, options: RequestOptions):
    Promise<Response> { /* ... */ }
```

тогда вы можете вынести проименованный тип для этой сигнатуры (правило 12):

```

type HTTPFunction = (url: string, options: RequestOptions) =>
    Promise<Response>;
const get: HTTPFunction = (url, options) => { /* ... */ };
const post: HTTPFunction = (url, options) => { /* ... */ };

```

Больше информации см. в правиле 12.

В примере с `Person` / `PersonWithDate` вы можете устранить повтор, выполнив расширение одного интерфейса другим:

```

interface Person {
    firstName: string;
    lastName: string;
}

interface PersonWithBirthDate extends Person {
    birth: Date;
}

```

Теперь вам нужно лишь дописать дополнительные поля. Если два интерфейса разделяют подмножества полей, вы можете вынести базовый класс с общими полями, как в случае с написанием `PI` и `2*PI` вместо `3.141593` и `6.283185`. Вы также можете использовать оператор пересечения (`&`) для расширения существующего типа, но это менее распространенный вариант:

```

type PersonWithBirthDate = Person & { birth: Date };

```

Такой подход наиболее полезен, когда вы хотите добавить свойства в тип объединения, не поддающийся расширению (правило 13).

Можно пойти и другим путем. Допустим, у вас есть тип `State`, который представляет состояние всего приложения, и тип `TopNavState`, представляющий только его часть:

```

interface State {
    userId: string;
    pageTitle: string;
    recentFiles: string[];
    pageContents: string;
}

interface TopNavState {
    userId: string;
    pageTitle: string;
    recentFiles: string[];
}

```

Вместо того чтобы формировать State посредством расширения TopNavState, можно определить TopNavState в качестве подмножества полей в State и сохранить единый интерфейс, определяющий состояние всего приложения.

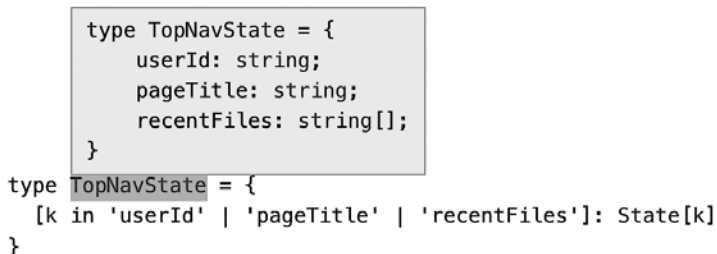
Можно устранить дублирование в типах свойств через указание на State:

```
type TopNavState = {
  userId: State['userId'];
  pageTitle: State['pageTitle'];
  recentFiles: State['recentFiles'];
};
```

Несмотря на то что код стал длиннее, изменение в типе pageTitle в State отразилось в TopNavState. Однако здесь все еще присутствуют повторы. Используйте отображенный тип:

```
type TopNavState = {
  [k in 'userId' | 'pageTitle' | 'recentFiles']: State[k]
};
```

Наведение курсора на TopNavState покажет, что это определение, по сути, такое же, как и предыдущее (рис. 2.10).



```
type TopNavState = {
  userId: string;
  pageTitle: string;
  recentFiles: string[];
}

type TopNavState = {
  [k in 'userId' | 'pageTitle' | 'recentFiles']: State[k]
}
```

Рис. 2.10. Показ расширенной версии сопоставленного типа в текстовом редакторе. То же самое, что и первоначальное определение, но без повторов

Отображенные типы в системе типов выступают в роли эквивалента итерации полей массива. Этот паттерн настолько распространен, что даже является частью стандартной библиотеки, где имеет название Pick:

```
type Pick<T, K> = { [k in K]: T[k] };
```

(Это определение недостаточно полное.) Используйте его таким образом:

```
type TopNavState = Pick<State, 'userId' | 'pageTitle' | 'recentFiles'>;
```

`Pick` является примером *обобщенного* типа. Если продолжить аналогию с дублированием кода, использование `Pick` является эквивалентом вызова функции. `Pick` берет два типа — `T` и `K` и возвращает третий. Во многом как и функция, которая берет два значения и возвращает третье.

Еще одна форма повторов может возникнуть в тип-суммах. Что, если вам нужен тип только для тега?

```
interface SaveAction {
    type: 'save';
    // ...
}
interface LoadAction {
    type: 'load';
    // ...
}
type Action = SaveAction | LoadAction;
type ActionType = 'save' | 'load'; // повторяющиеся типы!
```

Вы можете определить `ActionType` без повторов с помощью указания на объединение `Action`:

```
type ActionType = Action['type']; // тип "save" | "load"
```

По мере добавления новых типов в объединение `Action` `ActionType` будет встраивать их автоматически. Этот тип, в отличие от `Pick`, не выдаст интерфейс со свойством `type`:

```
type ActionRec = Pick<Action, 'type'>; // {type: "save" | "load"}
```

Если вы определяете класс, который может быть инициализирован и затем обновлен, то тип для параметра метода обновления будет опционально включать большую часть тех же параметров, что и конструктор:

```
interface Options {
    width: number;
    height: number;
    color: string;
    label: string;
}
interface OptionsUpdate {
    width?: number;
    height?: number;
    color?: string;
    label?: string;
}
```

```
class UIWidget {
  constructor(init: Options) { /* ... */ }
  update(options: OptionsUpdate) { /* ... */ }
}
```

Вы можете сформировать `OptionsUpdate` из `Options`, используя отображенный тип и `keyof`:

```
type OptionsUpdate = {[k in keyof Options]?: Options[k]};
```

`keyof` возьмет тип и выдаст объединения типов его ключей:

```
type OptionsKeys = keyof Options;
// тип "width" | "height" | "color" | "label"
```

Отображенный тип (`[k в keyof Options]`) произведет итерацию и найдет соответствующие значения типа в `Options`. Знак `?` сделает каждое свойство выборочным. Этот шаблон очень популярен и предусмотрен в стандартной библиотеке в виде `Partial`:

```
class UIWidget {
  constructor(init: Options) { /* ... */ }
  update(options: Partial<Options>) { /* ... */ }
}
```

Вам также может потребоваться определить тип, который совпадает с формой значения:

```
const INIT_OPTIONS = {
  width: 640,
  height: 480,
  color: '#00FF00',
  label: 'VGA',
};
interface Options {
  width: number;
  height: number;
  color: string;
  label: string;
}
```

Воспользуйтесь `typeof`:

```
type Options = typeof INIT_OPTIONS;
```

В результате будет намеренно вызван JavaScript-оператор выполнения `typeof`, но он будет работать на уровне типов TypeScript и окажется гораздо более

точным (правило 8). Будьте осторожны: лучше сперва определить типы, а затем объявить, что значения могут быть им назначены. Это делает типы более явными и менее подверженными странностям при расширении (правило 21).

Возможно, вы захотите создать именованный тип для выведенного возвращаемого значения функции или метода:

```
function getUserInfo(userId: string) {
    // ...
    return {
        userId,
        name,
        age,
        height,
        weight,
        favoriteColor,
    };
}
// Возвращаемый тип выведен как { userId: string; name: string; age:
// number, ... }
```

Для этого подойдут условные типы (правило 50). Однако стандартная библиотека определяет обобщенные типы для подобных распространенных шаблонов, поэтому можете использовать обобщение `ReturnType`:

```
type UserInfo = ReturnType<typeof getUserInfo>;
```

Имейте в виду, что `ReturnType` работает с `typeof` *имени* функции `getUserInfo`, а не ее *значения*. Используйте этот подход с осторожностью.

Обобщенные типы являются эквивалентом функций, поддерживающих принцип DRY. Но в этой аналогии кое-чего не хватает. Через систему типов вы определяете пределы значений, которые можно отображать посредством функции: добавляете числа, а не объекты и находите область форм, а не записи базы данных. Как же определить пределы параметров в обобщенном типе?

С помощью `extends` объявите, что любой обобщенный параметр расширяет тип.

Например:

```
interface Name {
    first: string;
    last: string;
}
type DancingDuo<T extends Name> = [T, T];
```

```
const couple1: DancingDuo<Name> = [
  {first: 'Fred', last: 'Astaire'},
  {first: 'Ginger', last: 'Rogers'}
]; // ok
const couple2: DancingDuo<{first: string}> = [
  // ~~~~~
  // Свойство 'last' отсутствует в типе
  // '{ first: string; }', но необходимо в типе
  // 'Name'.
  {first: 'Sonny'},
  {first: 'Cher'}
];
```

Дело в том, что `{first: string}` не расширяет `Name`, отсюда и ошибка.



На текущем этапе TypeScript требует, чтобы вы всегда прописывали обобщенный параметр в декларации. Написание `DancingDuo` вместо `DancingDuo<Name>` не сработает. Чтобы TypeScript сделал вывод типа обобщенного параметра, используйте внимательно набранную функцию тождественности:

```
const dancingDuo = <T extends Name>(x: DancingDuo<T>) => x;
const couple1 = dancingDuo([
  {first: 'Fred', last: 'Astaire'},
  {first: 'Ginger', last: 'Rogers'}
]);
const couple2 = dancingDuo([
  {first: 'Bono'},
  // ~~~~~
  {first: 'Prince'}
  // ~~~~~
  // Свойство 'last' отсутствует в типе
  // '{ first: string; }', но необходимо в типе 'Name'.
]);
```

Частные случаи этой ситуации показаны в правиле 26.

Вы можете использовать `extends` для завершения ранее начатого определения `Pick`. Если вы прогоните изначальную версию через модуль проверки типов, то получите ошибку:

```
type Pick<T, K> = {
  [k in K]: T[k]
  // ~ Тип 'K' не может быть назначен для типа 'string | number |
  // symbol'.
};
```

K не имеет ограничений в этом типе и оказывается слишком широким. Ему нужно быть похожим на что-то вроде индекса, а именно `string | number | symbol`. Но вы можете получить более узкое значение: K на самом деле должен быть подмножеством ключей T, а именно `keyof T`:

```
type Pick<T, K extends keyof T> = {  
  [k in K]: T[k]  
}; // ok
```

Воспринимайте типы как наборы значений (правило 7). В данном случае это позволит прочитать `extends` (расширяет) как `subset of` (подмножество).

В работе с абстрактными типами постарайтесь не упустить из виду цель: выбор рабочих программ вместо нерабочих. В данном случае передача `Pick` неверного ключа вызовет ошибку:

```
type FirstLast = Pick<Name, 'first' | 'last'>; // ok  
type FirstMiddle = Pick<Name, 'first' | 'middle'>;  
// ~~~~~  
// Тип "'middle'" не может быть назначен для  
// типа "'first' | 'last'".
```

Повторяемость равно плоха как для пространства типов, так и для пространства значений. Конструкции, которые вы используете во избежание повторов в пространстве типов, не так известны, как те, что используются в программной логике. Однако они однозначно заслуживают внимания. Не повторяйтесь!

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Принцип DRY (не повторяйтесь) применяется к типам в той же степени, что и к логике.
- ✓ Именуйте типы, вместо того чтобы их повторять. Используйте `extends`, чтобы избежать повторов полей в интерфейсах.
- ✓ Ознакомьтесь с инструментами, представленными в TypeScript для отображения между типами. Они включают в себя `keyof`, `typeof`, использование индексов и отображенные типы.
- ✓ Обобщенные типы являются эквивалентом функций, поддерживающих принцип DRY. Используйте их для отображения между типами вместо повторения типов. `extends` ограничит обобщенные типы.
- ✓ Ознакомьтесь с обобщенными типами, определенными в стандартной библиотеке (`Pick`, `Partial` и `ReturnType`).

ПРАВИЛО 15. Используйте сигнатуры индексов для динамических данных

JavaScript обладает удобным синтаксисом для создания объектов:

```
const rocket = {  
  name: 'Falcon 9',  
  variant: 'Block 5',  
  thrust: '7,607 kN',  
};
```

Объекты в JavaScript отображают строковые ключи в значения любого типа. TypeScript позволяет производить подобное гибкое отображение при помощи назначения для типа *сигнатуры индекса*:

```
type Rocket = {[property: string]: string};  
const rocket: Rocket = {  
  name: 'Falcon 9',  
  variant: 'v1.0',  
  thrust: '4,940 kN',  
}; // ok
```

`{[property: string]: string}` является сигнатурой индекса и определяет три параметра:

Имя ключа

Необходимо только для документации и не применяется для модуля проверки типов.

Тип ключа

Должен быть некоторой комбинацией `string`, `number` или `symbol`, но в основном вам понадобится использовать `string` (правило 16).

Тип значений

Может быть чем угодно.

Поскольку сигнатура индекса будет проходить проверку типов, нужно учитывать некоторые минусы:

- Она позволяет использовать любые ключи, включая неверные. Даже если вы напишете `Name` вместо `name`, тип `Rocket` будет по-прежнему считаться рабочим.

- Она не требует наличия каких-либо специфичных ключей. Даже при {} тип `Rocket` будет действующим.
- Она не может иметь различные типы для различных ключей. `Thrust`, вероятно, должен быть `number`, но не `string`.
- Языковые службы TypeScript не помогут вам прописывать подобные типы. В то время как вы печатаете `name:`, вам не будет предложена автоподстановка, так как ключ может оказаться любым.

Коротко говоря, сигнатуры индексов не отличаются точностью. Почти всегда можно найти лучшую альтернативу. В этом случае `Rocket` однозначно должен быть прописан как `interface`:

```
interface Rocket {
  name: string;
  variant: string;
  thrust_kN: number;
}
const falconHeavy: Rocket = {
  name: 'Falcon Heavy',
  variant: 'v1',
  thrust_kN: 15_200
};
```

`thrust_kN` является `number` и TypeScript проведет проверку наличия всех необходимых полей. При этом будут доступны все полезные языковые сервисы TypeScript: автоподстановка, переход к определению и переименование.

Так зачем же использовать сигнатуры индексов? Для динамических данных. Они могут применяться в CSV-файлах, где имеется строковый заголовок, и нужно представить строки данных в виде объектов, отображающих имена столбцов в значения:

```
function parseCSV(input: string): {[columnName: string]: string}[] {
  const lines = input.split('\n');
  const [header, ...rows] = lines;
  return rows.map(rowStr => {
    const row: {[columnName: string]: string} = {};
    rowStr.split(',').forEach((cell, i) => {
      row[header[i]] = cell;
    });
    return row;
  });
}
```

Если вы не знаете наперед, какие у столбцов будут имена, а пользователи будут лучше понимать значения колонок в конкретном контексте, сигнатура индекса будет уместна:

```
interface ProductRow {
  productId: string;
  name: string;
  price: string;
}

declare let csvData: string;
const products = parseCSV(csvData) as unknown as ProductRow[];
```

Чтобы столбцы точно оправдали ваши ожидания, можете добавить `undefined` к значению типа.

```
function safeParseCSV(
  input: string
): {[columnName: string]: string | undefined}[] {
  return parseCSV(input);
}
```

Но тогда каждое обращение потребует проверки:

```
const rows = parseCSV(csvData);
const prices: {[product: string]: number} = {};
for (const row of rows) {
  prices[row.productId] = Number(row.price);
}
const safeRows = safeParseCSV(csvData);
for (const row of safeRows) {
  prices[row.productId] = Number(row.price);
  // ~~~~~ Тип 'undefined' не может быть использован
  // в качестве типа индекса.
}
```

Конечно, это усложнит работу с типом. Поэтому выбор `undefined` остается за вами.

Если тип имеет ограниченный набор доступных полей, то не следует моделировать его с помощью сигнатуры индекса. Например, если известно, что данные будут иметь ключи вроде A, B, C, D, но при этом вы не знаете, сколько именно их будет, то смоделируйте тип с помощью либо опциональных полей, либо объединения:

```
interface Row1 { [column: string]: number } // слишком обширно
interface Row2 { a: number; b?: number; c?: number; d?: number } // лучше
type Row3 =
  | { a: number; }
  | { a: number; b: number; }
  | { a: number; b: number; c: number; }
  | { a: number; b: number; c: number; d: number };
```

Последняя форма является наиболее точной, но менее удобной в работе.

Если сложность в использовании сигнатуры индекса кроется в чрезмерной обширности `string`, то можно обратиться к ряду альтернатив.

Одна из них — использование `Record`. Это обобщенный тип, дающий типу ключа гибкость. В частности, он позволяет передавать подмножества `string`:

```
type Vec3D = Record<'x' | 'y' | 'z', number>;
// тип Vec3D = {
//   x: number;
//   y: number;
//   z: number;
// }
```

Еще один способ — это использование отображенного типа, с помощью которого можно применять разные типы для разных ключей.

```
type Vec3D = {[k in 'x' | 'y' | 'z']: number};
// так же, как и выше
type ABC = {[k in 'a' | 'b' | 'c']: k extends 'b' ? string : number};
// тип ABC = {
//   a: number;
//   b: string;
//   c: number;
// }
```

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Используйте сигнатуры индексов, когда свойства объекта не могут быть известны до момента выполнения программы. Например, если вы загружаете их из CSV-файла.
- ✓ Рассмотрите вариант добавления `undefined` к типу значения сигнатуры индекса для более безопасного обращения.
- ✓ По возможности старайтесь использовать не сигнатуры индексов, а более точные типы: `interface`, `Record` или отображенные типы.

ПРАВИЛО 16. В качестве сигнатур индексов используйте массивы, кортежи и `ArrayLike`, но не `number`

JavaScript известен своими причудами. К ним относятся неявные приведения типов:

```
> "0" == 0
true
```

которых можно избежать, используя `===` и `!==`.

Модели объектов в JavaScript также имеют свои странности (правило 10), которые нужно понять, так как некоторые из них моделируются системой типов TypeScript.

Что есть объект? В JavaScript это набор пар ключ — значение. Ключами обычно выступают строки (в ES2015 и позднее они могут быть `symbol`). Значения же могут быть чем угодно.

Здесь больше ограничений, чем в других языках программирования. В JavaScript отсутствует обозначение хешируемых объектов, которое есть в Java или Python. Если вы попытаетесь использовать более сложный объект в качестве ключа, то он преобразуется в строку посредством вызова его метода `toString`.

```
> x = {}
{}
> x[[1, 2, 3]] = 2
2
> x
{ '1,2,3': 1 }
```

В частности, `number` не может применяться в качестве ключа. Если вы попытаетесь использовать `number` как имя свойства, то при выполнении JavaScript преобразует его в строку:

```
> { 1: 2, 3: 4 }
{ '1': 2, '3': 4 }
```

Чем же в таком случае являются массивы? Объектами:

```
> typeof []
'object'
```

При этом вполне нормально использовать в них численные указатели:

```
> x = [1, 2, 3]
[ 1, 2, 3 ]
> x[0]
1
```

Преобразуются ли они в строки? Это прозвучит странно, но да. Вы можете обращаться к элементам массива, используя строковые ключи:

```
> x['1']
2
```

Если вы используете `Object.keys` для вывода списка ключей массива, то получите в ответ строки:

```
> Object.keys(x)
[ '0', '1', '2' ]
```

TypeScript пытается сделать этот процесс более разумным, допуская численные ключи и проводя различие между ними и строками. Если вы заглянете глубже в декларации типов для `Array` (массива) (правило 6), то обнаружите в `lib.es5.d.ts` следующее:

```
interface Array<T> {
  // ...
  [n: number]: T;
}
```

Здесь чистая фикция: строковые ключи принимаются во время выполнения в связи с тем, что так велит стандарт ECMAScript. Это помогает обнаруживать ошибки:

```
const xs = [1, 2, 3];
const x0 = xs[0]; // ok
const x1 = xs['1'];
           // ~~~ Элемент неявно имеет тип 'any',
           //      так как выражение индекса не относится к типу 'number'.

function get<T>(array: T[], k: string): T {
  return array[k];
           // ~ Элемент неявно имеет тип 'any',
           //      так как выражение индекса не относится к типу 'number'.
}
```

Но важно помнить, что это фикция. Как и все другие аспекты системы типов TypeScript, она удаляется при выполнении (правило 3). Это значит, что конструкции вроде `Object.keys` вернут строки:

```
const keys = Object.keys(xs); // тип string[]
for (const key in xs) {
    key; // тип string
    const x = xs[key]; // тип number
}
```

Удивительно, что последнее обращение работает, так как `string` не может быть назначен для `number`. Лучше всего представить это как прагматичное отклонение в стиле итерации по массивам, что является обыденным в JavaScript. Это не самый хороший способ производить цикл по массивам. Если вам не важен индекс, можете использовать `for...of`:

```
for (const x of xs) {
    x; // тип number
}
```

Если же индекс важен, то можно получить его в виде `number` с помощью `Array.prototype.forEach`:

```
xs.forEach((x, i) => {
    i; // тип number
    x; // тип number
});
```

Чтобы осуществить раннее прерывание цикла, лучше использовать цикл в стиле C `for(;;)`:

```
for (let i = 0; i < xs.length; i++) {
    const x = xs[i];
    if (x < 0) break;
}
```

Если типы не кажутся вам убедительными, то оцените качество: в большинстве браузеров и движках JavaScript циклы `for...in` по массивам оказываются на несколько порядков медленнее, чем `for...of` или `for(;;)`.

Сигнатура индекса `number` подразумевает, что вводимые вами значения должны быть `number` (за исключением случаев с циклами `for...in`), но на выходе вы получите `string`.

Это кажется запутанным, потому что так и есть. По большому счету, нет большой необходимости использовать `number` в качестве сигнатуры индекса типа вместо `string`. Если вы хотите определить нечто, что будет проиндексировано посредством чисел, то, возможно, вам нужно использовать массив или кортеж. Применение `number` в качестве индекса может породить дальнейшее заблуждение, что численные свойства приветствуются в JavaScript.

Если же вы противитесь использованию массива, так как он имеет много других свойств (наследованных от прототипов), которые вам не нужны (вроде `push` или `concat`), то это похвально — значит, вы мыслите структурно (правило 4). Если же вы действительно хотите применять кортежи любой длины или любую подобную массивную конструкцию, то вам подойдет тип `ArrayLike`:

```
function checkedAccess<T>(xs: ArrayLike<T>, i: number): T {
  if (i < xs.length) {
    return xs[i];
  }
  throw new Error(`Attempt to access ${i} which is past end of array.`)
}
```

У него есть только длина и численная сигнатура индекса. По возможности используйте такой вариант. Но помните, что ключи по-прежнему являются строками.

```
const tupleLike: ArrayLike<string> = {
  '0': 'A',
  '1': 'B',
  length: 2,
}; // ok
```

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Массивы — это объекты, поэтому их ключи являются строками, а не числами. Применение конструкции `number` в качестве сигнатуры индекса — это возможность TypeScript, внедренная для облегчения поиска ошибок.
- ✓ В качестве сигнатуры индекса стремитесь использовать `Array`, кортеж или тип `ArrayLike`, но не `number`.

ПРАВИЛО 17. Используйте readonly против ошибок, связанных с изменяемостью

Вот код, выводящий треугольные числа (1, 1+2, 1+2+3 и т. д.):

```
function printTriangles(n: number) {
    const nums = [];
    for (let i = 0; i < n; i++) {
        nums.push(i);
        console.log(arraySum(nums));
    }
}
```

Выглядит он достаточно простым. Однако вот что происходит при его запуске:

```
> printTriangles(5)
0
1
2
3
4
```

Предполагалось, что `arraySum` не модифицирует `num`. Вот мой вариант реализации:

```
function arraySum(arr: number[]) {
    let sum = 0, num;
    while ((num = arr.pop()) !== undefined) {
        sum += num;
    }
    return sum;
}
```

Эта функция считает сумму чисел в массиве. Но она имеет и побочный эффект — опустошает массив. Для TypeScript это не проблема, так как массивы в JavaScript являются изменяемыми. Чтобы `arraySum` не изменял массив, используйте модификатор типа `readonly` (только чтение) Вот что он делает:

```
function arraySum(arr: readonly number[]) {
    let sum = 0, num;
    while ((num = arr.pop()) !== undefined) {
        // ~~~ 'pop' не существует в типе 'readonly number[]'.
        sum += num;
    }
    return sum;
}
```

Это сообщение об ошибке стоит отдельного внимания. `readonly number[]`: это *тип*, и он отличен от `number[]` в нескольких аспектах:

- Вы можете проводить чтение его элементов, но не запись в них.
- Вы можете прочесть его длину (`length`), но не устанавливать ее.
- Вы не можете вызывать методы вроде `pop`, которые могут повлечь изменение массива.

Так как `number[]` является более функциональным, чем `readonly number[]`, то `number[]` должен быть подтипом `readonly number[]`. По ошибке можно сделать наоборот (вспомните правило 7). Внимательно приписывайте изменяемый массив к массиву `readonly`:

```
const a: number[] = [1, 2, 3];
const b: readonly number[] = a;
const c: number[] = b;
// ~ Тип 'readonly number[]' является 'readonly' и не может быть
// назначен для изменяемого типа 'number[]'.
```

Модификатор `readonly` не имел бы существенной пользы, если бы вы могли от него избавиться, не прибегая к утверждению типа.

Когда вы объявляете параметр `readonly`, происходит следующее:

- TypeScript убеждается, что параметр не изменен в теле функции.
- Инициаторы вызова убеждаются, что функция не изменяет параметр.
- Инициаторы вызова получают возможность передать функции массив `readonly`.

При разработке в JavaScript (и TypeScript) зачастую предполагается, что функции не изменяют свои параметры, пока это не назначено явно. Как мы неоднократно увидим (правила 30 и 31), неявные представления лучше приводить в явную форму и для читателей кода, и для `tsc`.

В случае с `ArraySum` все просто: не изменяйте массив.

```
function arraySum(arr: readonly number[]) {
  let sum = 0;
  for (const num of arr) {
    sum += num;
  }
  return sum;
}
```

Теперь `printTriangles` делает то, чего вы ожидаете:

```
> printTriangles(5)
0
1
3
6
10
```

Если ваша функция не изменяет параметры, объявите их как `readonly`. Первым недостатком этого модификатора станет дальнейший вызов параметров при помощи большего набора типов (правило 29), что приведет к обнаружению неумышленных изменений.

Другой недостаток в том, что функция `readonly` требует обозначить тем же модификатором все функции, которые она вызывает. Это ведет к более четким контрактам и повышает безопасность типов. Однако если вы вызываете функцию в другой библиотеке, то у вас может не быть возможности изменить ее декларацию типа и потребуется преобразование (`param as number[]`).

Модификатор `readonly` также может обнаруживать целый класс ошибок изменения, связанных с местными переменными. Представьте, что вы прописываете инструмент для обработки романа. У вас есть последовательность строк, которые нужно объединить в абзацы, разделенные строковыми пропусками:

```
Frankenstein; or, The Modern Prometheus
by Mary Shelley
```

```
You will rejoice to hear that no disaster has accompanied the commencement
of an enterprise which you have regarded with such evil forebodings. I
arrived here yesterday, and my first task is to assure my dear sister of
my welfare and increasing confidence in the success of my undertaking.
```

```
I am already far north of London, and as I walk in the streets of
Petersburgh, I feel a cold northern breeze play upon my cheeks, which
braces my nerves and fills me with delight.1
```

¹ Мэри Шелли. «Франкенштейн, или Современный Прометей».

Ты порадуешься, когда услышишь, что предприятие, вызывавшее у тебя столь мрачные предчувствия, началось вполне благоприятно. Я прибыл сюда вчера; и спешу прежде всего заверить мою милую сестру, что у меня все благополучно и что я все более убеждаюсь в счастливом исходе моего дела.

Я нахожусь уже далеко к северу от Лондона; прохаживаясь по улицам Петербурга, я ощущаю на лице холодный северный ветер, который меня бодрит и радует.

Вот попытка¹:

```
function parseTaggedText(lines: string[]): string[][] {
  const paragraphs: string[][] = [];
  const currPara: string[] = [];

  const addParagraph = () => {
    if (currPara.length) {
      paragraphs.push(currPara);
      currPara.length = 0; // очищает строки
    }
  };

  for (const line of lines) {
    if (!line) {
      addParagraph();
    } else {
      currPara.push(line);
    }
  }
  addParagraph();
  return paragraphs;
}
```

Когда вы запускаете этот код из начала правила, то вот что получается:

```
[ [], [], [] ]
```

Что-то пошло совсем не так.

Проблема в губительном сочетании искажения (правило 24) и изменений. Искажение происходит на этой строке:

```
paragraphs.push(currPara);
```

Вместо передачи содержимого `currPara` она передает ссылку на массив. Когда вы передаете новое значение в `currPara` либо очищаете его, это изменение также отражается во входных данных в `paragraphs`, так как они указывают на тот же объект.

Другими словами, совокупный эффект этого кода:

```
paragraphs.push(currPara);
currPara.length = 0; // очистить строки
```

в том, что вы передаете новый абзац в `paragraphs`, а затем тут же его стираете.

¹ Вы бы скорее всего написали `lines.join('\n').split(/\n\n/)`, но я все объясню.

И определение `currPara.length`, и вызов `currPara.push` изменяют массив `currPara`. Чтобы пресечь такое их поведение, объявите `readonly`. Вы сразу увидите несколько ошибок реализации:

```
function parseTaggedText(lines: string[]): string[][] {
    const currPara: readonly string[] = [];
    const paragraphs: string[][] = [];

    const addParagraph = () => {
        if (currPara.length) {
            paragraphs.push(
                currPara
            );
            // ~~~~~ Тип 'readonly string[]' является 'readonly' и не может быть
            //      назначен для изменяемого типа 'string[]'.
        };
        currPara.length = 0; // очистить строки
        // ~~~~~ Невозможно назначить для 'length', так как оно
        //      является свойством readonly.
    };

    for (const line of lines) {
        if (!line) {
            addParagraph();
        } else {
            currPara.push(line);
            // ~~~~ Свойство 'push' не существует в типе 'readonly
            //      string[]'
        }
    }
    addParagraph();
    return paragraphs;
}
```

Две из этих ошибок исправит объявление `currPara` с `let` и неизменяемые методы:

```
let currPara: readonly string[] = [];
// ...
currPara = []; // очистить строки
// ...
currPara = currPara.concat([line]);
```

В отличие от `push`, `concat` возвращает новый массив, оставляя оригинальный без изменений. Изменив объявление с `const` на `let` и добавив `readonly`, вы сменили один вид изменяемости на другой. Переменная `currPara` теперь может быть изменена для указания на нужный массив, но сами массивы изменены быть не могут.

Итак, остается ошибка с `paragraphs`, и есть три способа ее исправить.

Для начала можно сделать копию `currPara`:

```
paragraphs.push([...currPara]);
```

Пока `currPara` остается `readonly`, ее копия доступна для изменений.

Или можно изменить `paragraphs` (и возвращаемый тип функции) так, чтобы эта строка стала массивом `readonly string[]`:

```
const paragraphs: (readonly string[])[] = [];
```

(Здесь играет роль группировка: `readonly string[] []` станет `readonly`-массивом изменяемых массивов, а не изменяемым массивом массивов `readonly`.)

Это сработает, но для пользователей `parseTaggedText` выглядеть будет достаточно грубо. Стоит ли вам беспокоиться о том, что они сделают с абзацами после возврата функции? Еще можно использовать утверждение, чтобы устранить свойство `readonly` у массива:

```
paragraphs.push(currPara as string[]);
```

Поскольку вы приписываете `currPara` к новому массиву в следующей инструкции, это не кажется жестким преобразованием.

Важным недостатком `readonly` может стать его ограниченность. Вы видели ее пример в `readonly string [] []`. Если у вас есть `readonly`-массив объектов, то сами объекты не являются `readonly`:

```
const dates: readonly Date[] = [new Date()];
dates.push(new Date());
// ~~~~ Свойство 'push' не существует в типе 'readonly Date[]'.
dates[0].setFullYear(2037); // ok
```

Схожая ситуация и с аналогом `readonly` для объектов — обобщением `Readonly`:

```
interface Outer {
  inner: {
    x: number;
  }
}
const o: Readonly<Outer> = { inner: { x: 0 } };
o.inner = { x: 1 };
// ~~~~ нельзя назначить для 'inner', так как оно является readonly
// свойством.
o.inner.x = 1; // ok
```

Вы можете создать двойник типа и изучить его в редакторе, чтобы понять, что конкретно происходит:

```
type T = Readonly<Outer>;
// Тип T = {
//   readonly inner: {
//     x: number;
//   };
// }
```

Важно обратить внимание на то, что модификатор `readonly` распространяется на `inner`, но не на `x`. На момент написания книги не существует встроенной поддержки глубоких типов `readonly`, но для этой цели можно создать обобщение. Сделать это правильно весьма непросто, поэтому я советую использовать библиотеку. Обобщение `DeepReadonly` в `ts-essentials` является одной из возможных реализаций.

Вы также можете прописать `readonly` в сигнатуре индекса. Таким образом вы запретите запись, но оставите разрешение на чтение:

```
let obj: {readonly [k: string]: number} = {};
// or Readonly<{[k: string]: number}>
obj.hi = 45;
// ~~ Сигнатура индекса в типе ... разрешает только чтение.
obj = {...obj, hi: 12}; // ok
obj = {...obj, bye: 34}; // ok
```

Так вы предотвратите сложности, связанные с искажениями и изменениями, задействующими объекты вместо массивов.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Если функция не модифицирует свои параметры, то их следует объявить `readonly`. Так контракт функции станет более ясным и исчезнет вероятность непредусмотренных изменений при ее реализации.
- ✓ Используйте `readonly` против ошибок, связанных с изменениями, и для обнаружения мест в коде, где изменения происходят.
- ✓ Изучайте разницу между `const` и `readonly`.
- ✓ Имейте в виду, что `readonly` ограничен.

ПРАВИЛО 18. Используйте отображенные типы для синхронизации значений

Предположим, вы пишете компонент UI, рисующий графики рассеивания. В нем есть несколько различных типов свойств, контролирующих отображение и поведение:

```
interface ScatterProps {  
  // Данные  
  xs: number[];  
  ys: number[];  
  
  // Отображение  
  xRange: [number, number]; yRange: [number, number]; color: string;  
  
  // События  
  onClick: (x: number, y: number, index: number) => void;  
}
```

Во избежание ненужной работы вы наверняка предпочтете перерисовывать график только при необходимости. Изменение данных или свойств отображения требует перерисовки, но обработчик событий — нет. Такой способ оптимизации распространен в компонентах React, где обработчик событий `Prop` может быть направлен на новую стрелочную функцию при каждой отрисовке¹.

Вот один из способов реализовать это:

```
function shouldUpdate(  
  oldProps: ScatterProps,  
  newProps: ScatterProps  
) {  
  let k: keyof ScatterProps;  
  for (k in oldProps) {  
    if (oldProps[k] !== newProps[k]) {  
      if (k !== 'onClick') return true;  
    }  
  }  
  return false;  
}
```

(Пояснение объявления `keyof` в цикле смотрите в правиле 54.)

¹ В React хук `useCallback` является еще одним способом избежать создания новых функций при каждой визуализации.

Что произойдет при добавлении нового свойства? Функция `shouldUpdate` перерисует график. Этот подход можно назвать «закрыт при отказе», или просто консервативным. Его преимущество в том, что график всегда будет правильным. Недостатком же станет слишком частая прорисовка.

Альтернативный подход «при отказе открыт» может выглядеть так:

```
function shouldUpdate(
  oldProps: ScatterProps,
  newProps: ScatterProps
) {
  return (
    oldProps.xs !== newProps.xs ||
    oldProps.ys !== newProps.ys ||
    oldProps.xRange !== newProps.xRange ||
    oldProps.yRange !== newProps.yRange ||
    oldProps.color !== newProps.color
    // (нет проверки onClick)
  );
}
```

Здесь нет ненужных перерисовок, но могут быть упущены *необходимые*. Такой подход менее распространен, поскольку он нарушает первый принцип оптимизации — «не навреди».

Все перечисленные варианты не идеальны. Вы можете условиться на будущее с коллегами принимать согласованное решение о внесении каждого нового изменения. Например, добавлять комментарий:

```
interface ScatterProps {
  xs: number[];
  ys: number[];
  // ...
  onClick: (x: number, y: number, index: number) => void;
  // Заметка: если вы добавите здесь свойство, то обновите shouldUpdate!
}
```

Вы же не думаете, что это удобно? Лучше за вас поработает модуль проверки типов. Но нужно правильно его настроить: применить отображенный тип и объект.

```
const REQUIRES_UPDATE: {[k in keyof ScatterProps]: boolean} = {
  xs: true,
  ys: true,
  xRange: true,
```

```

    yRange: true,
    color: true,
    onClick: false,
  };

  function shouldUpdate(
    oldProps: ScatterProps,
    newProps: ScatterProps
  ) {
    let k: keyof ScatterProps;
    for (k in oldProps) {
      if (oldProps[k] !== newProps[k] && REQUIRES_UPDATE[k]) {
        return true;
      }
    }
    return false;
  }
}

```

[k in keyof ScatterProps] сообщает модулю проверки, что REQUIRES_UPDATES должен иметь те же свойства, что и ScatterProps. Если в будущем вы добавите новое свойство в ScatterProps:

```

interface ScatterProps {
  // ...
  onDoubleClick: () => void;
}

```

то произойдет ошибка в определении REQUIRES_UPDATE:

```

const REQUIRES_UPDATE: {[k in keyof ScatterProps]: boolean} = {
  // ~~~~~ Свойство 'onDoubleClick' упущено в типе ...
  // ...
};

```

Удаление или переименование свойства будет вызывать похожую ошибку.

Важно то, что здесь вы использовали объект с логическими значениями. Если бы вы применили массив:

```

const PROPS_REQUIRING_UPDATE: (keyof ScatterProps)[] = [
  'xs',
  'ys',
  // ...
];

```

то пришли бы к одному из решений: «закрыт при отказе» или «при отказе открыт».

Отображенные типы идеально подходят, когда нужно, чтобы один объект имел в точности такие же свойства, как другой. По аналогии с приведенным примером, вы можете с их помощью настроить TypeScript на внедрение дополнительных требований к коду.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Используйте отображенные типы для синхронизации значений и типов.
- ✓ Рассмотрите их применение для инициирования выборов при добавлении новых свойств в интерфейс.

Вывод типов

Для промышленных языков программирования выражения «статически типизированный» и «строго типизированный» традиционно считались синонимичными. С, С++, Java — все они подразумевают прописывание типов. Однако академические языки никогда эти два понятия не отождествляли: языки вроде ML или Haskell на протяжении долгого времени имели сложные системы вывода типов, а за последние 10 лет подобная специфика начала проявляться и в промышленных языках. С++ обзавелся `auto`, а Java получил `var`.

TypeScript также широко применяет эту функцию. При верном использовании она может существенно снизить число аннотируемых типов, необходимых для обеспечения типобезопасности. Один из простейших способов отличить начинающего TypeScript-разработчика от опытного — это количество использованных в коде аннотаций типов. Новичок может буквально утопить свой код в их избытке.

Эта глава демонстрирует проблемы, которые могут возникнуть при аннотации типов, и способы их решения. Прочитав ее, вы сможете отличить ситуации для самостоятельного прописывания типов от случаев, в которых прописывание желательно, даже несмотря на возможность автоматического вывода.

ПРАВИЛО 19. Не засоряйте код ненужными аннотациями типов

Многие разработчики, только начинающие знакомство с TypeScript, при переносе кодовой базы с JavaScript первым делом приступают к аннотированию типов. В конце концов, TypeScript подразумевает именно работу с типами. Но при этом многие аннотации оказываются необязательными.

Объявление типов для всех переменных подрывает продуктивность и считается плохим стилем.

Не пишите:

```
let x: number = 12;
```

Вместо этого просто укажите:

```
let x = 12;
```

Если в редакторе вы наведете курсор на `x`, то увидите, что его тип был выведен как `number` (рис. 3.1).

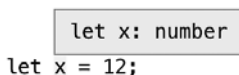


Рис. 3.1. Текстовый редактор, показывающий, что предполагаемый тип `x` — это число

Явная аннотация типов зачастую просто усложняет код, ведь тип можно проверить в редакторе.

TypeScript также способен выводить типы для более сложных объектов.

Вместо:

```
const person: {
  name: string;
  born: {
    where: string;
    when: string;
  };
  died: {
    where: string;
    when: string;
  }
} = {
  name: 'Sojourner Truth',
  born: {
    where: 'Swartekill, NY',
    when: 'c.1797',
  },
  died: {
    where: 'Battle Creek, MI',
    when: 'Nov. 26, 1883'
  }
};
```

Можно просто написать:

```
const person = {
  name: 'Sojourner Truth',
  born: {
    where: 'Swartekill, NY',
    when: 'c.1797',
  },
  died: {
    where: 'Battle Creek, MI',
    when: 'Nov. 26, 1883'
  }
};
```

И снова типы совпадают. Прописывание типов создает лишний шум. (Правило 21 содержит больше информации относительно вывода типов для объектных литералов.)

Что работает для объектов, уместно и для массивов. TypeScript легко определяет возвращаемый тип приведенной ниже функции, основываясь на вводных данных и производимых операциях:

```
function square(nums: number[]) {
  return nums.map(x => x * x);
}
const squares = square([1, 2, 3, 4]); // тип number[]
```

Иногда точность вывода TypeScript может превзойти ваши ожидания:

```
const axis1: string = 'x'; // тип string
const axis2 = 'y'; // тип "y"
```

"y" является более точным типом переменной `axis`. Правило 21 содержит пример того, как эта функциональная особенность может исправить ошибку типа.

Вывод типов может также помочь при рефакторинге. Допустим, у вас есть тип `Product` и функция для его записи:

```
interface Product {
  id: number;
  name: string;
  price: number;
}

function logProduct(product: Product) {
  const id: number = product.id;
  const name: string = product.name;
```

```

    const price: number = product.price;
    console.log(id, name, price);
}

```

В определенный момент вы понимаете, что ID продукта могут содержать не только цифры, но и буквы. В связи с этим вы меняете тип `id` в интерфейсе `Product`. Но так как вы ввели явную аннотацию всех переменных в `logProduct`, то в результате возникнет ошибка:

```

interface Product {
    id: string;
    name: string;
    price: number;
}

function logProduct(product: Product) {
    const id: number = product.id;
    // ~~ Тип 'string' не может быть назначен для типа 'number'.
    const name: string = product.name;
    const price: number = product.price;
    console.log(id, name, price);
}

```

Если вы исключите аннотации типов в теле функции `logProduct`, код пройдет проверку типов без модификации.

Использование деструктурирующего присваивания улучшит реализацию `logProduct` (правило 58):

```

function logProduct(product: Product) {
    const {id, name, price} = product;
    console.log(id, name, price);
}

```

В таком варианте активируется автоматический вывод типов для всех переменных. Альтернативный вариант с использованием явных аннотаций оказывается громоздким:

```

function logProduct(product: Product) {
    const {id, name, price}: {id: string; name: string; price: number } =
        product;
    console.log(id, name, price);
}

```

Тем не менее явные аннотации типов необходимы в ситуациях, где TypeScript не имеет достаточного контекста для самостоятельного их вывода. Например, при работе с параметрами функции из примера выше.

Некоторые языки могут делать вывод типов параметров, основываясь на их итоговом применении, но TypeScript чаще определяет тип переменной во время ее первого появления.

TypeScript-код включает аннотации типов для сигнатур функций и методов, но не для местных переменных, содержащихся в их телах, благодаря чему минимизируется число излишних конструкций и читатели кода фокусируются на его реализации.

Существуют также случаи, когда допустимо не проводить аннотацию типов и для параметров функции. Это касается значений по умолчанию:

```
function parseNumber(str: string, base=10) {  
    // ...  
}
```

Здесь тип `base` выведен как `number`, так как его значение по умолчанию `10`.

Обычно типы параметров могут быть выведены, если функция использована в качестве обратного вызова библиотеки с декларациями типов. В следующем примере с использованием HTTP библиотеки сервера, декларации для `request` и `response` не требуются:

```
// Не делайте так:  
app.get('/health', (request: express.Request, response:  
    express.Response) => {  
    response.send('OK');  
});  
// Делайте так:  
app.get('/health', (request, response) => {  
    response.send('OK');  
});
```

Правило 26 посвящено более глубокому рассмотрению воздействия контекста на вывод типов.

Возможны несколько случаев, в которых указание типа может быть предпочтительнее автоматического вывода.

Первый случай — это определение объектного литерала:

```
const elmo: Product = {  
    name: 'Tickle Me Elmo',  
    id: '048188 627152',  
    price: 28.99,  
};
```

Когда вы указываете тип в подобном определении, то активируется проверка лишних свойств (правило 11). Это может помочь обнаружить ошибки, особенно в типах с опциональными полями ввода.

Помимо этого вырастут шансы обнаружить ошибки в правильном месте. Если же вы опустите аннотацию типа, то ошибка в определении объекта выльется в ошибку типа не по месту определения, а по месту применения:

```
const furby = {
  name: 'Furby',
  id: 630509430963,
  price: 35,
};
logProduct(furby);
// ~~~~~ Аргумент .. не может быть назначен для параметра
//       типа 'Product'.
//       Типы свойства 'id' несовместимы.
//       Тип 'number' не может быть назначен для типа 'string'.
```

Сделав аннотацию, вы получаете более точное указание на место ошибки:

```
const furby: Product = {
  name: 'Furby',
  id: 630509430963,
// ~ Тип 'number' не может быть назначен для типа 'string'.
  price: 35,
};
logProduct(furby);
```

То же самое касается и возвращаемого типа функции. Можно сделать его аннотацию, чтобы гарантированно избежать проникновения ошибок реализации в использование функции.

Например, у вас есть функция, получающая котировки акций:

```
function getQuote(ticker: string) {
  return fetch(`https://quotes.example.com/?q=${ticker}`)
    .then(response => response.json());
}
```

Вы решаете ввести кэширование, чтобы избежать дублирования сетевых запросов:

```
const cache: {[ticker: string]: number} = {};
function getQuote(ticker: string) {
```

```

    if (ticker in cache) {
        return cache[ticker];
    }
    return fetch(`https://quotes.example.com/?q=${ticker}`)
        .then(response => response.json())
        .then(quote => {
            cache[ticker] = quote;
            return quote;
        });
}

```

Возникает потребность возвращать `Promise.resolve(cache[ticker])`, чтобы `getQuote` всегда возвращала `Promise`. И она приводит к ошибке... в коде, вызывающем `getQuote`, а не в самой `getQuote`:

```

getQuote('MSFT').then(considerBuying);
// ~~~~ Свойство 'then' не существует в типе
//       'number | Promise<any>'.
//       Свойство 'then' не существует в типе 'number'.

```

Если бы вы аннотировали возвращаемый тип (`Promise<number>`), то ошибка была бы обнаружена в правильном месте:

```

const cache: {[ticker: string]: number} = {};
function getQuote(ticker: string): Promise<number> {
    if (ticker in cache) {
        return cache[ticker];
        // ~~~~~ Тип 'number' не может быть назначен
        //       для типа 'Promise<number>'.
    }
    // ...
}

```

Когда вы аннотируете возвращаемый тип, то предотвращаете обнаружение ошибок реализации под видом ошибок кода пользователя. (Правило 25 посвящено асинхронным функциям, которые позволяют избежать подобных ошибок с `Promise`.)

Прописывание возвращаемого типа поможет понять функцию, так как вы будете знать начальные и конечные типы до ее выполнения. Реализация может изменяться, а контракт функции (ее сигнатура типа) изменению не подвержен, как в разработке через предварительное тестирование. Указание полной сигнатуры типа позволит получить желаемую функцию, а не ту, которая будет определена реализацией как наиболее подходящая.

Также аннотирование возвращаемых значений оправданно при желании использовать именованный тип. Вы можете предпочесть не прописывать тип для следующей функции:

```
interface Vector2D { x: number; y: number; }  
function add(a: Vector2D, b: Vector2D) {  
    return { x: a.x + b.x, y: a.y + b.y };  
}
```

TypeScript выводит возвращаемый тип как `{ x: number; y: number; }`. Он подходит для `Vector2D`, но может удивить других пользователей тем, что `Vector2D` является типом ввода, а не вывода (рис. 3.2).



```
add(a: Vector2D, b: Vector2D): { x: number; y: number; }  
add()
```

Рис. 3.2. Параметры функции `add` имеют именованные типы, а выводимое значение нет

Если вы сделаете аннотацию, то отображение станет проще. Также если прописать документацию для типа (правило 48), то он будет ассоциирован и с возвращаемым типом. Чем сложнее выводимый возвращаемый тип, тем полезнее именовать его.

Если вы пользуетесь линтером, то правило `eslint no-inferable-types` поможет вам убедиться, что все использованные аннотации типов являются действительно необходимыми.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Избегайте аннотирования типов, когда TypeScript может сделать их вывод.
- ✓ Код должен содержать аннотации типов для сигнатур функций и методов, но не для местных переменных в их телах.
- ✓ Чтобы предотвратить ошибки реализации в коде пользователя, попробуйте применить явные аннотации для объектных литералов и возвращаемых типов функций, даже если они могут быть выведены.

ПРАВИЛО 20. Для разных типов — разные переменные

В JavaScript можно повторно использовать переменную для другой задачи, назначив ей значение другого типа:

```
let id = "12-34-56";
fetchProduct(id); // ожидается строка

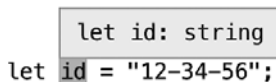
id = 123456;
fetchProductBySerialNumber(id); // ожидается строка
```

В TypeScript при таком назначении возникнет сразу две ошибки:

```
let id = "12-34-56";
fetchProduct(id);

id = 123456;
// ~~ '123456' не может быть назначено для типа 'string'.
fetchProductBySerialNumber(id);
// ~~ Аргумент типа 'string' не может быть
//    назначен параметру типа 'number'.
```

Наведение курсора на первый `id` в редакторе пояснит, в чем дело (рис. 3.3).



```
let id: string
let id = "12-34-56";
```

Рис. 3.3. Предполагаемый тип идентификатора — строка

Основываясь на значении `"12-34-56"`, TypeScript вывел тип `id` как `string`. Нельзя назначать `number` для `string` — отсюда и ошибка.

В TypeScript может изменяться значение переменной, но не ее тип. Единственный распространенный случай изменения типа — это его сужение (правило 22). Однако уменьшенный тип не может включать новые значения. Несмотря на то что все же существует несколько важных исключений (правило 41), это всего лишь исключения, но не правила.

Как же понимание этого принципа поможет исправить приведенный пример? Для того чтобы избежать изменения, тип `id` должен быть достаточно широким для охвата как `string`, так и `number`. Это и есть определение типа объединения `string|number`:

```
let id: string|number = "12-34-56";
fetchProduct(id);

id = 123456; // ok
fetchProductBySerialNumber(id); // ok
```

Ошибки устранены. Интересно, что TypeScript смог определить, что `id` является `string` в первом вызове и `number` — во втором. Он произвел сужение типа объединения, основываясь на назначении.

Несмотря на то что тип объединения сработал в этом варианте, далее он может вызвать сложности, так как придется проверять, чем являются переменные, перед их применением.

Лучшим решением будет ввести новую переменную:

```
const id = "12-34-56";
fetchProduct(id);
const serial = 123456; // ok
fetchProductBySerialNumber(serial); // ok
```

В предыдущем варианте первый и второй `id` не имели семантической связи друг с другом. Их связывало только использование той же переменной. Такой код трудно воспринимается и машиной, и человеком.

Вариант с двумя переменными лучше по нескольким причинам:

- Он разделяет два несвязанных понятия (ID и серийный номер).
- Он позволяет использовать разные имена переменных.
- Он повышает качество вывода типов, в связи с чем уже не требуется делать их аннотацию.
- В итоге получаются более простые типы (`string` и `number`, а не `string|number`).
- Он позволяет объявлять переменные через `const` вместо `let`, что упрощает их понимание.

Постарайтесь избежать переменных, изменяющих свой тип, и выбирайте разные имена для разных понятий.

Имейте в виду, что это не касается «теневых» переменных, как в приведенном примере:

```
const id = "12-34-56";
fetchProduct(id);
```

```
{
  const id = 123456; // ok
  fetchProductBySerialNumber(id); // ok
}
```

Теперь две разные переменные `id` имеют разные типы, которые легко отличит TypeScript, но не человек, читающий код. Поэтому переменным нужны разные имена. В некоторых командах устанавливают запрет на «теневое» применение переменных посредством правил линтера.

Мы описали скалярные величины, но все то же самое применимо и к объектам (правило 23).

СЛЕДУЕТ ЗАПОМНИТЬ:

- ✓ Может меняться значение переменной, но не ее тип.
- ✓ Во избежание путаницы не используйте имя переменной повторно для обозначения другого типа.

ПРАВИЛО 21. Контролируйте расширение типов

Как уже говорилось в правиле 7, каждая переменная при выполнении имеет единственное значение. Но в процессе статического анализа, когда TypeScript проверяет код, переменная имеет целый набор возможных значений, а именно тип. Когда вы инициализируете переменную константой, но при этом не указываете тип, модуль проверки типов определяет его сам. Он ищет целый набор возможных значений, исходя из единственного, вами указанного. В TypeScript этот процесс называется *расширением*. Его понимание поможет вам лучше разобраться в возможных ошибках и более эффективно аннотировать типы.

Предположим, вы пишете библиотеку для работы с векторами. Вы прописываете тип для 3D-вектора и функцию для получения значения любого из его компонентов:

```
interface Vector3 { x: number; y: number; z: number; }
function getComponent(vector: Vector3, axis: 'x' | 'y' | 'z') {
  return vector[axis];
}
```

Однако как только вы пытаетесь ее использовать, TypeScript указывает на ошибку:

```
let x = 'x';
let vec = {x: 10, y: 20, z: 30};
getComponent(vec, x);
// ~ Аргумент типа 'string' не может быть назначен
//   для параметра типа '"x" | "y" | "z"'.
```

Код отлично работает, так в чем же ошибка?

Тип `x` выводится как `string`, а функция `getComponent` ожидает более специфичный тип для второго аргумента. Так работает расширение, которое в данном случае привело к ошибке.

Этот процесс оказывается неоднозначным, ведь существует множество возможных типов для любого значения. Например, в следующей инструкции:

```
const mixed = ['x', 1];
```

каков должен быть тип `mixed`? Допустимо несколько вариантов:

- ☐ `('x' | 1)[]`
- ☐ `['x', 1]`
- ☐ `[string, number]`
- ☐ `readonly [string, number]`
- ☐ `(string|number)[]`
- ☐ `readonly (string|number)[]`
- ☐ `[any, any]`
- ☐ `any[]`

Не имея дополнительного контекста, TypeScript не способен понять, какой вариант является верным. Несмотря на все его возможности, он не читает мысли, и точность вывода не будет 100 %.

В первом примере TypeScript выводит тип `x` как `string`, так как допускает подобный код:

```
let x = 'x';
x = 'a';
x = 'Four score and seven years ago...';
```

Но также допустим вариант JavaScript:

```
let x = 'x';
x = /x|y|z/;
x = ['x', 'y', 'z'];
```

При выводе типа `x` как `string` TypeScript пытается уловить баланс между специфичностью и гибкостью. Он старается, чтобы тип переменной не изменялся после его объявления (правило 20), поэтому выбирает `string`, а не `string|RegExp`, `string|string[]` либо `any`.

TypeScript предоставляет несколько возможностей контроля расширения. Одна из них — это `const`. Если вы объявите переменную с `const` вместо `let`, она получит суженный тип и будет устранена ошибка в изначальном примере:

```
const x = 'x'; // тип "x"
let vec = {x: 10, y: 20, z: 30};
getComponent(vec, x); // ok
```

Поскольку `x` не может быть переназначен, TypeScript попытается вывести более узкий тип, не влекущий ошибок в нижеследующих назначениях. И так как тип строкового литерала `"x"` может быть назначен для `"x"|"y"|"z"`, то код пройдет проверку типов.

Но `const` допускает неоднозначность в случае с объектами и массивами. Пример с `mixed` демонстрирует проблему с массивами. Должен ли TypeScript вывести кортежный тип? Какой тип ему следует вывести для элементов? Схожие сложности возникают и с объектами. В JavaScript следующий код работает отлично:

```
const v = {
  x: 1,
};
v.x = 3;
v.x = '3';
v.y = 4;
v.name = 'Pythagoras';
```

При выводе варианты типов `v` будут иметь разную степень специфичности, начиная от `{readonly x: 1}` и переходя к `{x: number}` и `{[key: string]: number}` либо `object`. Алгоритм расширения TypeScript воспринимает каждый элемент объекта как назначенный с `let`. Поэтому тип `v` определится как `{x: number}`. Это позволит вам переназначить `v.x` для других чисел (не для `string`) и лишит вас возможности добавить другие свойства. (Поэтому объекты лучше создавать полностью (правило 23).)

Таким образом, последние три инструкции содержат ошибки:

```
const v = {
  x: 1,
};
v.x = 3; // ok
v.x = '3';
// ~ Тип '"3"' не может быть назначен для типа 'number'.
v.y = 4;
// ~ Свойство 'y' не существует в типе '{ x: number; }'.
v.name = 'Pythagoras';
// ~~~~ Свойство 'name' не существует в типе '{ x: number; }'.
```

И снова TypeScript пытается уловить баланс между специфичностью и гибкостью. Ему необходимо вывести такой тип, при котором будут правильно обнаруживаться ошибки без ложных срабатываний. Добивается он этого, делая вывод типа `number` для свойства, инициализированного со значением `1`.

Если вы знаете тип лучше, чем TypeScript, то существует несколько способов перенастроить его поведение. Первый — это произвести явную аннотацию типа:

```
const v: {x: 1|3|5} = {
  x: 1,
}; // Тип { x: 1 | 3 | 5; }
```

Второй способ — обеспечить дополнительный контекст для модуля проверки (правило 26). Например, передать значение в качестве параметра функции.

Третий способ заключается в утверждении `const`. Но не следует путать его с `let` и `const`, которые вводят символы в пространство значений. Эта конструкция относится исключительно к уровню типов. Взгляните на различные выведенные типы для переменных:

```
const v1 = {
  x: 1,
  y: 2,
}; // тип { x: number; y: number; }

const v2 = {
  x: 1 as const,
  y: 2,
}; // тип { x: 1; y: number; }
```

```
const v3 = {
  x: 1,
  y: 2,
} as const; // тип { readonly x: 1; readonly y: 2; }
```

Когда после значения вы прописываете `as const`, TypeScript выводит самый узкий возможный тип. При этом не происходит расширения. Чаще всего именно это и нужно для констант. Вы также можете использовать `as const` с массивами для вывода кортежного типа:

```
const a1 = [1, 2, 3]; // тип number[]
const a2 = [1, 2, 3] as const; // тип readonly [1, 2, 3]
```

Если вы получаете ошибки, которые, на ваш взгляд, связаны с расширением, то рассмотрите вариант добавления явных аннотаций типов или утверждений `const`. Инспектирование типов в редакторе поможет понять этот вопрос (правило 6).

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Постарайтесь понять, как именно TypeScript выводит тип из константы посредством ее расширения.
- ✓ Разберитесь с утверждениями уровней типов: `const`, аннотации типов, контекст и `as const`.

ПРАВИЛО 22. Старайтесь сужать типы

Сужение является процессом, обратным расширению. С его помощью TypeScript переходит от более широкого типа к более узкому. Простой пример этого процесса — проверка `null`:

```
const el = document.getElementById('foo'); // тип HTMLElement | null
if (el) {
  el // тип HTMLElement
  el.innerHTML = 'Party Time'.blink();
} else {
  el // тип null
  alert('No element #foo');
}
```

Если `e1` является `null`, то код в первой ветке не будет выполняться. TypeScript может исключить `null` из типа объединения внутри блока — провести сужение типа. Модуль проверки типов легко справляется с сужением типов в условных конструкциях вроде этой, но иногда ему может помешать наложение (правило 24).

Можно сузить тип переменной для оставшейся части блока отбросом или возвращением из ветки. Например:

```
const e1 = document.getElementById('foo'); // тип HTMLElement | null
if (!e1) throw new Error('Unable to find #foo');
e1; // теперь тип HTMLElement
e1.innerHTML = 'Party Time'.blink();
```

Существует множество способов для сужения типа. Например, использование `instanceof`:

```
function contains(text: string, search: string|RegExp) {
  if (search instanceof RegExp) {
    search // тип RegExp
    return !!search.exec(text);
  }
  search // тип string
  return text.includes(search);
}
```

или проверка свойств:

```
interface A { a: number }
interface B { b: number }
function pickAB(ab: A | B) {
  if ('a' in ab) {
    ab // тип A
  } else {
    ab // тип B
  }
  ab // тип A | B
}
```

Некоторые встроенные функции вроде `Array.isArray` также способны сужать типы:

```
function contains(text: string, terms: string|string[]) {
  const termList = Array.isArray(terms) ? terms : [terms];
  termList // тип string[]
  // ...
}
```

TypeScript хорошо справляется с отслеживанием типов через условные выражения. Не торопитесь добавлять утверждение. Например, здесь будет неверным подходом исключить `null` из типа объединения:

```
const el = document.getElementById('foo'); // тип HTMLElement | null
if (typeof el === 'object') {
  el; // тип HTMLElement | null
}
```

`typeof null` является "object" в JavaScript, и поэтому вы не исключили `null` этой проверкой. Схожие сюрпризы могут появляться с ложными примитивными значениями:

```
function foo(x?: number|string|null) {
  if (!x) {
    x; // тип string | number | null | undefined
  }
}
```

Так как и пустая строка, и `0` являются ложными, то `x` в этой ветке по-прежнему может быть `string` или `number`. TypeScript прав.

Еще один распространенный способ помочь модулю проверки в сужении типов — это добавление к ним явного тега:

```
interface UploadEvent { type: 'upload'; filename: string; contents: string }
interface DownloadEvent { type: 'download'; filename: string; }
type AppEvent = UploadEvent | DownloadEvent;

function handleEvent(e: AppEvent) {
  switch (e.type) {
    case 'download':
      e // тип DownloadEvent
      break;
    case 'upload':
      e; // тип UploadEvent
      break;
  }
}
```

Такой распространенный шаблон известен как тип-сумма, или размеченное объединение.

Если TypeScript не может выявить тип, попробуйте ввести пользовательскую функцию:

```
function isInputElement(el: HTMLElement): el is HTMLInputElement {
    return 'value' in el;
}

function getElementContent(el: HTMLElement) {
    if (isInputElement(el)) {
        el; // тип HTMLInputElement
        return el.value;
    }
    el; // тип HTMLElement
    return el.textContent;
}
```

Обеспечьте так называемую пользовательскую защиту типа. `el is HTMLInputElement`, будучи возвращаемым типом, сообщает модулю проверки, что он может сузить тип параметра, если функция вернет `true`.

Некоторые функции способны использовать защиту типа для сужения типов внутри массивов или объектов. Присмотритесь к массиву: возможно, он состоит из типов, допускающих значение `null`:

```
const jackson5 = ['Jackie', 'Tito', 'Jermaine', 'Marlon', 'Michael'];
const members = ['Janet', 'Michael'].map(
    who => jackson5.find(n => n === who)
); // тип (string | undefined)[]
```

Если вы используете `filter` для отсеивания значений `undefined`, то TypeScript этого не примет:

```
const members = ['Janet', 'Michael'].map(
    who => jackson5.find(n => n === who)
).filter(who => who !== undefined); // тип (string | undefined)[]
```

Однако он одобрит использование защиты типа:

```
function isDefined<T>(x: T | undefined): x is T {
    return x !== undefined;
}
const members = ['Janet', 'Michael'].map(
    who => jackson5.find(n => n === who)
).filter(isDefined); // тип string[]
```

Инспектирование типов в редакторе является ключом к пониманию темы сужения типов. Это позволит расширить понимание того, как работает вывод типов, понять ошибки и, как правило, иметь более продуктивные отношения с проверкой типов.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Постарайтесь понять, как именно TypeScript проводит сужение типов, основываясь на условных выражениях и других типах потока управления.
- ✓ Используйте тип-суммы, размеченные объединения и пользовательскую защиту типа для поддержки сужения.

ПРАВИЛО 23. Создавайте объекты целиком

Как объяснялось в правиле 20, значение переменной может изменяться, а вот ее тип чаще всего нет. В связи с этим некоторые паттерны JavaScript легче моделируются в TypeScript, чем другие. Поэтому лучше создавать объекты одновременно, а не по частям.

Вот один из способов создания объекта, представляющего двумерную точку в JavaScript:

```
const pt = {};  
pt.x = 3;  
pt.y = 4;
```

В TypeScript этот способ вызовет ошибку в обоих назначениях:

```
const pt = {};  
pt.x = 3;  
// ~ Свойство 'x' не существует в типе '{}'.  
pt.y = 4;  
// ~ Свойство 'y' не существует в типе '{}'.  

```

Все потому, что тип `pt` в первой строке выводится на основе его значения `{}`, а для назначения доступны только известные свойства.

Вы получите обратную проблему, если определите интерфейс `Point`:

```
interface Point { x: number; y: number; }  
const pt: Point = {};  
// ~~ В типе '{}' упущены свойства типа 'Point': x, y  
pt.x = 3;  
pt.y = 4;
```

Решением будет определить сразу весь объект:

```
const pt = {
  x: 3,
  y: 4,
}; // ok
```

При необходимости создать объект частями можно использовать утверждение типа (`as`), чтобы исключить нежелательную реакцию модуля проверки.

```
const pt = {} as Point;
pt.x = 3;
pt.y = 4; // ok
```

Но лучше создавать объект сразу полностью и использовать декларации (правило 9):

```
const pt: Point = {
  x: 3,
  y: 4,
};
```

Если вам нужно построить крупный объект из нескольких других, не делите работу на этапы:

```
const pt = {x: 3, y: 4};
const id = {name: 'Pythagoras'};
const namedPoint = {};
Object.assign(namedPoint, pt, id);
namedPoint.name;
// ~~~~ Свойство 'name' не существует в типе '{}'.

```

Вместо этого создайте крупный объект с помощью *оператора расширения* ...:

```
const namedPoint = {...pt, ...id};
namedPoint.name; // ok, тип string
```

Типобезопасный способ работы этого оператора заключается в использовании новой переменной для каждого обновления с присваиванием ей нового типа:

```
const pt0 = {};
const pt1 = {...pt0, x: 3};
const pt: Point = {...pt1, y: 4}; // ok
```

Несмотря на то что это окольный путь создания весьма простого объекта, он может пригодиться для добавления свойств, позволяя TypeScript сделать вывод нового типа. Для условного добавления свойства типобезопасным способом вы можете использовать оператор расширения объекта с `null` либо `{}`, что исключит добавления свойств:

```
declare let hasMiddle: boolean;
const firstLast = {first: 'Harry', last: 'Truman'};
const president = {...firstLast, ...(hasMiddle ? {middle: 'S'} : {})};
```

Наведя курсор на `president` в редакторе, вы увидите, что его тип выведен как тип объединения:

```
const president: {
  middle: string;
  first: string;
  last: string;
} | {
  first: string;
  last: string;
}
```

Возможно, вас удивит, что `middle` не является опциональным полем и его нельзя прочитать вне типа. Например:

```
president.middle
// ~~~~~ Свойство 'middle' не существует в типе
//       '{ first: string; last: string; }'.
```

Если вы условно добавляете несколько свойств, то объединение более точно представит набор возможных значений (правило 32). Но с опциональным полем проще работать. Создать его можно с помощью хелпера:

```
function addOptional<T extends object, U extends object>(  
  a: T, b: U | null  
) : T & Partial<U> {  
  return {...a, ...b};  
}
```

```
const president = addOptional(firstLast, hasMiddle ? {middle: 'S'} : null);  
president.middle // ok, тип string | undefined
```

Иногда вам нужно преобразовать один объект или массив в другой. В этом случае эквивалентом единовременного построения объекта будет использование встроенных функциональных конструкций либо библиотек утилит вроде `Lodash` вместо циклов (правило 27).

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Стройте объекты единовременно, а не частями. Используйте расширение типа (`...a, ...b`) для добавления свойств типобезопасным способом.
- ✓ Научитесь добавлять свойства к объекту условно.

ПРАВИЛО 24. Применяйте псевдонимы согласованно

Когда вы вводите новое имя для значения:

```
const borough = {name: 'Brooklyn', location: [40.688, -73.979]};  
const loc = borough.location;
```

создается *псевдоним* (*alias*). Изменение свойств псевдонима отразится и на оригинальном значении:

```
> loc[0] = 0;  
> borough.location  
[0, -73.979]
```

Псевдонимы делают процесс разработки компиляторов для любых языков более трудным, так как усложняют анализ потока команд. Используйте их обдуманно.

Предположим, у вас есть структура данных, представляющая многоугольник:

```
interface Coordinate {  
  x: number;  
  y: number;  
}  
  
interface BoundingBox {  
  x: [number, number];  
  y: [number, number];  
}  
  
interface Polygon {  
  exterior: Coordinate[];  
  holes: Coordinate[][];  
  bbox?: BoundingBox;  
}
```

Геометрия многоугольника определяется свойствами `exterior` и `holes`. Свойство `bbox` является необязательной оптимизацией. При желании ее можно использовать для ускорения проверки принадлежности точки многоугольнику:

```
function isPointInPolygon(polygon: Polygon, pt: Coordinate) {
  if (polygon.bbox) {
    if (pt.x < polygon.bbox.x[0] || pt.x > polygon.bbox.x[1] ||
        pt.y < polygon.bbox.y[1] || pt.y > polygon.bbox.y[1]) {
      return false;
    }
  }
  // ... более сложная проверка.
}
```

Этот код работает (и проходит проверку типов), но при этом он содержит повторы. Далее приведена попытка вынести за скобки вспомогательную переменную, чтобы уменьшить число повторов:

```
function isPointInPolygon(polygon: Polygon, pt: Coordinate) {
  const box = polygon.bbox;
  if (polygon.bbox) {
    if (pt.x < box.x[0] || pt.x > box.x[1] ||
        // ~~~
        pt.y < box.y[1] || pt.y > box.y[1]) {
      // ~~~
      return false;
    }
  }
  // ...
}
```

(Я предполагаю, что у вас включена опция `strictNullChecks`.)

Код по-прежнему работает, но откуда ошибка? Вынеся за скобки переменную `box`, вы создали псевдоним `polygon.bbox`, что нарушило анализ потока команд, который спокойно работал в первом варианте. Вы можете проинспектировать типы `box` и `polygon.bbox`, чтобы разобраться в происходящем:

```
function isPointInPolygon(polygon: Polygon, pt: Coordinate) {
  polygon.bbox // тип BoundingBox | undefined
  const box = polygon.bbox;
  box // тип BoundingBox | undefined
  if (polygon.bbox) {
    polygon.bbox // тип BoundingBox
    box // тип BoundingBox | undefined
  }
}
```

Проверка свойств уточняет тип `polygon.bbox`, но не `box`, отсюда и ошибка.

Важно использовать псевдонимы согласованно.

Внесение `box` в проверку свойств исправляет ошибку:

```
function isPointInPolygon(polygon: Polygon, pt: Coordinate) {
  const box = polygon.bbox;
  if (box) {
    if (pt.x < box.x[0] || pt.x > box.x[1] ||
        pt.y < box.y[1] || pt.y > box.y[1]) { // ok
      return false;
    }
  }
  // ...
}
```

Теперь модуль проверки типов сориентирован, но появилась сложность в чтении кода. Мы используем два имени для одного предмета: `box` и `bbox`, то есть имеем различие имен при отсутствии разницы понятий (правило 36).

Синтаксис деструктуризации помогает проводить согласование псевдонимов. Вы можете использовать его также для массивов и вложенных структур:

```
function isPointInPolygon(polygon: Polygon, pt: Coordinate) {
  const {bbox} = polygon;
  if (bbox) {
    const {x, y} = bbox;
    if (pt.x < x[0] || pt.x > x[1] ||
        pt.y < x[0] || pt.y > y[1]) {
      return false;
    }
  }
  // ...
}
```

Еще несколько замечаний:

- Этот код потребовал бы больше проверок свойств, если бы свойства `x` и `y` были опциональными, а не единым свойством `bbox`. Но мы последовали совету из правила 31.
- Опциональное свойство подошло бы `bbox`, но не `holes`, который мог бы оказаться упущенным или пустым массивом `[]`. Так возникло бы различие имен при отсутствии разницы понятий — пустой массив может обозначать "no holes".

Псевдонимы способны внести путаницу и при выполнении:

```
const {bbox} = polygon;
if (!bbox) {
  calculatePolygonBbox(polygon); // заполняет polygon.bbox
  // теперь polygon.bbox и bbox относятся к различным значениям!
}
```

Анализ потока команд TypeScript неплохо справляется с местными переменными, но в случае со свойствами будьте начеку:

```
function fn(p: Polygon) { /* ... */ }

polygon.bbox // тип BoundingBox | undefined
if (polygon.bbox) {
  polygon.bbox // тип BoundingBox
  fn(polygon);
  polygon.bbox // тип по-прежнему BoundingBox
}
```

Вызов `fn(polygon)` может вернуть к исходному состоянию `polygon.bbox`, поэтому типу безопаснее было бы снова стать `BoundingBox | undefined`. Но это привело бы к дополнительным сложностям, а именно к необходимости повторять проверку свойств при каждом вызове функции. Поэтому TypeScript делает прагматичный выбор и *ошибочно* предполагает, что функция не отменяет уточнения ее типов. Если бы вы вынесли за скобки местную переменную `bbox`, а не использовали `polygon.bbox`, то тип `bbox` остался бы точным, но при этом не смог бы иметь то же значение, что и `polygon.bbox`.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Псевдонимы могут помешать TypeScript сужать типы. Используйте их согласованно.
- ✓ Используйте деструктурирующий синтаксис, чтобы согласовать имена.
- ✓ Имейте в виду, что вызовы функций могут отменить уточнения типов для свойств. Доверяйте уточнениям местных переменных больше, чем уточнениям свойств.

ПРАВИЛО 25. Для асинхронного кода используйте функции `async` вместо обратных вызовов

В ранних версиях JavaScript моделировал асинхронное поведение с помощью обратных вызовов. Это неизбежно вело к печально известной «пирамиде судьбы»:

```
fetchURL(url1, function(response1) {
  fetchURL(url2, function(response2) {
    fetchURL(url3, function(response3) {
      // ...
      console.log(1);
    });
    console.log(2);
  });
  console.log(3);
});
console.log(4);

// Логи:
// 4
// 3
// 2
// 1
```

Как видно из логов, порядок выполнения противоположен порядку кода. Это усложняет чтение кода обратного вызова и вносит еще больше путаницы, когда вы хотите запустить запросы параллельно или сделать остановку при появлении ошибки.

В ES2015 для устранения «пирамиды судьбы» были введены промисы. `Promise` представляет то, что будет доступно в будущем (его называют будущим значением). Вот тот же код, написанный с их применением:

```
const page1Promise = fetch(url1);
page1Promise.then(response1 => {
  return fetch(url2);
}).then(response2 => {
  return fetch(url3);
}).then(response3 => {
  // ...
}).catch(error => {
  // ...
});
```

В нем содержится меньше вложений, налажен порядок выполнения, упрощены консолидация обработки ошибок и использование инструментов высшего порядка (вроде `Promise.all`).

В ES2017 для еще большего упрощения были введены ключевые слова `async` и `await`:

```
async function fetchPages() {
  const response1 = await fetch(url1);
  const response2 = await fetch(url2);
  const response3 = await fetch(url3);
  // ...
}
```

Ключевое слово `await` приостанавливает выполнение функции `fetchPages` до момента завершения каждого промиса. В рамках же функции `async` промис `await`, завершающийся отказом, выбрасывает исключение. Это позволяет использовать обычный механизм с `try` и `catch`:

```
async function fetchPages() {
  try {
    const response1 = await fetch(url1);
    const response2 = await fetch(url2);
    const response3 = await fetch(url3);
    // ...
  } catch (e) {
    // ...
  }
}
```

В случаях, когда вы пишете для ES5 или более ранних версий, компилятор TypeScript проводит необходимые преобразования, чтобы независимо от среды `async` и `await` работали.

Существует несколько причин использовать промисы или `async` и `await` вместо обратных вызовов:

- Промисы легче составлять.
- Типы легче передаются через промисы.

К примеру, чтобы вызвать страницы параллельно, можете составить промисы через `Promise.all`:

```
async function fetchPages() {
  const [response1, response2, response3] = await Promise.all([
```

```

    fetch(url1), fetch(url2), fetch(url3)
  ]);
  // ...
}

```

Использование деструктурирующего назначения вместе с `await` особенно удачно в этом контексте.

TypeScript способен вывести типы каждой из трех переменных `response` как `Response`. Аналогичный код для параллельного выполнения запросов с помощью обратных вызовов требует больше механизмов обработки и аннотации типа:

```

function fetchPagesCB() {
  let numDone = 0;
  const responses: string[] = [];
  const done = () => {
    const [response1, response2, response3] = responses;
    // ...
  };
  const urls = [url1, url2, url3];
  urls.forEach((url, i) => {
    fetchURL(url, r => {
      responses[i] = url;
      numDone++;
      if (numDone === urls.length) done();
    });
  });
}

```

Расширить этот код для добавления обработки ошибок или придания обобщенности, как в случае с `Promise.all`, будет трудно.

Вывод типов также хорошо работает с методом `Promise.race`, который завершается сразу после завершения его первого вводного промиса. Этот метод можно использовать для добавления в промисы тайм-аутов:

```

function timeout(millis: number): Promise<never> {
  return new Promise((resolve, reject) => {
    setTimeout(() => reject('timeout'), millis);
  });
}

async function fetchWithTimeout(url: string, ms: number) {
  return Promise.race([fetch(url), timeout(ms)]);
}

```

Возвращаемый тип `fetchWithTimeout` выведен как `Promise<Response>`, и аннотации типов не требуются. Это работает, потому что возвращаемый тип `Promise.race` является объединением типов его входных данных в кейсе `Promise<Response|never>`. Но объединение с `never` ведет к сокращению до `Promise<Response>`. Когда вы работаете с промисами, задействуются все механизмы вывода типов TypeScript для правильного их определения. Бывает, что вам нужно использовать непосредственно промисы, особенно когда вы делаете обертку обратного вызова API наподобие `setTimeout`. Но если у вас будет выбор, то стоит предпочесть использование `async` и `await` по двум причинам:

- Код станет более лаконичным и простым.
- Функции `async` гарантированно возвратят промисы.

Функция `async` возвращает `Promise`, даже если не применяет `await`:

```
// функция getNumber(): Promise<number>
async function getNumber() {
  return 42;
}
```

Вы также можете создавать стрелочные функции `async`:

```
const getNumber = async () => 42; // тип () => Promise<number>
```

Эквивалентом функции `async` с использованием промиса будет:

```
const getNumber = () => Promise.resolve(42); // тип () => Promise<number>
```

Возврат промиса для доступного значения помогает реализовать важное правило, гласящее, что функция должна всегда запускаться либо синхронно, либо асинхронно. Смешения происходить не должно. Например, попробуйте добавить кэш в функцию `fetchURL`:

```
// Не делайте так!
const _cache: {[url: string]: string} = {};
function fetchWithCache(url: string, callback: (text: string) => void) {
  if (url in _cache) {
    callback(_cache[url]);
  } else {
    fetchURL(url, text => {
      _cache[url] = text;
      callback(text);
    });
  }
}
```

Это выглядит как оптимизация, но для клиента использование функции усложнилось.

```
let requestStatus: 'loading' | 'success' | 'error';
function getUser(userId: string) {
  fetchWithCache(`/user/${userId}`, profile => {
    requestStatus = 'success';
  });
  requestStatus = 'loading';
}
```

Каково будет значение `requestStatus` после вызова `getUser`? Оно будет зависеть от того, кэширован ли профиль. Если нет, то `requestStatus` станет «успешным», если да, то он тоже станет «успешным», но затем обратно вернется к загрузке.

Использование `async` для обеих функций гарантирует согласованное поведение:

```
const _cache: {[url: string]: string} = {};
async function fetchWithCache(url: string) {
  if (url in _cache) {
    return _cache[url];
  }
  const response = await fetch(url);
  const text = await response.text();
  _cache[url] = text;
  return text;
}

let requestStatus: 'loading' | 'success' | 'error';
async function getUser(userId: string) {
  requestStatus = 'loading';
  const profile = await fetchWithCache(`/user/${userId}`);
  requestStatus = 'success';
}
```

Теперь `requestStatus` точно будет завершаться «успешно». Легко по случайности создать полусинхронный код, используя обратные вызовы или непосредственно промисы, но сложно ошибиться с `async`.

Обратите внимание, что если вы возвращаете промис из функции `async`, то он не будет обернут в другой промис — возвращаемый тип будет `Promise<T>`, а не `Promise<Promise<T>>`:

```
// функция getJSON(url: string): Promise<any>
async function getJSON(url: string) {
```

```
const response = await fetch(url);
const jsonPromise = response.json(); // тип Promise<any>
return jsonPromise;
}
```

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Предпочитайте промисы обратным вызовам для улучшения сочетаемости и движения типов.
- ✓ По возможности стремитесь использовать `async` и `await` вместо непосредственно промисов. Таким образом получится более лаконичный и простой код, исключающий многие виды ошибок.
- ✓ Если функция возвращает промис, объявите ее `async`.

ПРАВИЛО 26. Используйте контекст при выводе типов

TypeScript выводит типы не только на основе значений — он также учитывает контекст, в котором они появляются. Иногда эта схема преподносит сюрпризы.

В JavaScript вы можете выделить выражение в отдельную константу, не меняя поведения кода (до тех пор пока вы не меняете порядок его выполнения). Следующие две инструкции будут эквивалентом друг друга:

```
// строковая форма
setLanguage('JavaScript');

// ссылочная форма
let language = 'JavaScript';
setLanguage(language);
```

В TypeScript такой рефакторинг также работает:

```
function setLanguage(language: string) { /* ... */ }

setLanguage('JavaScript'); // ok

let language = 'JavaScript';
setLanguage(language); // ok
```

Теперь предположим, что вы серьезно отнеслись к совету из правила 33 и заменили строковый тип на более точный тип объединения строковых литералов:

```
type Language = 'JavaScript' | 'TypeScript' | 'Python';
function setLanguage(language: Language) { /* ... */ }

setLanguage('JavaScript'); // ok

let language = 'JavaScript';
setLanguage(language);
// ~~~~~ Аргумент типа 'string' не может быть назначен
// параметру типа 'Language'
```

Что же пошло не так? В строковой форме TypeScript, исходя из декларации функции, знает, что параметр должен принадлежать типу `Language`, а строковый литерал `'JavaScript'` может быть назначен для этого типа. Здесь все в порядке. Но когда вы выносите переменную, то TypeScript вынужден делать вывод ее типа сразу при назначении. В этом случае он выводит `string`, который не может быть назначен для `Language`.

(Некоторые языки выводят переменную на основе ее применения. Однако это может сбить с толку. Создатель TypeScript Андерс Хейлсберг описывает это как «тяжелое последствие». В большинстве случаев TypeScript выводит тип переменной при ее первом появлении, за исключением примеров из правила 41.)

Существует два способа исправить ошибку. Первый — установить пределы возможных значений `language`, объявив тип:

```
let language: Language = 'JavaScript';
setLanguage(language); // ok
```

Появится возможность обнаружить опечатку в `language`. К примеру, слово `'Typescript'` должно быть напечатано с заглавной «S».

Еще одним решением будет сделать переменную константой:

```
const language = 'JavaScript';
setLanguage(language); // ok
```

Исходя из неизменности переменной, TypeScript может вывести тип `language` более точно, а именно как строковый литерал `'JavaScript'`. Если назначить его для `Language`, код пройдет проверку типов. Но переназначение `language` потребует объявления типа (правило 21).

Основной проблемой в нашем случае стало то, что мы вырвали значение из контекста. Ниже описаны другие случаи ошибок при утрате контекста.

Кортежные типы

Предположим, вы работаете с визуализацией карты и хотите сделать панораму:

```
// Параметр является парой (latitude, longitude).
function panTo(wher: [number, number]) { /* ... */ }

panTo([10, 20]); // ok

const loc = [10, 20];
panTo(loc);
//   ~~ Аргумент типа 'number[]' не может быть назначен
//       параметру типа '[number, number]'.
```

Снова значение вырвано из контекста. В первом случае `[10, 20]` могут быть назначены для кортежного типа `[number, number]`, во втором TypeScript выводит тип `loc` как `number[]`, то есть как массив чисел неизвестной длины, который не может быть назначен для кортежного типа из-за неподходящего числа элементов.

Как исправить эту ошибку без `any`? Можно объявить тип, чтобы TypeScript понял ваши намерения:

```
const loc: [number, number] = [10, 20];
panTo(loc); // ok
```

Еще один способ — сделать утверждение `const`. Так вы сообщите TypeScript постоянный контекст, а не введете простую константу:

```
const loc = [10, 20] as const;
panTo(loc);
//   ~~ Тип 'readonly [10, 20]' является 'readonly'
//       и не может быть назначен для изменяемого типа '[number, number]'.
```

Если вы наведете курсор на `loc` в редакторе, то увидите, что ее тип выведен как `readonly [10, 20]`, а не `number[]`. К сожалению, это окончательно. Поэтому лучшим решением будет добавить аннотацию `readonly` к функции `panTo`:

```
function panTo(wher: readonly [number, number]) { /* ... */ }
const loc = [10, 20] as const;
panTo(loc); // ok
```

Когда сигнатура типа оказывается вне вашего контроля, используйте аннотацию.

Утверждение `const` может четко разрешить сложности вывода типа при утрате контекста, но и здесь есть обратная сторона. Если вы допустите ошибку в определении (например, добавьте третий элемент в кортеж), тогда ошибка будет отображена в месте вызова, а не в месте определения, что усложнит ее предупреждение:

```
const loc = [10, 20, 30] as const; // на самом деле ошибка здесь.
panTo(loc);
//    ~~~ Аргумент типа '[10, 20, 30]' не может быть назначен
//           параметру типа '[number, number]'.
//           Типы свойства 'length' несовместимы.
//           Тип '3' не может быть назначен для типа '2'.
```

Объекты

Проблема отрыва значения от контекста также возникает, когда вы выносите константу из крупного объекта, содержащего строковые литералы или кортежи. Например:

```
type Language = 'JavaScript' | 'TypeScript' | 'Python';
interface GovernedLanguage {
  language: Language;
  organization: string;
}

function complain(language: GovernedLanguage) { /* ... */ }

complain({ language: 'TypeScript', organization: 'Microsoft' }); // ok

const ts = {
  language: 'TypeScript',
  organization: 'Microsoft',
};
complain(ts);
//    ~ Аргумент типа '{ language: string; organization: string; }'
//      не может быть назначен параметру типа 'GovernedLanguage'.
//      Типы свойства 'language' несовместимы.
//      Тип 'string' не может быть назначен для типа 'Language'.
```

В объекте `ts` тип `language` выведен как `string`. Как и прежде, решением будет добавить декларацию типа (`const ts: GovernedLanguage = ...`) либо использовать утверждение `const` (`as const`).

Обратные вызовы

Когда вы передаете обратный вызов другой функции, TypeScript использует контекст для вывода типов его параметров:

```
function callWithRandomNumbers(fn: (n1: number, n2: number) => void) {
    fn(Math.random(), Math.random());
}

callWithRandomNumbers((a, b) => {
    a; // Тип number
    b; // Тип number
    console.log(a + b);
});
```

Типы `a` и `b` выведены как `number` согласно декларации типа для `callWithRandom`. Если вы вынесете обратный вызов в виде константы, то утратите контекст и получите ошибки от `noImplicitAny`:

```
const fn = (a, b) => {
    // ~    Параметр 'a' неявно имеет тип 'any'.
    // ~    Параметр 'b' неявно имеет тип 'any'.
    console.log(a + b);
}
callWithRandomNumbers(fn);
```

Решением будет добавить аннотацию типа для параметров:

```
const fn = (a: number, b: number) => {
    console.log(a + b);
}
callWithRandomNumbers(fn);
```

либо применить декларацию типа для всего выражения функции, если оно будет доступно (правило 12).

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Оценивайте влияние контекста на вывод типов.
- ✓ Если вынесение переменной вызывает ошибки типа, рассмотрите вариант добавления декларации типа.
- ✓ Если переменная однозначно является константой, то используйте утверждение `const` (`as const`). Но помните, что это может вызвать ошибки, всплывающие в местах вызова, а не определения.

ПРАВИЛО 27. Используйте функциональные конструкции и библиотеки для содействия движению типов

В JavaScript никогда не было стандартных библиотек, как в Python, C или Java. В течение ряда лет сторонние библиотеки восполняют этот пробел. jQuery предоставляет хелперы не только для взаимодействия с DOM, но также для итерации и отображения объектов и массивов. Underscore и Lodash больше сфокусированы на предоставлении общих функциональных утилит. Современные библиотеки, такие как Ramda, продолжают привносить идеи из функционального программирования в мир JavaScript.

Некоторые особенности библиотек `map`, `flatMap`, `filter` и `reduce` попали в JavaScript выборочно. А когда в дело вступил TypeScript, их конструкции и декларации типов, гарантирующие передачу типов через сами конструкции, стали еще полезнее по сравнению с ручным прописыванием циклов, при котором вся ответственность за типы ложится на человека.

Например, рассмотрим парсинг CSV-данных. Вы могли бы сделать его в обычном JavaScript императивно:

```
const csvData = "...";
const rawRows = csvData.split('\n');
const headers = rawRows[0].split(',');
const rows = rawRows.slice(1).map(rowStr => {
  const row = {};
  rowStr.split(',').forEach((val, j) => {
    row[headers[j]] = val;
  });
  return row;
});
```

Разработчики с функциональным мышлением построили бы групповые объекты с помощью `reduce`:

```
const rows = rawRows.slice(1)
  .map(rowStr => rowStr.split(',').reduce(
    (row, val, i) => (row[headers[i]] = val, row),
    {}));
```

Такой вариант экономит три строки (почти 20 знаков без пробелов!), но при этом может оказаться менее понятным. Функция `zipObject` от Lodash,

формирующая объект посредством «уплотнения» массива ключей и значений, может сжать всю конструкцию еще больше:

```
import _ from 'lodash';
const rows = rawRows.slice(1)
    .map(rowStr => _.zipObject(headers, rowStr.split(','))));
```

Мне она кажется самой понятной. Но оправдывает ли себя добавление в проект зависимости от сторонней библиотеки? Особенно если вы не используете бандлер (bundler), а издержки, связанные с его применением, существенны.

Когда вы привносите в рабочий процесс TypeScript, то применение Lodash становится оправданным.

Обе версии CSV парсера VanillaJS выдают одну и ту же ошибку в TypeScript:

```
const rowsA = rawRows.slice(1).map(rowStr => {
    const row = {};
    rowStr.split(',').forEach((val, j) => {
        row[headers[j]] = val;
// ~~~~~ В типе '{}' не была обнаружена сигнатура индекса
// с параметром типа 'string'.
    });
    return row;
});
const rowsB = rawRows.slice(1)
    .map(rowStr => rowStr.split(',').reduce(
        (row, val, i) => (row[headers[i]] = val, row),
        // ~~~~~ В типе '{}' не была обнаружена
        // сигнатура индекса с параметром
        // типа 'string'.
        {}));
```

Во всех этих случаях возможно аннотирование типа для {}, а именно {[column: string]: string} либо Record<string, string>.

Вариант с Lodash, напротив, проходит проверку типов без модификации:

```
const rows = rawRows.slice(1)
    .map(rowStr => _.zipObject(headers, rowStr.split(',')));
// тип _.Dictionary<string>[]
```

Dictionary в Lodash является псевдонимом {[key: string]: string} или Record<string, string>. Важно здесь то, что тип rows оказывается в точности верным и не требуется аннотации типа.

Эти преимущества становятся более заметными, когда процесс преобразования данных усложняется. Например, у вас есть список составов всех команд NBA:

```
interface BasketballPlayer {
  name: string;
  team: string;
  salary: number;
}
declare const rosters: {[team: string]: BasketballPlayer[]};
```

Для построения плоского списка посредством цикла вы можете применить к массиву функцию `concat`. Следующий код исправно запускается, но не проходит проверку типов:

```
let allPlayers = [];
// ~~~~~ Переменная 'allPlayers' неявно имеет тип 'any[]'
// ~~~~~ в местах, где ее тип не может быть определен.
for (const players of Object.values(rosters)) {
  allPlayers = allPlayers.concat(players);
  // ~~~~~ Переменная 'allPlayers' неявно имеет
  // ~~~~~ тип 'any[]'.
}
```

Для исправления ошибки можно добавить аннотацию типа `allPlayers`:

```
let allPlayers: BasketballPlayer[] = [];
for (const players of Object.values(rosters)) {
  allPlayers = allPlayers.concat(players); // ok
}
```

Но еще лучшим решением будет использовать `Array.prototype.flat`:

```
const allPlayers = Object.values(rosters).flat();
// ok, тип BasketballPlayer[]
```

Метод `flat` выравнивает многомерный массив. Его сигнатура типа изменяется по принципу `T[][] => T[]`. Этот вариант оказывается самым лаконичным и не требует аннотации типов. В качестве бонуса вы можете использовать `concat` вместо `let` для предотвращения последующих изменений переменной `allPlayers`.

Допустим, вы хотите начать с `allPlayers` и создать список самых высокооплачиваемых игроков каждой команды, упорядоченный по сумме заработка.

Без `Lodash` потребуется аннотирование типа там, где не используются функциональные конструкции:

```
const teamToPlayers: {[team: string]: BasketballPlayer[]} = {};  
for (const player of allPlayers) {  
  const {team} = player;  
  teamToPlayers[team] = teamToPlayers[team] || [];  
  teamToPlayers[team].push(player);  
}  
  
for (const players of Object.values(teamToPlayers)) {  
  players.sort((a, b) => b.salary - a.salary);  
}  
  
const bestPaid = Object.values(teamToPlayers).map(players => players[0]);  
bestPaid.sort((playerA, playerB) => playerB.salary - playerA.salary);  
console.log(bestPaid);
```

Таков результат:

```
[  
  { team: 'GSW', salary: 37457154, name: 'Stephen Curry' },  
  { team: 'HOU', salary: 35654150, name: 'Chris Paul' },  
  { team: 'LAL', salary: 35654150, name: 'LeBron James' },  
  { team: 'OKC', salary: 35654150, name: 'Russell Westbrook' },  
  { team: 'DET', salary: 32088932, name: 'Blake Griffin' },  
  ...  
]
```

А вот эквивалент с использованием `Lodash`:

```
const bestPaid = _(allPlayers)  
  .groupBy(player => player.team)  
  .mapValues(players => _.maxBy(players, p => p.salary))  
  .values()  
  .sortBy(p => -p.salary)  
  .value() // тип BasketballPlayer[]
```

Этот код вдвое короче и яснее и требует лишь одно ненулевое утверждение (модуль проверки типов не знает, что массив `players`, переданный `.maxBy`, не пустой). Здесь применяется принцип «цепочки», использующийся в `Lodash` и `Underscore`. Он позволяет прописывать последовательность операций в более естественном порядке. Вместо написания:

```
_.a(_(b(_(c(v))))
```

вы пишете:

```
_(v).a().b().c().value()
```

_(v) «оборачивает» значение, а .value() его «разворачивает».

Если проинспектировать вызовы функции в цепочке по отдельности, чтобы увидеть тип обернутого значения, они всегда будут верными.

Многие характерные сокращения Lodash могут быть точно смоделированы в TypeScript. Например, вы можете использовать `_.map` вместо встроенного `Array.prototype.map`, чтобы передать имя (name) свойства, а не обратный вызов. Следующие вызовы дадут одинаковый результат:

```
const namesA = allPlayers.map(player => player.name) // тип string[]
const namesB = _.map(allPlayers, player => player.name) // тип string[]
const namesC = _.map(allPlayers, 'name'); // тип string[]
```

Эта способность TypeScript точно моделировать подобные конструкции является доказательством скрупулезности его системы типов. Однако она не справляется при комбинировании типов строковых литералов и типов индексов (правило 14). Если вы привыкли к C++ или Java, то подобный вывод типов может показаться трудным.

```
const salaries = _.map(allPlayers, 'salary'); // тип number[]
const teams = _.map(allPlayers, 'team'); // тип string[]
const mix = _.map(allPlayers, Math.random() < 0.5 ? 'name' : 'salary');
// тип (string | number)[]
```

При взаимодействии с конструкциями библиотек вроде Lodash типы без изменения и возвращения новых значений после каждого вызова способны производить новые типы (правило 20). TypeScript ориентирован на точное моделирование поведения библиотек, связанных с JavaScript, в реальных условиях. Воспользуйтесь этим!

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Используйте встроенные функциональные конструкции наряду с другими вспомогательными возможностями сторонних библиотек вместо ручного управления, чтобы улучшить движение типов, повысить читаемость кода и снизить потребность в явных аннотациях типов.

Проектирование типов

Покажите мне блок-схемы, скрыв таблицы, и я останусь в неведении.

Покажите таблицы, и мне, скорее всего, уже не понадобятся блок-схемы, так как они станут очевидными.

Фред Брукс. «Мифический человеко-месяц»

По-прежнему актуален смысл высказывания Фреда Брукса: сложно понять код, когда не видишь данные или типы данных, которые он обрабатывает. Прописывание типов делает их наглядными для читателя. А это, в свою очередь, делает код читабельным.

Предыдущие главы были посвящены основным моментам работы с типами: их использованию, выводу и прописыванию деклараций. Эта же глава описывает проектирование типов. Все примеры в этой главе были выражены через TypeScript, но многие из них имеют более широкий спектр возможного применения.

Если вы будете прописывать типы грамотно, то ваши блок-схемы станут очевидными.

ПРАВИЛО 28. Используйте типы, имеющие допустимые состояния

Продуманная структуризация типов не только облегчает написание кода, но и спасает от путаницы и багов.

Ключом к эффективному коду является проектирование типов, способных иметь только допустимые состояния. Мы рассмотрим несколько неудачных примеров и вариантов их исправления.

Предположим, вы пишете веб-приложение, которое загружает и затем отображает содержимое станицы. Можно прописать состояние так:

```
interface State {  
  pageText:string;  
  isLoading: boolean;  
  error?: string;  
}
```

Для отображения страницы нужно учесть все эти поля:

```
function renderPage(state: State) {  
  if (state.error) {  
    return `Error! Unable to load ${currentPage}: ${state.error}`;  
  } else if (state.isLoading) {  
    return `Loading ${currentPage}...`;  
  }  
  return `

# 

}
```

Все ли здесь хорошо? Что, если `isLoading` и `error` активируются вместе? Должно ли отображаться сообщение об ошибке или о загрузке? Сложно сказать.

А что, если вы пишете функцию `changePage`? Вот пример:

```
async function changePage(state: State, newPage: string) {  
  state.isLoading = true;  
  try {  
    const response = await fetch(getUrlForPage(newPage));  
    if (!response.ok) {  
      throw new Error(`Unable to load ${newPage}:  
                        ${response.statusText}`);  
    }  
    const text = await response.text();  
    state.isLoading = false;  
    state.pageText = text;  
  } catch (e) {  
    state.error = '' + e;  
  }  
}
```

Здесь присутствует множество проблем, вот некоторые из них:

- Не задано значение `false` для `state.isLoading` в случае возникновения ошибки.
- Не прояснено `state.error`, поэтому при провале предыдущего запроса вы продолжите видеть сообщение об ошибке вместо сообщения о загрузке.
- Если пользователь повторно изменит страницы в процессе загрузки, то неизвестно, чем это обернется. Либо он сперва увидит новую страницу, а затем ошибку, либо первую страницу вместо второй. Это будет зависеть от порядка возвращения запросов.

Состояние одновременно содержит недостаток информации (неясно, какой запрос провалился, а какой загружается) и ее избыток: тип `State` позволяет срабатывать и `isLoading`, и `error`. Корректное выполнение `render()` и `changePage()` невозможно.

Вот лучший вариант представления состояний:

```
interface RequestPending {
  state: 'pending';
}
interface RequestError {
  state: 'error';
  error: string;
}
interface RequestSuccess {
  state: 'ok';
  pageText: string;
}
type RequestState = RequestPending | RequestError | RequestSuccess;

interface State {
  currentPage: string;
  requests: {[page: string]: RequestState};
}
```

Здесь для явного моделирования различных состояний сетевых запросов применяется размеченное объединение. Такой вариант длиннее, но в нем реализовано важное преимущество, исключающее недопустимые запросы. Текущая страница и состояние каждого запроса смоделированы явно, благодаря чему применение функций `renderPage` и `changePage` дает нужный результат:

```

function renderPage(state: State) {
  const {currentPage} = state;
  const requestState = state.requests[currentPage];
  switch (requestState.state) {
    case 'pending':
      return `Loading ${currentPage}...`;
    case 'error':
      return `Error! Unable to load ${currentPage}: ${requestState.error}`;
    case 'ok':
      return `

# 


```

Двусмысленность первой реализации устранена. Даже если пользователь изменит страницу после отправки запроса, это не вызовет проблем. Предшествующий запрос по-прежнему будет завершаться, не оказывая влияния на интерфейс пользователя.

В качестве простого, но ужасающего примера рассмотрим судьбу самолета Airbus A330 авиакомпании «Эйр Франс», летевшего рейсом 447, который пропал над Атлантическим океаном 1 июня 2009 года. Самолет был оснащен электронной системой управления полетом, обрабатывающей команды от пилотов. В процессе расследования причин крушения было поднято множество вопросов относительно разумности использования автопилота в ситуациях, связанных с угрозой для жизни. Спустя два года, когда были восстановлены записи черного ящика, вскрылось немало факторов, приведших к трагедии. Однако решающими среди них оказались плохо сконструированные состояния.

Пульты управления пилота и второго пилота были реализованы в кабине отдельно. Контроль угла набегающего потока (угла атаки) осуществлялся

боковыми рычагами управления (слева и справа). Если пилот тянул рычаг «на себя», самолет начинал движение вверх, если направлял «от себя» — самолет устремлялся вниз. В самолете была реализована система с режимом dual input (двухканальный ввод команд), позволявшая двум рычагам двигаться независимо. Так данный режим можно смоделировать в TypeScript:

```
interface CockpitControls {  
    /** Угол левого рычага в градусах, 0 = нейтральный, + = вперед */  
    leftSideStick: number;  
    /** Угол правого рычага в градусах, 0 = нейтральный, + = вперед */  
    rightSideStick: number;  
}
```

Предположим, вам нужно для этой структуры данных написать функцию `getStickSetting`, которая будет просчитывать текущую позицию рычага. Как вы это сделаете? Допустим, контроль находится у пилота, сидящего слева:

```
function getStickSetting(controls: CockpitControls) {  
    return controls.leftSideStick;  
}
```

Но что если второй пилот возьмет контроль на себя? Возможно, следует применить условие, при котором какой-либо из рычагов отклонен от положения 0.

```
function getStickSetting(controls: CockpitControls) {  
    const {leftSideStick, rightSideStick} = controls;  
    if (leftSideStick === 0) {  
        return rightSideStick;  
    }  
    return leftSideStick;  
}
```

Но возникает сложность с реализацией: мы можем быть уверены в возвращении значения левого рычага только при нейтральном (0) значении правого. Тогда следует рассмотреть такой вариант:

```
function getStickSetting(controls: CockpitControls) {  
    const {leftSideStick, rightSideStick} = controls;  
    if (leftSideStick === 0) {  
        return rightSideStick;  
    } else if (rightSideStick === 0) {  
        return leftSideStick;  
    }  
    // ???  
}
```

Что же делать, если оба они не будут равны 0? Рассчитывая на то, что они будут примерно равны, вы можете просто их усреднить:

```
function getStickSetting(controls: CockpitControls) {
  const {leftSideStick, rightSideStick} = controls;
  if (leftSideStick === 0) {
    return rightSideStick;
  } else if (rightSideStick === 0) {
    return leftSideStick;
  }
  if (Math.abs(leftSideStick - rightSideStick) < 5) {
    return (leftSideStick + rightSideStick) / 2;
  }
  // ???
}
```

Но что, если они не будут таковыми? Можно ли выбросить ошибку? По сути, нет: элероны обязательно должны быть установлены под каким-то углом.

В случае с рейсом 447 второй пилот молча потянул рычаг со своей стороны, как только самолет вошел в шторм. Удалось набрать высоту, но в итоге упала скорость и самолет утратил возможность эту высоту удерживать.

В такой ситуации пилоты должны направлять рычаги «от себя», чтобы снова набрать скорость. Именно так и поступил пилот, но второй пилот продолжал тянуть свой рычаг на себя, вследствие чего функция программы управления аэробусом выглядела примерно так:

```
function getStickSetting(controls: CockpitControls) {
  return (controls.leftSideStick + controls.rightSideStick) / 2;
}
```

Даже несмотря на то что первый пилот полностью отжал рычаг «от себя», итоговая общая команда была усреднена к нулю. Он никак не мог понять, почему самолет не начинает «нырок». К тому моменту, когда второй пилот осознал свою ошибку, оставшейся высоты уже не хватало для выравнивания и произошло падение в океан, которое привело к гибели 228 человек.

Удачного способа реализации `getStickSetting` при таком режиме управления (ввода данных) нет. Функция была обречена на провал. В большинстве самолетов обе панели управления механически соединены. Если второй пилот потянет рычаг, то и рычаг пилота также подастся назад. Состояние такого управления легко выразить:

```
interface CockpitControls {  
    /** Угол атаки в градусах, 0 = нейтральный, + = вперед */  
    stickAngle: number;  
}
```

Теперь «блок-схемы очевидны», и вам больше не нужна функция `getStickSetting`.

В процессе конструирования типов обратите внимание на значения, предоставляемые типами, и применяйте только допустимые. Этот принцип очень важен, и некоторые из последующих правил этой главы будут посвящены рассмотрению отдельных случаев, связанных с ним.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Типы, способные представлять допустимые и недопустимые состояния, чаще всего ведут к путанице и появлению ошибок в коде.
- ✓ Предпочитайте использовать типы, способные представлять только допустимые значения. Даже если их формулировка сложнее, вы сэкономите время при работе с кодом.

ПРАВИЛО 29. Будьте либеральны в том, что берете, но консервативны в том, что даете

Эта идея известна как «Принцип надежности», или «Закон Постела», выведенный Джоном Постелом в контексте ТСП:

Схожие правила действуют для контрактов функций. Они допускают широкий спектр принимаемых входных значений, но должны быть более конкретны в том, что производят на выходе.

Например, API 3D-отображения может позволять позиционировать камеру и рассчитывать окно просмотра в ограничивающем параллелепипеде:

```
declare function setCamera(camera: CameraOptions): void;  
declare function viewportForBounds(bounds: LngLatBounds): CameraOptions;
```

Удобно то, что результат `viewportForBounds` может быть передан напрямую в `setCamera` для позиционирования камеры.

Давайте взглянем на определения этих типов:

```
interface CameraOptions {
  center?: LngLat;
  zoom?: number;
  bearing?: number;
  pitch?: number;
}
type LngLat =
  { lng: number; lat: number; } |
  { lon: number; lat: number; } |
  [number, number];
```

Все поля в `CameraOptions` являются опциональными и позволяют выбирать только центр или зум без изменения направления или угла наклона. Тип `LngLat` также дает свободу `setCamera` в получении элементов. Вы можете передать ей объект `{lng, lat}`, объект `{lon, lat}` или пару `[lng, lat]`, если уверены в последовательности. Такое расположение облегчает вызов функции.

Функция `viewportForBounds` получает другой, «либеральный» тип:

```
type LngLatBounds =
  {northeast: LngLat, southwest: LngLat} |
  [LngLat, LngLat] |
  [number, number, number, number];
```

Вы можете определить границы с помощью либо именованных углов, либо пары `lat/lngs`, либо четырехзначного кортежа, если уверены, что знаете правильный порядок. Так как `LngLat` уже содержит три формы, то остается не менее 19 возможных форм для `LngLatBounds`. Поистине либерально!

Теперь давайте напишем функцию, которая настраивает окно просмотра для добавления компонента (`Feature`) GeoJSON и сохраняет обновленное окно просмотра в URL (определение `CalculateBoundingBox` читайте в главе 31):

```
function focusOnFeature(f: Feature) {
  const bounds = calculateBoundingBox(f);
  const camera = viewportForBounds(bounds);
  setCamera(camera);
  const {center: {lat, lng}, zoom} = camera;
  // ~~~ Свойство 'lat' не существует в типе ...
  // ~~~ Свойство 'lng' не существует в типе ...
  zoom; // тип number | undefined
  window.location.search = `?v=@${lat},${lng}z${zoom}`;
}
```

Упс! Существует только свойство `zoom`, но его тип выведен как `number|undefined`, поэтому декларация типа `viewportForBounds` оказывается либеральной не только в том, что принимает, но и в том, что производит. Единственный типобезопасный способ использования результата `camera` — это выделение разных веток кода для каждого компонента типа объединения (правило 22).

Возвращаемый тип со множеством опциональных свойств и типов объединения усложняет использование `viewportForBounds`. Его широкий параметрический тип удобен, но широкий возвращаемый тип — нет. API окажется куда удобнее, если будет строже к тому, что производит.

Один из способов сделать это — определить образцовый формат для координат. Следуя правилу JavaScript о различии массива и его подобия (правило 16), вы найдете отличия между `LngLat` и `LngLatLike`, а также между полностью определенным типом `Camera` и его частичной версией, принимаемой `setCamera`:

```
interface LngLat { lng: number; lat: number; };
type LngLatLike = LngLat | { lon: number; lat: number; } | [number,
                                                                number];

interface Camera {
  center: LngLat;
  zoom: number;
  bearing: number;
  pitch: number;
}
interface CameraOptions extends Omit<Partial<Camera>, 'center'> {
  center?: LngLatLike;
}
type LngLatBounds =
  {northeast: LngLatLike, southwest: LngLatLike} |
  [LngLatLike, LngLatLike] |
  [number, number, number, number];

declare function setCamera(camera: CameraOptions): void;
declare function viewportForBounds(bounds: LngLatBounds): Camera;
```

Свободный тип `CameraOptions` приспособливает более строгий тип `Camera` (правило 14). Применение `Partial<Camera>` в качестве параметрического типа в `setCamera` не сработает, поскольку вам нужно сделать объекты `LngLatLike` доступными для свойства `center`. В то же время вы не можете написать `"CameraOptions extends Partial<Camera>"`, так как `LngLatLike` яв-

ляется надмножеством, а не подмножеством `LngLat` (правило 7). Проще можно прописать тип явно, но с повторами:

```
interface CameraOptions {  
  center?: LngLatLike;  
  zoom?: number;  
  bearing?: number;  
  pitch?: number;  
}
```

В обоих случаях с этими новыми декларациями функция `focusOnFeature` проходит проверку типов:

```
function focusOnFeature(f: Feature) {  
  const bounds = calculateBoundingBox(f);  
  const camera = viewportForBounds(bounds);  
  setCamera(camera);  
  const {center: {lat, lng}, zoom} = camera; // ok  
  zoom; // тип number  
  window.location.search = `?v=@${lat},${lng}z${zoom}`;  
}
```

На этот раз типом `zoom` является `number`, а не `number|undefined`. Функцию `viewportForBounds` стало гораздо проще использовать. Наличие других функций, проводящих границы, потребует введения образцовой формы и обозначения различий между `LngLatBounds` и `LngLatBoundsLike`.

Хороша ли конструкция, допускающая 19 вариантов ограничивающего параллелепипеда? Вероятно, нет. Поэтому если вы пишете декларации типов для библиотеки, которая такое позволяет, смоделируйте ее поведение. Главное, не создавайте 19 возвращаемых типов!

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Вводные типы, как правило, оказываются шире, чем типы вывода. Опциональные свойства и типы объединения больше свойственны параметрическим типам, чем возвращаемым.
- ✓ Чтобы заново использовать типы между параметрами и возвращаемыми типами, вводите образцовую форму (для возвращаемых типов) и свободную форму (для параметров).

ПРАВИЛО 30. Не повторяйте информацию типа в документации

Что не так с этим кодом?

```
/**
 * Возвращает строку с основным цветом.
 * Получает либо 1 аргумент, либо не получает совсем. Не получив аргумент,
 * возвращает стандартный основной цвет. Получив один аргумент, возвращает
 * основной цвет для конкретной страницы.
 */

function getForegroundColor(page?: string) {
  return page === 'login' ? {r: 127, g: 127, b: 127} : {r: 0, g: 0, b: 0};
}
```

Код и комментарий не согласованы. Без контекста сложно понять, что правильно, а что нет. Как говорил мой профессор: «Если код не согласуется с комментарием, значит, они оба неверны!»

Давайте предположим, что верен код. Тогда в комментарии есть несколько ошибок:

- Он сообщает, что функция возвращает цвет в виде `string`, но она возвращает объект `{r, g, b}`.
- В нем сказано, что функция не получает аргументы или получает один аргумент, что и так ясно из сигнатуры типа.
- Он неоправданно многословен, так как оказывается длиннее декларации функции и реализации.

Система аннотации типов TypeScript компактна, наглядна и читабельна. Ее разрабатывали эксперты, имеющие многолетний опыт работы с языком. Ни к чему повторять в комментарии информацию о типах входных и выходных данных функции.

Аннотации типов проходят проверку компилятором TypeScript и не утратят синхронности с реализацией. Вероятно, `getForegroundColor` ранее возвращала `string`, но позже была изменена для возвращения объекта. Человек, внесший эти изменения, похоже, просто забыл обновить длинный комментарий.

Синхронность должна быть предписана. В случае с аннотациями типов эту функцию на себя берет модуль проверки. Если вы добавите информацию

о типах в их аннотации, а не в документацию, то гарантируете их стабильность в процессе развития кода.

Лучший комментарий выглядел бы так:

```
/** Получает основной цвет приложения или конкретной страницы. */  
function getForegroundColor(page?: string): Color {  
    // ...  
}
```

Чтобы описать конкретный параметр, используйте JSDoc-аннотацию `@param` (правило 48).

Комментарии, сообщающие об отсутствии изменения, также излишни:

```
/** Не модифицирует nums */  
function sort(nums: number[]) { /* ... */ }
```

Вместо них объявите `readonly` (правило 17), и TypeScript сам проследит за соблюдением контракта:

```
function sort(nums: readonly number[]) { /* ... */ }
```

Что оказывается верным для комментариев, сомнительно для имен переменных. Избегайте примешивать к ним типы: вместо именованной переменной `ageNum` назовите ее `age` и убедитесь, что она действительно является `number`.

Исключением здесь станут числа с единицами измерения. Если они не очевидны, то их можно включить в имя свойства или переменной. Например, имя `timeMS` окажется гораздо понятнее, чем просто `time`, и `temperatureC` будет более ясным, чем `temperature`. Правило 37 описывает «стандарты», определяющие наиболее типобезопасный подход к моделированию единиц измерения.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Избегайте повторения информации типов в комментариях и именах переменных. В лучшем случае это приведет к дублированию аннотаций типов, в худшем — к конфликту информации.
- ✓ Рассмотрите добавление единиц измерения в имена переменных, если они не очевидны из самих типов (`timeMS`, `temperatureC`).

ПРАВИЛО 31. Смещайте нулевые значения на периферию типов

Когда вы впервые включаете опцию `strictNullChecks`, может показаться, что она потребует великого множества инструкций `if` для проверки значения `null` и `undefined` по всему коду. Ведь связи между нулевыми и ненулевыми значениями неявные: когда переменная `A` является ненулевой, вы знаете, что переменная `B` также не равна нулю, и наоборот.

Со значениями гораздо легче работать, если они однозначно либо нулевые, либо ненулевые. Это можно смоделировать, сдвинув нулевые значения на периферию конструкции.

Предположим, вы хотите вычислить минимум (`min`) и максимум (`max`) в списке чисел. Назовем это интервалом (`extent`). Вот вариант:

```
function extent(nums: number[]) {
  let min, max;
  for (const num of nums) {
    if (!min) {
      min = num;
      max = num;
    } else {
      min = Math.min(min, num);
      max = Math.max(max, num);
    }
  }
  return [min, max];
}
```

Код проходит проверку типов (без включенной опции `strictNullChecks`) и имеет корректный возвращаемый тип `number[]`. Однако в нем есть баг и конструктивный дефект:

- Если значение `min` или `max` будет равно нулю, то оно может быть переназначено. Например, `extent([0, 1, 2])` вернет `[1, 2]` вместо `[0, 2]`.
- Если массив `nums` окажется пуст, то функция вернет `[undefined, undefined]`. Подобные объекты, имеющие несколько значений `undefined`, представляют сложности в работе для клиентов. Исходный код гласит, что `min` и `max` либо могут оба быть `undefined`, либо оба им не быть, но эта информация не представлена в системе типов.

Включение `strictNullChecks` делает обе эти проблемы более наглядными:

```
function extent(nums: number[]) {
  let min, max;
  for (const num of nums) {
    if (!min) {
      min = num;
      max = num;
    } else {
      min = Math.min(min, num);
      max = Math.max(max, num);
      // ~~~ Аргумент типа 'number | undefined' не может быть
      //      назначен параметру типа 'number'.
    }
  }
  return [min, max];
}
```

Возвращаемый тип `extent` теперь выведен как `(number | undefined)[]`, что делает ошибку конструирования более очевидной. Вероятнее всего, она будет выражаться в виде ошибки типа везде, где будет происходить вызов `extent`:

```
const [min, max] = extent([0, 1, 2]);
const span = max - min;
// ~~~ ~~~ Объект, вероятно, 'undefined'.
```

Ошибка в реализации `extent` появляется из-за того, что вы исключили возможное значение `undefined` для `min`, но не для `max`. Они инициализируются вместе, но информация об этом отсутствует в системе типов. Вы бы могли устранить эту ошибку, добавив проверку и для `max`, но в таком случае баг удвоится.

Лучшим решением будет поместить `min` и `max` в один объект и определить этот объект либо однозначно нулевым, либо однозначно ненулевым:

```
function extent(nums: number[]) {
  let result: [number, number] | null = null;
  for (const num of nums) {
    if (!result) {
      result = [num, num];
    } else {
      result = [Math.min(num, result[0]), Math.max(num, result[1])];
    }
  }
  return result;
}
```

С возвращаемым типом `[number, number] | null` клиентам будет гораздо легче работать. `min` и `max` могут быть получены либо путем ненулевого утверждения:

```
const [min, max] = extent([0, 1, 2])!;  
const span = max - min; // ok
```

либо с помощью единичной проверки:

```
const range = extent([0, 1, 2]);  
if (range) {  
  const [min, max] = range;  
  const span = max - min; // ok  
}
```

Используя один объект для вычисления интервала, мы улучшили структуру, помогли TypeScript понять связи между нулевыми значениями и исправили баг: проверка `if (!result)` происходит без проблем.

Смешение нулевых и ненулевых значений может также вести к сложностям в классах. Предположим, у вас есть класс, представляющий пользователя форума и его посты.

```
class UserPosts {  
  user: UserInfo | null;  
  posts: Post[] | null;  
  
  constructor() {  
    this.user = null;  
    this.posts = null;  
  }  
  
  async init(userId: string) {  
    return Promise.all([  
      async () => this.user = await fetchUser(userId),  
      async () => this.posts = await fetchPostsForUser(userId)  
    ]);  
  }  
  
  getName() {  
    // ...?  
  }  
}
```

Пока происходит загрузка двух сетевых запросов, свойства `user` и `posts` будут `null`. В любой момент времени они либо могут оба быть `null`, либо одно из

них будет `null`, либо ни одно. Получается четыре варианта, которые в своем множестве проникнут в каждый метод класса. Такая конструкция, скорее всего, приведет к запутанности, а также увеличению проверок `null` и багов.

Грамотная конструкция будет ожидать доступности всех данных, используемых классом.

```
class UserPosts {
  user: UserInfo;
  posts: Post[];

  constructor(user: UserInfo, posts: Post[]) {
    this.user = user;
    this.posts = posts;
  }

  static async init(userId: string): Promise<UserPosts> {
    const [user, posts] = await Promise.all([
      fetchUser(userId),
      fetchPostsForUser(userId)
    ]);
    return new UserPosts(user, posts);
  }

  getUserName() {
    return this.user.name;
  }
}
```

Теперь класс `UserPosts` полностью ненулевой, и в нем легко писать корректные методы. Конечно, если вам необходимо выполнять операции в процессе загрузки данных, то придется иметь дело со множеством нулевых и ненулевых состояний.

(Не заменяйте свойства, допускающие нулевые значения, промисами, чтобы методы не стали асинхронными. Промисы проясняют загружающий данные код, но оказывают противоположный эффект на класс, использующий эти данные.)

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Избегайте конструкций, где одно нулевое /ненулевое значение неявно связано с другим нулевым /ненулевым значением.

- ✓ Смещайте значения `null` на периферию API, создавая более крупные однозначно нулевые либо ненулевые объекты. Так вы сделаете код более понятным и для читателей, и для модуля проверки типов.
- ✓ Рассмотрите создание полностью ненулевого класса и его конструирование в момент доступности всех значений.
- ✓ Опция `strictNullChecks` может не только указывать на проблемы в коде, но и выявлять поведение функций, связанное с нулевыми значениями.

ПРАВИЛО 32. Предпочитайте объединения интерфейсов интерфейсам объединений

Если вам нужен интерфейс, свойствами которого являются типы объединения, рассмотрите вариант создания типа объединения двух более точных интерфейсов.

Предположим, в программе отрисовки вектора вы хотите определить интерфейс слоев с особыми геометрическими типами:

```
interface Layer {  
    layout: FillLayout | LineLayout | PointLayout;  
    paint: FillPaint | LinePaint | PointPaint;  
}
```

Поле `layout` контролирует, как и где прорисовываются формы, а поле `paint` определяет стили (например, толщину и цвет линий).

Есть ли смысл создавать слой, где `layout` является `LineLayout`, но свойство `paint` является `FillPaint`? Вероятно, нет. Такое допущение повышает вероятность возникновения ошибок при использовании библиотеки, а также усложняет работу с интерфейсом.

Лучшим способом будет разделить интерфейсы для каждого типа слоя:

```
interface FillLayer {  
    layout: FillLayout;  
    paint: FillPaint;  
}  
interface LineLayer {  
    layout: LineLayout;  
    paint: LinePaint;  
}
```

```
interface PointLayer {
  layout: PointLayout;
  paint: PointPaint;
}
type Layer = FillLayer | LineLayer | PointLayer;
```

Определив `Layer` таким образом, вы исключите вероятность смешения свойств `layout` и `paint` и последуете совету из правила 28, где говорится о типах, представляющих только допустимые значения.

Сравните этот паттерн с размеченным объединением. Одно из свойств будет объединением типов строковых литералов:

```
interface Layer {
  type: 'fill' | 'line' | 'point';
  layout: FillLayout | LineLayout | PointLayout;
  paint: FillPaint | LinePaint | PointPaint;
}
```

Будут ли иметь смысл создание `type: 'fill'` и последующее добавление `LineLayout` и `PointPaint`? Конечно нет. Преобразуйте `Layer` в объединение интерфейсов:

```
interface FillLayer {
  type: 'fill';
  layout: FillLayout;
  paint: FillPaint;
}
interface LineLayer {
  type: 'line';
  layout: LineLayout;
  paint: LinePaint;
}
interface PointLayer {
  type: 'point';
  layout: PointLayout;
  paint: PointPaint;
}
type Layer = FillLayer | LineLayer | PointLayer;
```

Свойство `type` является тегом и может быть использовано для определения, с каким типом `Layer` вы работаете при выполнении. Помимо этого, TypeScript способен сузить тип `Layer`, основываясь на теге:

```
function drawLayer(layer: Layer) {
  if (layer.type === 'fill') {
    const {paint} = layer; // Тип FillPaint
```

```

    const {layout} = layer; // Тип FillLayout
  } else if (layer.type === 'line') {
    const {paint} = layer; // Тип LinePaint
    const {layout} = layer; // Тип LineLayout
  } else {
    const {paint} = layer; // Тип PointPaint
    const {layout} = layer; // Тип PointLayout
  }
}

```

Правильно смоделировав связи между свойствами в этом типе, вы помогаете TypeScript проверить корректность кода. Такой же код, с применением изначального определения `Layer`, оказался бы загроможден утверждениями типов.

Размеченные объединения часто встречаются в TypeScript, так как они «дружат» с модулем проверки типов. Распознавайте этот паттерн и применяйте его где только можете.

Если вам удобно объединять опциональные поля с `undefined`, рассмотрите следующий тип:

```

interface Person {
  name: string;
  // Они оба будут либо представлены, либо нет.
  placeOfBirth?: string;
  dateOfBirth?: Date;
}

```

Комментарий с информацией типа — явный признак того, что здесь может скрываться проблема (правило 30). Между полями `placeOfBirth` и `dateOfBirth` имеется связь, о которой вы не сообщили TypeScript.

Лучшим решением будет переместить оба этих свойства в один объект. Это напоминает перемещение значений `null` на периферию (правило 31):

```

interface Person {
  name: string;
  birth?: {
    place: string;
    date: Date;
  }
}

```

Теперь TypeScript жалуется, что есть значение `place`, но отсутствует значение `date of birth`:

```
const alanT: Person = {
  name: 'Alan Turing',
  birth: {
    // ~~~~~ Свойство 'date' упущено в типе
    //       '{ place: string; }' но необходимо в типе
    //       '{ place: string; date: Date; }'.
    place: 'London'
  }
}
```

Помимо этого, функция, получающая объект `Person`, должна провести всего одну проверку:

```
function eulogize(p: Person) {
  console.log(p.name);
  const {birth} = p;
  if (birth) {
    console.log(`was born on ${birth.date} in ${birth.place}.`);
  }
}
```

Если структура типа находится вне вашего контроля (например, поступает от API), то вы все еще можете смоделировать связи между этими полями, используя объединение интерфейсов:

```
interface Name {
  name: string;
}

interface PersonWithBirth extends Name {
  placeOfBirth: string;
  dateOfBirth: Date;
}

type Person = Name | PersonWithBirth;
```

Вы получите ряд преимуществ, как в случае со вложенным объектом:

```
function eulogize(p: Person) {
  if ('placeOfBirth' in p) {
    p // Тип PersonWithBirth
    const {dateOfBirth} = p // ok, тип Date
  }
}
```

В обоих вариантах определение типа делает связь между свойствами более очевидной.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Использование интерфейсов, свойствами которых являются типы объединения, скрывает связи между этими свойствами и приводит к ошибкам.
- ✓ Объединения интерфейсов оказываются более точными, и TypeScript лучше их понимает.
- ✓ Рассмотрите вариант добавления тега для облегчения анализа потока управления. Благодаря хорошей поддержке размеченные объединения так популярны.

ПРАВИЛО 33. Используйте более точные альтернативы типов `string`

Область типа `string` может вместить весь текст «Моби Дика» (он содержит 1,2 миллиона знаков). Прежде чем присваивать его переменной, подумайте, не будет ли лучше использовать более узкий тип.

Предположим, вы создаете музыкальную коллекцию и хотите определить тип альбома. Вот вариант реализации:

```
interface Album {  
  artist: string;  
  title: string;  
  releaseDate: string; // ГГГГ-ММ-ДД  
  recordingType: string; // например "live" или "studio"  
}
```

Превалирование типов `string` наряду с информацией типов в комментариях (правило 30) является признаком проблем в `interface`. Вот что может пойти не так:

```
const kindOfBlue: Album = {  
  artist: 'Miles Davis',  
  title: 'Kind of Blue',  
  releaseDate: 'August 17th, 1959', // унс!  
  recordingType: 'Studio', // унс!  
}; // ok
```

Поле `releaseDate` имеет формат, не соответствующий комментарию, а в `"Studio"` использована заглавная буква вместо строчной. Однако оба

значения являются строками, и объект может быть назначен для `Album`, а модуль проверки не обнаруживает ошибок.

Будучи обширными, эти типы `string` могут скрывать ошибки в приемлемых объектах `Album`. Например:

```
function recordRelease(title: string, date: string) { /* ... */ }
recordRelease(kindOfBlue.releaseDate, kindOfBlue.title); // ok, должна
                                                         // быть ошибка
```

В вызове `recordRelease` параметры поменялись местами, но модуль проверки снова не жалуется, так как они являются строками. Из-за превалирования типов `string` подобный код часто называют строчно-типизированным.

Можно ли сузить типы для предотвращения подобных проблем? Тип `string` вполне подходит для имени артиста или названия альбома, но для поля `releaseDate` лучше использовать объект `Date`, чтобы избежать сложностей с проектированием. Для поля `recordingType` можно определить тип объединения, имеющий только два значения (также можно использовать `enum`, но я бы не рекомендовал это делать в данном случае (правило 53)).

```
type RecordingType = 'studio' | 'live';
```

```
interface Album {
  artist: string;
  title: string;
  releaseDate: Date;
  recordingType: RecordingType;
}
```

Внеся эти изменения, вы позволили TypeScript произвести более тщательную проверку:

```
const kindOfBlue: Album = {
  artist: 'Miles Davis',
  title: 'Kind of Blue',
  releaseDate: new Date('1959-08-17'),
  recordingType: 'Studio'
// ~~~~~ Тип '"Studio"' не может быть назначен
// для типа 'RecordingType'.
};
```

Помимо строгой проверки, в таком подходе присутствуют и другие преимущества. Во-первых, явное определение типа гарантирует, что его значение не затеряется в процессе передачи. Чтобы находить альбомы или конкретные записи по типам, можете определить подобную функцию:

```
function getAlbumsOfType(recordingType: string): Album[] {
  // ...
}
```

Откуда человек, вызывающий эту функцию, знает, какой `recordingType` ожидается? Ведь это просто `string`. Комментарий, поясняющий, что он может быть "studio" либо "live", спрятан в определении `Album`, куда пользователь может и не заглянуть.

Во-вторых, явное определение типа позволяет его задокументировать (правило 48):

```
/** В каких условиях была проведена эта запись */
type RecordingType = 'live' | 'studio';
```

Когда вы предпишете `getAlbumsOfType` получать `RecordingType`, пользователь сможет выйти на указанную документацию, сделав всего один клик (рис. 4.1).

<pre>type RecordingType = "live" "studio"</pre>
<pre>What type of environment was this recording made in?</pre>

```
function getAlbumsOfType(recordingType: RecordingType): Album[] {
```

Рис. 4.1. Использование именованного типа вместо строки позволяет прикрепить документацию к типу, отображаемому в редакторе

Еще одна распространенная ошибка — применение `string` в параметрах функции. К примеру, вы пишете функцию, которая извлекает все значения для одного поля массива. В библиотеке `Underscore` это называется "pluck" (выщипывание):

```
function pluck(records, key) {
  return record.map(record => record[key]);
}
```

Как бы вы ее типизировали? Вот первая попытка:

```
function pluck(record: any[], key: string): any[] {
  return record.map(r => r[key]);
}
```

Этот код пройдет проверку, но далек от совершенства. Типы `any` весьма проблематичны, особенно в возвращаемых значениях (правило 38). Пер-

вым шагом по улучшению сигнатуры типа будет введение обобщенного параметра типа:

```
function pluck<T>(record: T[], key: string): any[] {
    return record.map(r => r[key]);
    // ~~~~~ Элемент неявно имеет тип 'any', так как
    //      тип '{}' не имеет сигнатуры индекса.
}
```

Теперь TypeScript жалуется на то, что тип `string` оказывается слишком обширным для `key`. В этом он прав. При передаче массива альбомов для `key` окажется только четыре допустимых значения ("`artist`", "`title`", "`releaseDate`", и "`recordingDate`") в отличие от огромного числа вариантов `string`. Вот чем в точности является тип `keyof Album`:

```
type K = keyof Album;
// тип "artist" | "title" | "releaseDate" | "recordingType"
```

Поэтому исправить это можно, заменив `string` на `keyof T`:

```
function pluck<T>(record: T[], key: keyof T) {
    return record.map(r => r[key]);
}
```

Теперь код пройдет проверку типов, и TypeScript выведет возвращаемый тип. Если навести курсор на `pluck` в редакторе, то выведенный тип будет указан как:

```
function pluck<T>(record: T[], key: keyof T): T[keyof T][]
```

`T[keyof T]` является типом любого возможного значения в `T`. Если вы передадите одну строку в качестве `key`, то он окажется слишком обширным. Например:

```
const releaseDates = pluck(albums, 'releaseDate'); // тип (string | Date)[]
```

Тип должен быть `Date[]`, а не `(string | Date)[]`. Хотя тип `keyof T` и гораздо уже, чем `string`, он все еще слишком обширен. Для его дальнейшего сужения нужно ввести второй обобщенный параметр, который будет подмножеством `keyof T` (вероятно, одно значение):

```
function pluck<T, K extends keyof T>(record: T[], key: K): T[K][] {
    return record.map(r => r[key]);
}
```

(Более подробно об `extends` в этом контексте читайте в правиле 14.)

Теперь сигнатура типа верна. Мы можем проверить это, вызвав `pluck` несколькими способами:

```
pluck(albums, 'releaseDate'); // тип Date[]
pluck(albums, 'artist'); // тип string[]
pluck(albums, 'recordingType'); // тип RecordingType[]
pluck(albums, 'recordingDate');
    // ~~~~~ Аргумент типа '"recordingDate"' не может
    // быть назначен параметру типа ...
```

Языковая служба даже предлагает произвести автоподстановку в ключах (`keys`) `Album` (рис. 4.2):

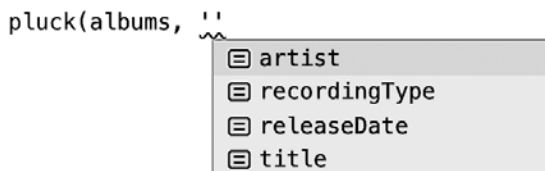


Рис. 4.2. Использование типа параметра `keyof Album` вместо строки приводит к улучшению автозаполнения в редакторе

Тип `string` имеет проблемы, присущие `any`. Он допускает неверные значения и скрывает связи между типами, мешая модулю проверки заметить некоторые баги. К счастью, способность TypeScript обнаруживать подмножества `string` добавляет безопасность типов в код JavaScript. Использование более точных типов поможет не только обнаружить ошибки, но и повысить читабельность самого кода.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Избегайте написания строчно-типизированного кода. Предпочитайте более конкретные типы там, где `string` не является однозначно подходящим.
- ✓ Отдавайте предпочтение использованию типов объединения строковых литералов вместо `string`, если это более точно опишет область переменной. Таким образом вы получите более строгую проверку типов и дополнительный опыт в разработке.
- ✓ Используйте `keyof T` вместо `string` для параметров функции, которые должны стать свойствами объекта.

ПРАВИЛО 34. Лучше сделать тип незавершенным, чем ошибочным

Прописывая декларации, вы столкнетесь с вопросом точности типов, необходимой для обнаружения багов и расширения возможностей TypeScript. Но будьте внимательны: чем выше точность деклараций типов, тем легче допустить ошибку, а ошибочный тип может навредить больше, чем недоработанный. Предположим, вы пишете декларации для GeoJSON, с которым встречались в правиле 31. Геометрия GeoJSON может иметь один из нескольких типов, каждый из которых содержит массивы координат различной формы:

```
interface Point {
  type: 'Point';
  coordinates: number[];
}
interface LineString {
  type: 'LineString';
  coordinates: number[][];
}
interface Polygon {
  type: 'Polygon';
  coordinates: number[][][];
}
type Geometry = Point | LineString | Polygon; // Также несколько других
```

Все отлично, но тип `number[]` недостаточно точен для координат. Кортежный тип будет более уместен для описания широты и долготы:

```
type GeoPosition = [number, number];
interface Point {
  type: 'Point';
  coordinates: GeoPosition;
}
// и т. д.
```

Так, преподнеся пользователям более точные типы, вы будете ждать похвалы. Но получите жалобы. Даже если вы ранее использовали только широту и долготу, то пространственная позиция в GeoJSON допускает наличие третьего элемента — высоты и потенциально других элементов. Чтобы применять настолько точные декларации типов, пользователь будет вынужден ввести утверждения типов либо заглушить модуль проверки типов утверждением `as any`.

В качестве другого примера рассмотрите написание деклараций типов для лисподобного языка, представленного в формате JSON:

```

12
"red"
["+", 1, 2] // 3
["/", 20, 2] // 10
["case", [ ">", 20, 10], "red", "blue"] // "red"
["rgb", 255, 0, 127] // "#FF007F"

```

Библиотека Marbox использует подобную систему для управления отображением возможностей карты на разных устройствах.

Существует целый спектр уточнений, которые можно использовать для его типизации:

- ☐ допустить все;
- ☐ допустить строки, числа и массивы;
- ☐ допустить строки, числа и массивы, начинающиеся с известных имен функций;
- ☐ убедиться, что каждая функция получает верное количество аргументов;
- ☐ убедиться, что каждая функция получает аргументы с верными типами.

Две первые опции достаточно просты:

```

type Expression1 = any;
type Expression2 = number | string | any[];

```

Помимо этого вы должны ввести тестовый набор верных и неверных выражений. Увеличение точности типов поможет предотвратить возможную регрессию (правило 52):

```

const tests: Expression2[] = [
  10,
  "red",
  true,
  // ~~~ Тип 'true' не может быть назначен для типа 'Expression2'.
  ["+", 10, 5],
  ["case", [ ">", 20, 10], "red", "blue", "green"], // слишком много
                                                    // значений
  ["**", 2, 31], // должна быть обнаружена ошибка: отсутствует функция "**"
  ["rgb", 255, 128, 64],
  ["rgb", 255, 0, 127, 0] // слишком много значений
];

```

Для дальнейшего повышения точности можно использовать объединение типов строковых литералов в качестве первого элемента кортежа:

```

type FnName = '+' | '-' | '*' | '/' | '>' | '<' | 'case' | 'rgb';
type CallExpression = [FnName, ...any[]];
type Expression3 = number | string | CallExpression;

const tests: Expression3[] = [
  10,
  "red",
  true,
  // ~~~ Тип 'true' не может быть назначен для типа 'Expression3'
  ["+", 10, 5],
  ["case", [ ">", 20, 10], "red", "blue", "green"],
  ["**", 2, 31],
  // ~~~~~ Тип '***' не может быть назначен для типа 'FnName'
  ["rgb", 255, 128, 64]
];

```

Была обнаружена еще одна ошибка при отсутствии регрессии. Неплохо! Но как убедиться, что каждая функция получает верное число аргументов? Это сложно, так как для достижения всех вызовов функций тип должен быть рекурсивным. Как пояснялось в правиле 13, для выполнения этой задачи потребуется ввести как минимум один `interface`. Но так как `interface` не может быть объединением, то придется прописывать выражения вызовов, также используя `interface`. Это несколько затруднительно, ведь массивы с фиксированной длиной гораздо легче выражаются кортежными типами. Тем не менее вы *можете* это сделать:

```

type Expression4 = number | string | CallExpression;
type CallExpression = MathCall | CaseCall | RGBCall;

interface MathCall {
  0: '+' | '-' | '/' | '*' | '>' | '<';
  1: Expression4;
  2: Expression4;
  length: 3;
}

interface CaseCall {
  0: 'case';
  1: Expression4;
  2: Expression4;
  3: Expression4;
  length: 4 | 6 | 8 | 10 | 12 | 14 | 16 // и т.д.
}

interface RGBCall {
  0: 'rgb';
  1: Expression4;
  2: Expression4;
  3: Expression4;
  length: 4;
}

```

```

}

const tests: Expression4[] = [
  10,
  "red",
  true,
  // ~~~ Тип 'true' не может быть назначен для типа 'Expression4'
  ["+", 10, 5],
  ["case", [ ">", 20, 10], "red", "blue", "green"],
  // ~~~~~
  // Тип ['"case", [ ">", ...], ...]' не может быть назначен для типа 'string'
  ["**", 2, 31],
  // ~~~~~ Тип ['"', number, number]' не может быть назначен
  // для типа 'string'
  ["rgb", 255, 128, 64],
  ["rgb", 255, 128, 64, 73]
  // ~~~~~ Тип ['"rgb", number, number, number, number]'
  // не может быть назначен для типа 'string'
];

```

Теперь все неверные выражения выдают ошибки, и вы даже можете выразить нечто вроде «массива четной длины» с помощью `interface`. Но ошибка в `**` усложнилась с момента предыдущих типизаций.

Это улучшение по сравнению с предыдущими, менее точными типами? То, что при неправильном использовании возникают ошибки, — это хорошо, но ошибки затруднят работу с этим типом. Языковые службы являются такой же частью опыта работы с TypeScript, как и проверка типов (см. правило 6), поэтому рекомендуется взглянуть на сообщения об ошибках, возникающих в результате объявлений типов, и попробовать автоподстановку там, где она должна работать. Если новые объявления типов более точны, но нарушают автоподстановку, то разработка в TypeScript станет менее приятной.

Сложность объявления типа также повысила вероятность появления ошибки. Например, `Expression4` требует, чтобы все математические операторы принимали два параметра, но спецификация выражения `Marbox` говорит, что `+` и `*` могут принимать больше. Кроме того, `-` может принимать один параметр, и в этом случае отрицает свой ввод. `Expression4` неправильно отмечает ошибки во всех этих:

```

const okExpressions: Expression4[] = [
  ['-', 12],
  // ~~~~~ Тип ['"-", number]' нельзя назначить типу 'string'
  ['+', 1, 2, 3],
  // ~~~~~ Тип ['"+", number, ...]' нельзя назначить типу 'string'
  ['*', 2, 3, 4],
  // ~~~~~ Тип ['"', number, ...]' нельзя назначить типу 'string'
];

```

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Избегайте «зловещей долины» типобезопасности: неверные типы часто вредят больше, чем недоуточненные.
- ✓ Если не удастся смоделировать тип более точно, остановитесь в уточнении! Лучше заполните возникший пробел с помощью `any` или `unknown`.
- ✓ В процессе повышения точности типов уделите внимание сообщениям об ошибках и автоподстановке. Помимо повышения корректности кода, вы приобретете интересный опыт в разработке.

ПРАВИЛО 35. Генерируйте типы на основе API и спецификаций, а не данных

В предыдущих правилах этой главы были рассмотрены преимущества грамотного конструирования типов. Но этот процесс требует времени. Было бы здорово автоматизировать его, не так ли?

Некоторые типы поступают от файловых форматов, API или спецификаций, то есть их нужно не прописывать, а генерировать. Если при этом вы будете ориентироваться на спецификации, а не на имеющиеся данные, то не упустите пограничные случаи, способные нарушить работу вашей программы.

В правиле 31 мы писали функцию для просчета ограничивающего параллелепипеда элемента GeoJSON. Выглядела она так:

```
function calculateBoundingBox(f: GeoJSONFeature): BoundingBox | null {
  let box: BoundingBox | null = null;

  const helper = (coords: any[]) => {
    // ...
  };

  const {geometry} = f;
  if (geometry) {
    helper(geometry.coordinates);
  }

  return box;
}
```

Тип `GeoJSONFeature` не был явно определен. Можно прописать его, основываясь на некоторых вариантах из правила 31. Однако лучше использовать для этого официальную спецификацию GeoJSON¹. К счастью, в репозитории `DefinitelyTyped` уже содержатся соответствующие декларации типов TypeScript. Добавить их можно обычным способом:

```
$ npm install --save-dev @types/geojson
+ @types/geojson@7946.0.7
```

После встраивания деклараций GeoJSON TypeScript сразу обозначит ошибку:

```
import {Feature} from 'geojson';

function calculateBoundingBox(f: Feature): BoundingBox | null {
  let box: BoundingBox | null = null;

  const helper = (coords: any[]) => {
    // ...
  };

  const {geometry} = f;
  if (geometry) {
    helper(geometry.coordinates);
    // ~~~~~
    // Свойство 'coordinates' не существует в типе
    // 'Geometry'.
    // Свойство 'coordinates' не существует в типе
    // 'GeometryCollection'.
  }

  return box;
}
```

Проблема в том, что код предполагает наличие в `geometry` свойства `coordinates`. Но `geometry` в GeoJSON может быть `GeometryCollection` — коллекцией других геометрий, которая, в отличие от аналогичных типов, не имеет свойства `coordinates`. Если вы вызовете `calculateBoundingBox` для `Feature`, чья геометрия является `GeometryCollection`, то произойдет ошибка, сообщающая о невозможности прочитать свойство `0` в `undefined`. Это баг, обнаруженный при помощи определения типов согласно спецификации.

¹ GeoJSON также известен как RFC 7946. Подробная спецификация доступна на <http://geojson.org>.

Есть два варианта его устранения. Первый — явно запретить `GeometryCollection`:

```
const {geometry} = f;
if (geometry) {
  if (geometry.type === 'GeometryCollection') {
    throw new Error('GeometryCollections are not supported.');
```

```
  }
  helper(geometry.coordinates); // ok
}
```

TypeScript способен уточнить тип `geometry`, основываясь на проверке, и обращение к `geometry.coordinates` окажется возможно. Если нет иных проблем, то это приведет к выдаче пользователю более ясного сообщения об ошибке.

Однако лучше поддержать все типы геометрий с помощью другой функции-хелпера:

```
const geometryHelper = (g: Geometry) => {
  if (geometry.type === 'GeometryCollection') {
    geometry.geometries.forEach(geometryHelper);
  } else {
    helper(geometry.coordinates); // ok
  }
}

const {geometry} = f;
if (geometry) {
  geometryHelper(geometry);
}
```

Если бы вы прописали декларации типов самостоятельно, то так и не поняли бы суть этого формата. Без учета `GeometryCollections` вы пришли бы к ложной уверенности в корректности кода. Использование типов на основе спецификации гарантирует, что код будет работать со всеми значениями, а не только с теми, которые вам знакомы.

По тому же принципу генерируйте типы для вызовов API, особенно если речь идет о типизированных API.

В качестве примера можно взять GraphQL, в API которого возможные запросы и интерфейсы определяются системой типов, похожей на систему TypeScript. Она позволяет писать запросы, которые обращаются к определенным полям в интерфейсах. Чтобы получить информацию о репозитории на GitHub с помощью GraphQL, можно написать:

```

query {
  repository(owner: "Microsoft", name: "TypeScript") {
    createdAt
    description
  }
}

```

Результатом будет:

```

{
  "data": {
    "repository": {
      "createdAt": "2014-06-17T15:28:39Z",
      "description":
        "TypeScript is a superset of JavaScript that compiles to
        JavaScript."
    }
  }
}

```

Хорошо в этом подходе то, что вы генерируете типы TypeScript для вашего конкретного запроса. Как и в примере с GeoJSON, это помогает убедиться, что вы безошибочно моделируете связи между типами и их возможными нулевыми значениями. Вот вариант запроса получения лицензии открытого программного обеспечения для репозитория на GitHub:

```

query getLicense($owner:String!, $name:String!){
  repository(owner:$owner, name:$name) {
    description
    licenseInfo {
      spdxId
      name
    }
  }
}

```

`$owner` и `$name` — это типизированные переменные GraphQL, синтаксис типов которого похож на синтаксис TypeScript. Вас может запутать, например, что в GraphQL тип `string` именуется как `String` (правило 10), а его типы допускают значение `null`. Оператор `!` после типа означает, что он гарантированно не будет нулевым.

Существует много инструментов, способных помочь вам перейти от запросов GraphQL к типам TypeScript. Один из них — Apollo. Вот пример его использования:

```
$ apollo client:codegen \
  --endpoint https://api.github.com/graphql \
  --includes license.graphql \
  --target typescript
Loading Apollo Project
Generating query files with 'typescript' target - wrote 2 files
```

Если вам нужна схема GraphQL, чтобы генерировать типы для запроса, Apollo получает ее из api.github.com/graphql. Результат будет таким:

```
export interface getLicense_repository_licenseInfo {
  __typename: "License";
  /** Короткий идентификатор определен <https://spdx.org/licenses> */
  spdxId: string | null;
  /** Полное имя лицензии определено <https://spdx.org/licenses> */
  name: string;
}

export interface getLicense_repository {
  __typename: "Repository";
  /** Описание репозитория. */
  description: string | null;
  /** Лицензия ассоциирована с репозиторием */
  licenseInfo: getLicense_repository_licenseInfo | null;
}

export interface getLicense {
  /** Поиск указанного репозитория по имени владельца и репозитория */
  repository: getLicense_repository | null;
}

export interface getLicenseVariables {
  owner: string;
  name: string;
}
```

Здесь важно отметить следующее:

- Интерфейсы генерируются и для параметров запроса (`getLicenseVariables`), и для отклика (`getLicense`).
- Информация о допустимости нулевых значений передается от схемы к интерфейсам отклика. Поля `repository`, `description`, `licenseInfo` и `spdxId` допускают нулевые значения, в то время как поле `name` лицензии и переменные запросов — нет.
- Документация передается в виде JSDoc, в связи с чем она появляется в вашем редакторе (правило 48). Комментарии поступают из самой схемы GraphQL.

Информация типов позволяет убедиться в верном использовании API. Если ваши запросы изменятся, то изменятся и типы. Если изменится схема, то следом за ней и типы. Здесь нет опасности, что типы не совпадут с реальными, так как источник изменений общий — схемы GraphQL.

А что, если нет доступных спецификаций или схемы? Тогда придется генерировать типы, основываясь на данных. Инструменты вроде `quicktype` способны в этом помочь, но имейте в виду, что типизация может не совпасть с действительностью и возникнут пограничные случаи, которые вы упустили.

Польза от генерации кода есть, даже если вы ее не замечаете. Декларации типов TypeScript для DOM генерируются из официальных интерфейсов браузера (правило 55). Это гарантирует корректное моделирование ими сложных систем, а также помогает TypeScript обнаруживать ошибки и разногласия в коде.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Рассмотрите применение генерации типов для API вызовов и форматов данных, чтобы гарантировать типобезопасность на протяжении всего кода.
- ✓ Отдайте предпочтение генерации кода на основе спецификаций, а не данных, так как вы можете упустить из виду маловероятные случаи его поведения.

ПРАВИЛО 36. Именуйте типы согласно области их применения

В computer science существует всего две серьезные проблемы: недействительность кэша и именование элементов.

Фил Карлтон

В этой книге уже много было сказано о форме типов и наборах значений в их областях, но меньше была затронута тема именования типов. Грамотно подобранные *имена* типов, свойств и переменных могут прояснить намерения разработчика, а также повысить уровень абстрактности кода и типов.

Неудачные же имена, наоборот, делают код менее понятным и формируют неверные ментальные модели.

Предположим, вы создаете базу данных животных. Далее приведен интерфейс, описывающий одного из них:

```
interface Animal {
    name: string;
    endangered: boolean;
    habitat: string;
}

const leopard: Animal = {
    name: 'Snow Leopard',
    endangered: false,
    habitat: 'tundra',
};
```

Здесь есть несколько проблем:

- Понятие `name` весьма обширно. Какое конкретно имя вы имеете в виду — научное или общепринятое?
- Логическое поле `endangered` также является неоднозначным, так как не уточнен охранный статус вида.
- Поле `habitat` (среда обитания) еще более неоднозначно. Не только потому, что оно относится к обширному типу `string` (правило 33), но также потому, что здесь неясно, что подразумевается под средой обитания.
- Имя переменной `leopard`, но свойство `name` имеет значение `"Snow leopard"`. Имеет ли значение это отличие?

Вот вариант с более точными типами и значениями:

```
interface Animal {
    commonName: string;
    genus: string;
    species: string;
    status: ConservationStatus;
    climates: KoppenClimate[];
}

type ConservationStatus = 'EX' | 'EW' | 'CR' | 'EN' | 'VU' | 'NT' | 'LC';
type KoppenClimate = |
    'Af' | 'Am' | 'As' | 'Aw' |
    'BSh' | 'BSk' | 'BWh' | 'Bwk' |
```

```

'Cfa' | 'Cfb' | 'Cfc' | 'Csa' | 'Csb' | 'Csc' | 'Cwa' | 'Cwb' | 'Cwc' |
'Dfa' | 'Dfb' | 'Dfc' | 'Dfd' |
'Dsa' | 'Dsb' | 'Dsc' | 'Dwa' | 'Dwb' | 'Dwc' | 'Dwd' |
'EF' | 'ET';
const snowLeopard: Animal = {
  commonName: 'Snow Leopard',
  genus: 'Panthera',
  species: 'Uncia',
  status: 'VU', // находящийся под угрозой исчезновения
  climates: ['ET', 'EF', 'Dfd'], // Альпийский или субальпийский
};

```

Здесь присутствует ряд улучшений:

- Свойство `name` было заменено более конкретными выражениями: `commonName`, `genus` и `species`.
- Свойство `endangered` сменилось на `conservationStatus`, которое использует стандартную систему классификации МСОП (Международного союза охраны природы).
- Свойство `habitat` стало `climates` и теперь использует иную таксономию — классификацию климатов Кеппена.

В первом варианте, чтобы узнать больше информации, потребовалось бы найти человека, который создавал базу данных, а он либо ушел из компании, либо не вспомнит детали этого проекта. Еще хуже, если вы запустите `git blame`, чтобы выяснить, кто написал такие низкопробные типы, и поймете, что это были вы.

Во втором варианте можно узнать больше о классификации климатов Кеппена или найти точное значение природоохранного статуса на онлайн-ресурсах.

Вместо изобретения определений позаимствуйте их из области вашей задачи. Это поможет вести понятный диалог с пользователями и повысить ясность применяемых типов.

Но будьте осторожны с малознакомой терминологией, чтобы еще больше не запутать себя и других.

Вот еще несколько правил, которые стоит учитывать при именовании типов, свойств и переменных.

- Придавайте различиям реальное значение. На письме и в речи мы украшаем текст синонимами, но в коде разные выражения должны называть разные реалии.

- Избегайте таких неоднозначных имен, как "data", "info", "thing", "item", "object" или популярного "entity" (сущность).
- Именувайте элементы согласно тому, чем они являются, а не тому, что содержат или как вычисляются. Например, имя `Directory` выражает значение лучше, чем `INodeList`, так как позволяет воспринимать директорию в общем, а не посредством описывающих ее выражений. Хорошие имена могут повысить уровень абстракции, понизив при этом число случайных противоречий.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Заимствуйте имена для элементов из области вашей задачи везде, где это возможно. Это повысит читаемость и уровень абстракции вашего кода.
- ✓ Избегайте употребления разных имен для одного понятия.

ПРАВИЛО 37. Рассмотрите использование маркировок для номинального типизирования

В правиле 4 рассматривалась утиная типизация, которая иногда приводит к неожиданным результатам:

```
interface Vector2D {  
  x: number;  
  y: number;  
}  
function calculateNorm(p: Vector2D) {  
  return Math.sqrt(p.x * p.x + p.y * p.y);  
}  
  
calculateNorm({x: 3, y: 4}); // ok, результат 5  
const vec3D = {x: 3, y: 4, z: 1};  
calculateNorm(vec3D); // ok! Результат также 5.
```

Как заставить функцию `calculateNorm` отклонять 3D-векторы? Это намерение противоречит модели структурной типизации TypeScript, но при этом математически оправданно.

Попробуйте использовать *номинальную типизацию* и самостоятельно определить значение как `Vector2D`. Или добавьте маркировку (`_brand`):

```

interface Vector2D {
  _brand: '2d';
  x: number;
  y: number;
}
function vec2D(x: number, y: number): Vector2D {
  return {x, y, _brand: '2d'};
}
function calculateNorm(p: Vector2D) {
  return Math.sqrt(p.x * p.x + p.y * p.y); // совпадает с предыдущим
                                           // вариантом
}

calculateNorm(vec2D(3, 4)); // ok, возвращает 5
const vec3D = {x: 3, y: 4, z: 1};
calculateNorm(vec3D);
// ~~~~ Свойство '_brand' отсутствует в типе...

```

Маркировка гарантирует, что вектор поступил из правильного источника. Менее безопасный способ — добавление `_brand: '2d'` к значению `vec3D`.

Вы устранили ограничения среды выполнения, поскольку сможете маркировать встроенные типы вроде `string` или `number` там, где нет возможности присоединить к ним дополнительные свойства.

Представьте функцию, которая работает с файловой системой и требует указания точного, а не относительного пути. Его легко проверить в среде выполнения (начинается ли путь с `"/"`), но не так легко в системе типов.

Вот подход с использованием маркировки:

```

type AbsolutePath = string & {_brand: 'abs'};
function listAbsolutePath(path: AbsolutePath) {
  // ...
}
function isAbsolutePath(path: string): path is AbsolutePath {
  return path.startsWith('/');
}

```

Вы не можете создать объект, являющийся `string` и имеющий свойство `_brand`. Этот прием работает только для системы типов.

Если у вас есть путь `string`, который может быть как точным, так и относительным, выясните его тип с помощью защиты типа:

```

function f(path: string) {
  if (isAbsolutePath(path)) {

```

```

    listAbsolutePath(path);
}
listAbsolutePath(path);
    // ~~~~ Аргумент типа 'string' не может быть назначен
    //      параметру типа 'AbsolutePath'
}

```

Такой подход поможет задокументировать, какие функции ожидают точные (**Absolute**), а какие — относительные пути и какой тип пути содержит каждая переменная. Он не даст стопроцентной гарантии, но утверждение `path as AbsolutePath` сработает для любого типа `string`. Без подобных утверждений единственным способом получить `AbsolutePath` будет его передача извне или проверка.

Этот вариант подходит для моделирования свойств, которые не могут быть выражены внутри системы типов, например бинарного поиска для обнаружения элемента в списке:

```

function binarySearch<T>(xs: T[], x: T): boolean {
    let low = 0, high = xs.length - 1;
    while (high >= low) {
        const mid = low + Math.floor((high - low) / 2);
        const v = xs[mid];
        if (v === x) return true;
        [low, high] = x > v ? [mid + 1, high] : [low, mid - 1];
    }
    return false;
}

```

Код работает, если список упорядочен. В противном случае результатом будут ложные отрицания. В системе типов TypeScript нельзя представить упорядоченный список, и его заменяет маркировка:

```

type SortedList<T> = T[] & {_brand: 'sorted'};

function isSorted<T>(xs: T[]): xs is SortedList<T> {
    for (let i = 1; i < xs.length; i++) {
        if (xs[i] > xs[i - 1]) {
            return false;
        }
    }
    return true;
}

function binarySearch<T>(xs: SortedList<T>, x: T): boolean {
    // ...
}

```

Для вызова этой версии `binarySearch` подтвердите упорядоченность списка с помощью `SortedList` или `isSorted`. Линейное сканирование не самое лучшее, но безопасное решение.

Этот подход можно успешно применять для модуля проверки типов. Например, для вызова метода или объекта потребуется либо получить ненулевой (`non-null`) объект извне, либо доказать, что он ненулевой, с помощью условия.

Вы также можете маркировать типы `number` для добавления единиц измерения:

```
type Meters = number & { _brand: 'meters' };
type Seconds = number & { _brand: 'seconds' };

const meters = (m: number) => m as Meters;
const seconds = (s: number) => s as Seconds;

const oneKm = meters(1000); // тип is Meters
const oneMin = seconds(60); // тип is Seconds
```

На практике это неудобно, так как после арифметических операций числа забывают свои маркировки:

```
const tenKm = oneKm * 10; // тип number
const v = oneKm / oneMin; // тип number
```

Даже если ваш код задействует множество чисел со смешанными единицами измерения, документирование ожидаемых типов числовых параметров будет по-прежнему хорошей идеей.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ TypeScript использует утиную типизацию, которая иногда ведет к неожиданным результатам. Если вам требуется номинальная типизация, то рассмотрите возможность добавления маркировок значений.
- ✓ В некоторых случаях доступно добавление маркировок внутри системы типов, а не в среде выполнения, которое позволяет моделировать свойства, недоступные в системе типов TypeScript.

Эффективное применение `any`

Традиционно системы типов могли быть либо статическими, либо динамическими. В TypeScript граница между ними размыта, так как его система типов опциональна и последовательна. Вы можете добавлять типы к частям собственного кода, но не чужого.

Это очень важно для переноса уже существующей кодовой базы JavaScript в TypeScript шаг за шагом (глава 8). Благодаря типам `any` вы можете игнорировать проверку типов для целых частей кода. Такая возможность одновременно дает преимущества и провоцирует массу проблем. Эта глава научит вас, как сократить отрицательное влияние типов `any`, не утратив при этом их преимуществ.

ПРАВИЛО 38. Используйте максимально узкий диапазон для типов `any`

Рассмотрите этот код:

```
function processBar(b: Bar) { /* ... */ }

function f() {
  const x = expressionReturningFoo();
  processBar(x);
  //      ~ Аргумент типа 'Foo' не может быть назначен
  //      параметру типа 'Bar'.
}
```

Если вам известно из контекста, что `x` может быть назначен не только для `Foo`, но и для `Bar`, то вы можете двумя способами принудить TypeScript принять такой код:

```
function f1() {
  const x: any = expressionReturningFoo(); // не делайте так
  processBar(x);
}

function f2() {
  const x = expressionReturningFoo();
  processBar(x as any); // лучше так
}
```

Почему второй вариант лучше? В нем диапазон типа `any` сокращен до единичного выражения в аргументе функции. Если код, следующий за вызовом `processBar`, обратится к `x`, то его тип по-прежнему будет `Foo`, что позволит обнаружить в нем возможные ошибки типов. В первом варианте диапазон действия типа `any` распространяется по всей функции.

Если вы *вернете* `x` из функции, произойдет следующее:

```
function f1() {
  const x: any = expressionReturningFoo();
  processBar(x);
  return x;
}

function g() {
  const foo = f1(); // Тип any
  foo.fooMethod(); // этот вызов не проверен!
}
```

Тип `any` в возвращаемом виде «заразен» и способен распространиться на всю базу кода. Изменения, внесенные нами в `f`, добавили тип `any` в `g`. Этого бы не произошло в суженном диапазоне, как в случае с `f2`.

(Рассмотрите необходимость явных аннотаций возвращаемых типов даже в случаях, когда они могут быть выведены. Это лишит типы `any` возможности выскользнуть (правило 19).)

Мы использовали `any`, чтобы отключить реакцию модуля проверки на ошибку, которая нам показалась неверной. Другой способ сделать то же самое — использовать `// @ts-ignore`:

```
function f1() {
  const x = expressionReturningFoo();
  // @ts-ignore
  processBar(x);
  return x;
}
```

Эта команда отключит обнаружение ошибки на следующей строке, позволяя оставить `x` неизменной. Постарайтесь не увлечься использованием `// @ts-ignore`, так как TypeScript имеет веские основания для указания ошибок. Кроме того, если ошибка на этой строке перерастет в большую проблему, то вы об этом не узнаете.

Бывают ситуации, при которых ошибка типа возникает только в одном свойстве крупного объекта:

```
const config: Config = {
  a: 1,
  b: 2,
  c: {
    key: value
  }
  // ~~~ Свойство ... отсутствует в типе 'Bar', но необходимо в типе 'Foo'.
};
```

Реакцию на подобные ошибки можно отключить, объявив `as any` для всего объекта `config`:

```
const config: Config = {
  a: 1,
  b: 2,
  c: {
    key: value
  }
} as any; // Не делайте так!
```

Но при этом отключится проверка типов для остальных его свойств (`a` и `b`). Сокращение же диапазона воздействия `any` устранил этот побочный эффект:

```
const config: Config = {
  a: 1,
  b: 2, // эти свойства по-прежнему проверяются
  c: {
    key: value as any
  }
};
```

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Применяйте `any` в максимально узком диапазоне, чтобы избежать нежелательного снижения безопасности типов на других участках кода.

- ✓ Никогда не используйте в функциях возвращаемый тип `any`. Это приведет к скрытой утрате безопасности типов в любом вызывающем функцию клиенте.
- ✓ Если вам необходимо отключить реакцию модуля проверки на конкретную ошибку, рассмотрите использование `// @ts-ignore` в качестве альтернативы `any`.

ПРАВИЛО 39. Используйте более точные варианты `any`

Тип `any` охватывает все значения, которые могут быть выражены в JavaScript. Их количество огромно! Он включает не только все `numbers` и `strings`, но и все массивы, объекты, регулярные выражения, функции, классы и элементы DOM, не говоря уже о `null` и `undefined`. Прежде чем использовать тип `any`, конкретизируйте свои намерения:

```
function getLengthBad(array: any) { // не делайте так!
    return array.length;
}

function getLength(array: any[]) {
    return array.length;
}
```

Последний вариант, использующий `any[]` вместо `any`, выигрывает по трем причинам:

- Обращение к `array.length` в теле функции проходит проверку типов.
- Возвращаемый тип функции выведен как `number`, а не `any`.
- Вызовы `getLength` будут проверены для подтверждения того, что параметр является массивом:

```
getLengthBad(/123/); // ошибки нет, возвращает undefined
getLength(/123/);
// ~~~~~ Аргумент типа 'RegExp' не может быть назначен
//        параметру типа 'any[]'.
```

Если вы ожидаете, что параметр будет массивом или массивами, но вам не важен его тип, то можно использовать `any[] []`. Если же вы ожидаете

некий объект, но не знаете, какие у него будут значения, то используйте `{[key: string]: any}`:

```
function hasTwelveLetterKey(o: {[key: string]: any}) {
  for (const key in o) {
    if (key.length === 12) {
      return true;
    }
  }
  return false;
}
```

В этой ситуации также можно применить тип `object`, который включает все непримитивные типы. Помните, что хотя вы и перечислите ключи, вы не сможете обратиться к их значениям:

```
function hasTwelveLetterKey(o: object) {
  for (const key in o) {
    if (key.length === 12) {
      console.log(key, o[key]);
      // ~~~~~ Элемент неявно имеет тип 'any',
      // так как тип '{}' не имеет сигнатуры индекса.
    }
    return true;
  }
  return false;
}
```

Если такой тип отвечает вашим требованиям, то вас заинтересует тип `unknown` (правило 42).

Избегайте применения `any`, когда ожидаете функциональный тип. В зависимости от того, насколько конкретный тип вы хотите получить, есть разные способы типизации:

```
type Fn0 = () => any; // любая функция, которая может быть вызвана
                      // без параметров
type Fn1 = (arg: any) => any; // ... с одним параметром
type FnN = (...args: any[]) => any; // с любым числом параметров,
                      // совпадающих с функциональным типом
```

Любая из приведенных альтернатив точнее, чем `any`. Обратите внимание на использование `any[]` в качестве типа для остальных параметров в последнем примере. Здесь бы подошел и обычный `any`, но он был бы менее точен:

```
const numArgsBad = (...args: any) => args.length; // возвращает any
const numArgsGood = (...args: any[]) => args.length; // возвращает number
```

Это, вероятно, самый распространенный вариант применения типа `any[]`.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Прежде чем использовать `any`, постарайтесь конкретизировать свои ожидания.
- ✓ Отдавайте предпочтение более точным формам `any`, таким как `any[]`, `{[id: string]: any}` или `() => any`, если они способны смоделировать данные более точно.

ПРАВИЛО 40. Скрывайте небезопасные утверждения типов в грамотно типизированных функциях

Сигнатуры типов многих функций легко прописываются, чего нельзя сказать об их реализациях в типобезопасном коде. Проще и надежнее использовать небезопасное утверждение типа, спрятав его внутри функции с верной сигнатурой типа.

Предположим, вы хотите, чтобы функция кэшировала свой последний вызов. Это распространенный способ устранения дорогостоящих вызовов функций фреймворками вроде `React`¹. Напишите обычную обертку `cacheLast`, которая добавит кэширование в функцию:

```
declare function cacheLast<T extends Function>(fn: T): T;
```

Попытка реализации:

```
function cacheLast<T extends Function>(fn: T): T {
  let lastArgs: any[]|null = null;
  let lastResult: any;
  return function(...args: any[]) {
    // ~~~~~
    //          Тип '(...args: any[]) => any' не может быть назначен
    //          для типа 'T'.
  }
```

¹ Работая с `React`, вам следует использовать встроенный хук `useMemo` вместо добавления своего.

```

    if (!lastArgs || !shallowEqual(lastArgs, args)) {
      lastResult = fn(...args);
      lastArgs = args;
    }
    return lastResult;
  };
}

```

У TypeScript нет оснований верить, что эта неустойчивая функция имеет какое-либо отношение к `T`. Но вы-то знаете, что система типов гарантирует ее вызов с верными параметрами и ее возвращаемое значение имеет верный тип. Поэтому не стоит ожидать проблем от утверждения типа:

```

function cacheLast<T extends Function>(fn: T): T {
  let lastArgs: any[] | null = null;
  let lastResult: any;
  return function(...args: any[]) {
    if (!lastArgs || !shallowEqual(lastArgs, args)) {
      lastResult = fn(...args);
      lastArgs = args;
    }
    return lastResult;
  } as unknown as T;
}

```

Это отлично сработает для любой простой функции, предложенной вами. В этой реализации внедрено немало типов `any`, но они находятся вне сигнатуры типа, и код, вызывающий `cacheLast`, о них не знает.

(Безопасна ли такая реализация? В ней есть проблемы. Во-первых, она не проверяет, совпадают ли значения `this` в «успешных» вызовах. Во-вторых, свойства в оригинальной функции были определены, а в оберточной — нет, и их типы могут не совпадать. Но если вы уверены в их совпадении, то реализация в порядке. Написать функцию в типобезопасном варианте гораздо сложнее.)

Функция `shallowEqual` из предыдущего примера, работающая с двумя массивами, легко типизируется и реализуется. Но ситуация с изменением объекта еще интереснее. Как в случае с `cacheLast`, сигнатуру типа функции легко написать:

```

declare function shallowObjectEqual<T extends object>(a: T, b: T):
  boolean;

```

но ее реализация потребует больше внимания, так как нет гарантии, что `a` и `b` имеют одинаковые ключи (правило 54):

```
function shallowObjectEqual<T extends object>(a: T, b: T): boolean {
  for (const [k, aVal] of Object.entries(a)) {
    if (!(k in b) || aVal !== b[k]) {
      // ~~~~ Элемент неявно имеет тип 'any', так как
      //      тип '{}' не имеет сигнатуры индекса.
      return false;
    }
  }
  return Object.keys(a).length === Object.keys(b).length;
}
```

Немного неожиданно, что TypeScript находит ошибку в обращении к `b[k]`, несмотря на подтверждение, что `k in b` верно. Поэтому вам остается одно — преобразовать тип:

```
function shallowObjectEqual<T extends object>(a: T, b: T): boolean {
  for (const [k, aVal] of Object.entries(a)) {
    if (!(k in b) || aVal !== (b as any)[k]) {
      return false;
    }
  }
  return Object.keys(a).length === Object.keys(b).length;
}
```

Это преобразование безвредно, поскольку вы проверили `k in b` и в результате получили исправную функцию с чистой сигатурой типа. Такой подход гораздо лучше, чем проверка равенства объектов с помощью итераций и преобразований по всему коду.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Скрывайте небезопасные преобразования внутри функции с верной сигатурой.

ПРАВИЛО 41. Распознавайте изменяющиеся `any`

В TypeScript тип переменной чаще всего определяется в момент ее объявления. Далее он может быть *уточнен* (например, проверкой `null`), но он не может расширяться и включать новые значения. Существует лишь одно исключение из этого правила, связанное с применением типов `any`.

В JavaScript функцию, генерирующую заданный диапазон чисел, можно написать так:

```
function range(start, limit) {
  const out = [];
  for (let i = start; i < limit; i++) {
    out.push(i);
  }
  return out;
}
```

При ее конвертации в TypeScript вы получите то, что ожидали:

```
function range(start: number, limit: number) {
  const out = [];
  for (let i = start; i < limit; i++) {
    out.push(i);
  }
  return out; // возвращаемый тип выведен как number[]
}
```

Откуда TypeScript знает, что тип `out` — это `number[]`, ведь он инициализирован как `[]`, что допускает массив любого типа?

Рассмотрите подробнее случаи с `out`:

```
function range(start: number, limit: number) {
  const out = []; // тип any[]
  for (let i = start; i < limit; i++) {
    out.push(i); // тип out является any[]
  }
  return out; // тип number[]
}
```

Изначально `out` имеет тип `any[]` — недифференцированный массив. Но при передаче в него значений `number` его тип изменяется на `number[]`.

Это не сужение (правило 22). Тип массива может расширяться при передаче в него различных элементов:

```
const result = []; // тип any[]
result.push('a');
result // тип string[]
result.push(1);
result // тип (string | number)[]
```

В условных конструкциях тип может зависеть от ветки реализации. Здесь продемонстрировано аналогичное поведение типа с единичным значением вместо массива:

```
let val; // тип any
if (Math.random() < 0.5) {
  val = /hello/;
  val // тип RegExp
} else {
  val = 12;
  val // тип number
}
val // тип number | RegExp
```

В последнем случае переменная изначально является `null`. Это часто выясняется, когда вы устанавливаете значение в блоке `try / catch`:

```
let val = null; // тип any
try {
  somethingDangerous();
  val = 12;
  val // тип number
} catch (e) {
  console.warn('alas!');
}
val // тип number | null
```

Так бывает в случаях, когда тип переменной неявно является `any`, а опция `noImplicitAny` включена. Добавление же явного `any` оставляет тип неизменным:

```
let val: any; // тип any
if (Math.random() < 0.5) {
  val = /hello/;
  val // тип any
} else {
  val = 12;
  val // тип any
}
val // тип any
```



Отслеживание `any` в редакторе может запутать, поскольку тип изменяется только при назначении или передаче элемента. При инспектировании типов в строке редактор определяет их как `any` или `any[]`.

Если же вы используете значение до присвоения ему типа `any`, то получите такую ошибку:

```
function range(start: number, limit: number) {
  const out = [];
  //   ~~~ Переменная 'out' неявно имеет тип 'any[]' в местах, где ее тип
  //       не может быть определен
  if (start === limit) {
    return out;
    //   ~~~ Переменная 'out' неявно имеет тип 'any[]'
  }
  for (let i = start; i < limit; i++) {
    out.push(i);
  }
  return out;
}
```

То есть изменяющиеся типы `any` останутся `any`, если вы совершаете в них запись. Если же вы попытаетесь провести их чтение, когда они все еще `any`, то получите ошибку.

Неявные `any` не изменяются в процессе вызовов функций. В следующем примере стрелочная функция срывает вывод типа:

```
function makeSquares(start: number, limit: number) {
  const out = [];
  //   ~~~ Переменная 'out' неявно имеет тип 'any[]' в местах, где тип
  //       не может быть определен.
  range(start, limit).forEach(i => {
    out.push(i * i);
  });
  return out;
  //   ~~~ Переменная 'out' неявно имеет тип 'any[]'
}
```

В подобных случаях попробуйте использовать методы массива `map` и `filter`, чтобы построить массивы одной инструкцией, избежав итерации и изменяющихся `any`. Подробнее об этом читайте в правилах 23 и 27.

Применение изменяющихся `any` подразумевает все те же предосторожности, касающиеся вывода типов. Является ли `(string|number)[]` верным типом для вашего массива? Возможно, им должен быть `number[]`, а `string` вы передали по ошибке? Вы по-прежнему можете использовать явные аннотации типов вместо изменяющихся `any` для повышения качества обнаружения ошибок.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Обычно типы в TypeScript могут быть только *уточнены*, но неявные `any` и `any[]` допускают *изменение*. Вам следует научиться распознавать и понимать эту конструкцию при ее появлении.
- ✓ Рассмотрите применение явной аннотации типов вместо изменяющихся `any` для повышения качества проверки ошибок.

ПРАВИЛО 42. Используйте `unknown` вместо `any` для значений с неизвестным типом

Предположим, вы хотите написать парсер YAML (YAML может представлять тот же набор значений, что и JSON, но также допускает его расширенный синтаксис). Каким должен быть возвращаемый тип вашего метода `parseYAML`?

Возникает желание сделать его `any` (как `JSON.parse`):

```
function parseYAML(yaml: string): any {  
  // ...  
}
```

Но по правилу 38 нужно избегать «заразных» типов `any`, в частности, не допускать их возвращения функциями.

По идее, пользователи должны назначить результату другой тип:

```
interface Book {  
  name: string;  
  author: string;  
}  
const book: Book = parseYAML(`  
  name: Wuthering Heights  
  author: Emily Bronte  
`);
```

Без деклараций типов переменная `book` получит тип `any`, затруднив проверку в местах ее применения:

```
const book = parseYAML(`  
  name: Jane Eyre
```

```

    author: Charlotte Brontë
  `);
  alert(book.title); // нет ошибки, уведомление "undefined" в среде
                    // выполнения
  book('read'); // нет ошибки,
                // выбрасывает "TypeError: book не является функцией"
                // в среде выполнения

```

Более безопасно заставить `parseYAML` возвращать тип `unknown`:

```

function safeParseYAML(yaml: string): unknown {
  return parseYAML(yaml);
}
const book = safeParseYAML(`
  name: The Tenant of Wildfell Hall
  author: Anne Bronte
`);
alert(book.title);
// ~~~~ Объект имеет тип 'unknown'
book("read");
// ~~~~~~ Объект имеет тип 'unknown'

```

Для лучшего понимания типа `unknown` рассмотрите типы `any` с позиции возможного присвоения. И преимущества, и опасности `any` обусловлены двумя его свойствами:

- Типу `any` может быть присвоен любой тип.
- Тип `any` может быть присвоен любому типу¹.

Тип `any` не соответствует системе типов — наборов значений (правило 7), поскольку набор не может одновременно быть и подмножеством, и надмножеством других наборов. Так как модуль проверки типов опирается на наборы значений, применение `any` нарушает его работу.

Тип `unknown` является той альтернативой `any`, которая *согласуется* с системой типов. Тип `any` может быть присвоен `unknown`, но `unknown` может быть присвоен только `unknown` и `any`. (Тип `never`, напротив, может быть присвоен любому типу, но никакой тип не может быть присвоен ему.)

Обращение к свойству значения с типом `unknown` приведет к ошибке. То же самое произойдет при попытке вызвать его или провести с ним арифметические действия. Он не дает широких возможностей, но ошибки, связанные с `unknown`, обяжут вас найти другой подходящий тип:

¹ За исключением `never`.

```
const book = safeParseYAML(`
  name: Villette
  author: Charlotte Brontë
`) as Book;
alert(book.title);
// ~~~~~ Свойство 'title' не существует в типе 'Book'
book('read');
// ~~~~~ Это выражение не может быть вызвано
```

Действительно, мы знаем о типе итогового объекта больше, чем TypeScript.

Тип `unknown` подойдет везде, где вы не уверены в типе ожидаемого значения. Например, в спецификации GeoJSON свойство `properties`, принадлежащее `Feature`, может быть чем угодно в рамках сериализации JSON. Поэтому имеет смысл применить `unknown`:

```
interface Feature {
  id?: string | number;
  geometry: Geometry;
  properties: unknown;
}
```

Утверждение типа — это не единственный способ восстановить тип из объекта `unknown`. Проверка `instanceof` тоже может сделать это:

```
function processValue(val: unknown) {
  if (val instanceof Date) {
    val // тип Date
  }
}
```

Еще можно использовать задаваемую пользователем защиту типа:

```
function isBook(val: unknown): val is Book {
  return (
    typeof(val) === 'object' && val !== null &&
    'name' in val && 'author' in val
  );
}
function processValue(val: unknown) {
  if (isBook(val)) {
    val; // Тип Book
  }
}
```

TypeScript требует обосновать сужение типа `unknown`, поэтому во избежание ошибок в проверках `in` покажите, что `val` является ненулевым типом объекта (поскольку `typeof null === 'object'`).

Иногда вместо `unknown` вы будете встречать обобщенный параметр. Объявить функцию `safeParseYAML` таким образом:

```
function safeParseYAML<T>(yaml: string): T {  
    return parseYAML(yaml);  
}
```

в TypeScript считается плохим стилем. Лучше вернуть тип `unknown`, чтобы побудить пользователей применить утверждение или сузить тип.

`unknown` может также быть использован вместо `any` в двойных утверждениях:

```
declare const foo: Foo;  
let barAny = foo as any as Bar;  
let barUnk = foo as unknown as Bar;
```

Как эквивалент `any` форма `unknown` менее рискованная. Если при рефакторинге вы разорвете два преобразования, `any` незаметно выскользнет и распространится, а `unknown` спровоцирует ошибку.

В завершение отмечу, что вы можете встретить код, где `object` или `{}` используется таким же образом, как `unknown`. Они тоже являются расширенными типами, но при этом более узкими, чем `unknown`:

- Тип `{}` содержит все значения, кроме `null` и `undefined`.
- Тип `object` содержит все непримитивные типы. Это не относится к `true`, `12` или `"foo"`, но касается объектов и массивов.

Применение `{}` было более распространено, пока не был введен тип `unknown`. Теперь `{}` встречается редко. Используйте его, только если уверены, что значения не допускают `null` или `undefined`.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Тип `unknown` — это безопасная альтернатива `any`. Используйте его, когда уверены, что получите значение, но не знаете его тип.
- ✓ Используйте `unknown`, чтобы побудить пользователей к преобразованию или проверке типов.
- ✓ Уясните разницу между `{}`, `object` и `unknown`.

ПРАВИЛО 43. Используйте типобезопасные подходы вместо обезьяньего патча

Одна из наиболее известных особенностей JavaScript — открытость объектов и классов для добавления к ним произвольных свойств. Иногда глобальные переменные на веб-страницах создаются присвоением данных к `window` или `document`:

```
window.monkey = 'Tamarin';
document.monkey = 'Howler';
```

Также данные присоединяются к DOM:

```
const el = document.getElementById('colobus');
el.home = 'tree';
```

Такой стиль особенно распространен в коде, использующем jQuery.

Вы даже можете присоединять свойства к прототипам встроенных объектов:

```
> RegExp.prototype.monkey = 'Capuchin'
"Capuchin"
> /123/.monkey >
"Capuchin"
```

Но результатом этого бывают плохие конструкции. Присоединяя данные к `window` или узлу DOM, вы превращаете их в глобальную переменную и неумышленно устанавливаете зависимости между отдаленными частями программы, провоцируя появление побочных эффектов при любом вызове функции.

Добавление TypeScript привносит еще одну проблему: хотя модуль проверки типов и знает о встроенных свойствах `Document` и `HTMLElement`, он точно не знает о свойствах, которые вы добавили:

```
document.monkey = 'Tamarin';
// ~~~~~ в типе 'Document' нет свойства 'monkey'.
```

Наиболее простым способом устранения этой ошибки будет ввод `any`:

```
(document as any).monkey = 'Tamarin'; // ok
```

Это устроит модуль проверки, но вы пожертвуете безопасностью типов и поддержкой языковых служб:

```
(document as any).monkey = 'Tamarin'; // также ок, опечатка
(document as any).monkey = /Tamarin/; // также ок, неверный тип
```

Наилучшим решением будет вывести данные из `document` или `DOM`. Однако если вы не можете этого сделать (из-за нежелания потерять зависящую от них библиотеку или запущенного процесса переноса JavaScript-приложения), то есть еще несколько хороших вариантов.

Один из них — это аугментация, относящаяся к числу особых возможностей `interface` (правило 13):

```
interface Document {
  /** Разновидность обезьяньего патча */
  monkey: string;
}

document.monkey = 'Tamarin'; // ок
```

Она имеет несколько преимуществ перед `any`:

- Безопасность типов. Модуль проверки будет указывать на опечатки и неверно назначенные типы.
- Возможность добавить к свойству документацию (пункт 48).
- Работа функции автоподстановки для свойства.
- Присутствие записи о том, в чем конкретно заключается обезьяний патч.

В контексте модуля (то есть файла TypeScript, который использует импорт/экспорт), следует добавить объявление `global`:

```
export {};
declare global {
  interface Document {
    /** Genus or species of monkey patch */
    monkey: string;
  }
}
document.monkey = 'Tamarin'; // ок
```

Основная сложность в применении аугментации связана с ее диапазоном. Во-первых, она применяется глобально, и вы не можете скрыть ее от других участков кода и библиотек. Во-вторых, если присвоить свойство во время выполнения приложения, то не получится отложить аугментацию до завершения. Это особенно проблематично при патчинге HTML-элементов, не все из которых успели получить свои свойства. Можно объявить свойство с более точным типом `string|undefined`, но он усложнит работу с типами.

Другой подход связан с использованием более точного приведения:

```
interface MonkeyDocument extends Document {  
    /** Разновидность манкипатчинга */  
    monkey: string;  
}  
  
(document as MonkeyDocument).monkey = 'Macaque';
```

TypeScript согласен, так как `Document` и `MonkeyDocument` разделяют свойства (правило 9). Вы же при назначении получаете безопасные типы. Диапазон становится более управляемым: нет глобальной модификации типа `Document`, а введение нового типа в диапазон происходит только при условии, что вы его туда импортируете. Всегда используйте приведение (либо вводите новую переменную) при обращении к пропатченному свойству.

Но вы можете рассмотреть обезьяний патч как стимул привести рефакторинг в более структурированный вид.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Лучше иметь структурированный код, чем хранить данные в глобальных переменных или DOM.
- ✓ Если вы вынуждены хранить данные во встроенных типах, используйте один из типобезопасных подходов (аугментацию или приведение через пользовательский интерфейс).
- ✓ Помните о сложности аугментации, связанной с диапазоном. Предпочитайте утверждения типов.

ПРАВИЛО 44. Отслеживайте зону охвата типов для сохранения типобезопасности

Когда вы добавляете аннотации типов для значений с неявными типами `any` и включаете опцию `noImplicitAny`, оказываетесь ли вы защищены? Нет — типы `any` по-прежнему могут попасть в программу двумя способами:

Из явных any

Даже если вы следуете советам из правил 38 и 39, делая типы `any` более узкими и специфичными, они все равно остаются типами `any`. В частности, такие типы, как `any[]` и `{[key: string]: any}`, после индексирования в них становятся обычными `any`, которые могут распространить свое воздействие по всему коду.

От сторонних деклараций типов

Типы `any` из файла декларации `@types` внедряются незаметно: даже если у вас включена `noImplicitAny` и вы ни разу не вводили `any`, они все равно разлетятся по коду.

Из-за негативного влияния типов `any` на безопасность типов и ваш опыт (правило 5) нужно отслеживать их количество в базе кода. Например, включить для этого пакет `type-coverage` из npm (Node Packet Manager):

```
$ npx type-coverage
9985 / 10117 98.69%
```

Здесь указано, что из общего числа символов проекта (10 117) 9985 (98,69 %) имели тип, отличный от `any` или псевдонима `any`. Если какие-либо изменения приведут к появлению типа `any`, вы увидите соответствующее уменьшение в процентах.

Этот процентный показатель может также продемонстрировать, насколько хорошо вы следуете советам других правил этой главы. Использование `any` с суженным диапазоном снизит число символов, связанных с типами `any`. Аналогичный эффект вызовет использование более конкретных форм вроде `any[]`.

Сбор дополнительной информации о зоне охвата типов также может пригодиться. Запуск `type-coverage` с флагом `-detail` выдаст информацию о местонахождении каждого типа `any` в коде:

```
$ npx type-coverage --detail
path/to/code.ts:1:10 getColumnInfo
path/to/module.ts:7:1 pt2
...
```

Явные `any` часто являются результатом спешки: не было времени разобраться с ошибкой или решили прописать тип позже.

Приведение `any` может помешать типам проникать в нужные места. Возможно, вы создали приложение, работающее с табличными данными, и нуждались в функции с единственным параметром, которая создавала некий столбец описания:

```
function getColumnInfo(name: string): any {
    return utils.buildColumnInfo(appState.dataSchema, name); // возвращает any
}
```

Функция `utils.buildColumnInfo` в определенной точке возвращает `any`. Вы добавили в функцию комментарий и явную аннотацию `:any`, но позднее ввели тип для `ColumnInfo`, и `utils.buildColumnInfo` больше не возвращает `any`. Теперь аннотация `any` исключает важную информацию типа. Вам необходимо от этого избавиться!

Особенно опасно наделять типом `any` целый модуль:

```
declare module 'my-module';
```

Теперь вы можете импортировать из `my-module` что угодно без ошибки. Все эти символы имеют типы `any`, и значения, переданные через них, получают тот же тип:

```
import {someMethod, someSymbol} from 'my-module'; // ok

const pt1 = {
    x: 1,
    y: 2,
}; // тип {x: number, y: number}
const pt2 = someMethod(pt1, someSymbol); // ok, тип pt2 any
```

Поскольку при использовании модуль по-прежнему выглядит правильно типизированным, легко забыть, что вы (или ваш коллега) его заглушили. Поэтому стоит иногда следить за `any`. Возможно, для модуля появились официальные декларации типов или вы научились сами прописывать типы и готовы предложить их сообществу.

Еще одна причина появления `any` вместе со сторонними декларациями типов — это наличие в них бага. Например, декларации ошибочно объявляют, что функция возвращает тип объединения (правило 29). При первом использовании функции этот факт мог не показаться вам достойным исправления и вы просто использовали утверждение `any`. Однако декларации могли получить исправления или пришло время исправить их самостоятельно.

Тип `any` в вашем коде может потерять актуальность, а баг в декларации типов может быть исправлен. Отслеживайте зоны охвата типов, чтобы не упустить такие случаи.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Даже при включенной опции `noImplicitAny` типы `any` могут пробраться в код либо через явные `any`, либо через сторонние декларации типов (`@types`).
- ✓ Отслеживайте, насколько хорошо типизирована ваша программа. Это подтолкнет вас к регулярному пересмотру решений об использовании типов `any` и повысит типобезопасность кода.

Декларации типов и @types

В любом языке управление зависимостями может быть несколько запутанным, и TypeScript не исключение. Эта глава научит вас работать с зависимостями и создавать собственные файлы деклараций типов. Новые хорошие декларации типов будут полезны не только вам, но и всему сообществу TypeScript.

ПРАВИЛО 45. Размещайте TypeScript и @types в devDependencies

Node Packet Manager широко применяется в разработке на JavaScript. Он обеспечивает и репозиторий библиотек JavaScript (реестр npm), и возможность определить, от каких их версий вы зависите (`package.json`).

Реестр npm проводит различие между видами зависимостей, относящихся к отдельным разделам `package.json`:

Dependencies

Это пакеты, необходимые для запуска кода JavaScript. Если вы импортируете `Lodash` во время выполнения, то она отправится в `dependencies`. Когда вы публикуете код в npm, а другой пользователь его устанавливает, будут установлены и эти зависимости (известные как переходящие).

devDependencies

Это пакеты, которые используются для разработки и тестирования кода, но не требуются при его выполнении. Тестовая среда — это пример `devDependency`. В отличие от `dependencies`, они *не* переходят при установке вместе с вашими пакетами.

peerDependencies

Это пакеты, которые требуются во время выполнения, но вы не несете ответственности за соблюдение их версий. К ним относятся плагины. Ваш плагин jQuery совместим с рядом версий самой jQuery, но лучше, чтобы нужную версию выбрал пользователь.

Из этих трех видов **dependencies** и **devDependencies** более распространены. Работая в TypeScript, будьте внимательны к зависимости. Поскольку TypeScript — это инструмент разработки и его типы не существуют во время выполнения (правило 3), пакеты, относящиеся к нему, главным образом принадлежат к разряду **devDependencies**.

Первая важная зависимость — это сам TypeScript. Можно установить его в масштабе всей системы, но есть две серьезные причины этого не делать:

- Нет гарантии, что у вас и ваших коллег будут установлены одинаковые версии.
- Добавляется еще один шаг к установке проекта.

Поместите TypeScript в пакет **devDependency**, и вы с коллегами будете получать верную версию при запуске **npm install**. А обновление версии при этом будет происходить аналогично другим пакетам.

Ваш IDE и система сборки будут легко обнаруживать версию TypeScript, установленную таким образом. В командной строке вы можете использовать **npx** для запуска версии **tsc**, установленной **npm**:

```
$ npx tsc
```

Следующий вид зависимости — это *зависимости по типам* или **@types**. Если сама библиотека не содержит декларации типов TypeScript, то вы можете найти типизации в DefinitelyTyped — коллекции определений типов для библиотек JavaScript, развиваемой сообществом. Ее определения типов опубликованы в реестре **npm** по диапазонам: **@types/jquery** содержит определения для jQuery, **@types/lodash** — для Lodash и т. д. Эти пакеты **@types** содержат только *типы*, но не реализации.

Ваши зависимости **@types** также должны относиться к **devDependencies**, даже если сам пакет является прямой зависимостью. Например, для установления зависимости от React и его деклараций типов вы можете запустить:

```
$ npm install react
$ npm install --save-dev @types/react
```

Результат следующим образом отразится в файле `package.json`:

```
{
  "devDependencies": {
    "@types/lodash": "^16.8.19",
    "typescript": "^3.5.3"
  },
  "dependencies": {
    "react": "^16.8.6"
  }
}
```

Смысл здесь в том, что вам следует публиковать код не TypeScript, а JavaScript, который при запуске не зависит от `@types`. Сложности с зависимостями `@types` будут подробно рассмотрены в следующем пункте.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Избегайте установки TypeScript во всей системе. Поместите его в пакет `devDependency` вашего проекта, чтобы все участники команды использовали одну версию.
- ✓ Помещайте зависимости `@types` в `devDependencies`, а не в `dependencies`. Если `@types` необходимы вам при выполнении, то, возможно, следует перестроить процесс.

ПРАВИЛО 46. Проверяйте совместимость трех версий, задействованных в декларациях типов

Разработчики не любят возиться с зависимостями. Конечно, удобнее просто использовать библиотеку и не контролировать ее переходящие зависимости.

Плохо то, что TypeScript эту ситуацию не улучшает. Честно говоря, он даже *усложняет* управление зависимостями, заставляя вас задумываться о трех версиях:

- версии пакета;
- версии его деклараций типов (`@types`);
- версии TypeScript.

Их несоответствие друг другу ведет к ошибкам, которые не будут прямо указывать на проблемы с зависимостями. Но как говорится, делайте вещи настолько простыми, насколько возможно, но не более. Понимание всей сложности управления пакетами в TypeScript поможет вам находить и устранять проблемы.

Работа с зависимостями происходит следующим образом. Вы устанавливаете пакет в качестве прямой зависимости, а его типы — как `devDependency` (правило 45):

```
$ npm install react
+ react@16.8.6

$ npm install --save-dev @types/react
+ @types/react@16.8.19
```

Основная и второстепенная версии (16.8) совпадают, но версии патча (.6 и .19) — нет. `@types` описывают API `react` версии 16.8. При условии что модуль `react` соблюдает семантическую чистоту контроля версий, версии патча (16.8.1, 16.8.2, ...) не будут менять его публичный API и требовать обновления деклараций типов. Но *сами* декларации могут иметь баги и недоработки, исправленные в следующем патче. В нашем примере в декларациях было проведено больше обновлений, чем в библиотеке (19 против 6).

Здесь возможно появление сложностей.

Во-первых, вы можете обновить библиотеку, забыв обновить декларации типов, и получите ошибки типов при использовании новых возможностей библиотеки. Если в ней были произведены критические изменения, то могут возникнуть ошибки при выполнении, несмотря на успешную проверку типов.

Для решения этой проблемы просто обновите декларации типов, чтобы их версии синхронизировались с библиотекой. Если декларации не обновлялись, у вас есть два пути. Можно использовать аугментацию в проекте для добавления нужных вам функций и методов либо отправить свои обновленные декларации сообществу.

Во-вторых, декларации типов могут опережать библиотеку. Если вы пользовались библиотекой без ее типизаций (возможно, наделили ее типом `any`, используя `declare module`), попробуйте установить типы. Если были выпущены новые релизы библиотеки (признаки этого будут обратны признакам из предыдущего примера), модуль проверки типов сравнит ваш код с последней версией API, вы же при выполнении будете использовать

старую версию. В качестве решения можете обновить версии библиотеки или использовать более ранние версии деклараций типов.

В-третьих, декларации типов могут требовать версию TypeScript новее той, что вы используете в проекте. При разработке системы типов в TypeScript было уделено много внимания максимальной точности типизации популярных библиотек, работающих с JavaScript, вроде Lodash, React и Ramda. Поэтому декларации типов для этих библиотек стремятся использовать последние возможности для обеспечения лучшей безопасности типов.

При таком несоответствии вы получите ошибки типов в декларациях `@types`. Можно обновить TypeScript, использовав более старые версии деклараций типов, либо заглушить типы посредством `declare module`. Библиотека способна обеспечить разные декларации типов для разных версий TypeScript с помощью `typesVersions`, но это редкий случай — на момент написания книги в DefinitelyTyped эту возможность поддерживают менее 1 % пакетов.

Чтобы установить `@types` для конкретной версии TypeScript, можно использовать:

```
npm install --save-dev @types/lodash@ts3.1
```

Согласование версий библиотек с их типами требует много усилий и не всегда оказывается корректным. Однако чем популярнее библиотека, тем вероятнее успех.

В-четвертых, вы можете столкнуться с повторами зависимостей `@types`. Предположим, вы зависите от `@types/foo` и `@types/bar`. Если `@types/bar` зависит от несовместимой версии `@types/foo`, тогда npm постарается исправить это, установив обе версии и разместив одну из них во вложенной папке:

```
node_modules/  
  @types/  
    foo/  
      index.d.ts @1.2.3  
  bar/  
    index.d.ts  
    node_modules/  
      @types/  
        foo/  
          index.d.ts @2.3.4
```

Этот вариант больше подходит для узловых модулей, используемых при выполнении, чем для деклараций типов, которые существуют в плоском глобальном пространстве имен. Вы получите ошибки либо о повторях

деклараций, либо о невозможности их слияния. Можно отследить, почему возникли повторения деклараций, запустив `npm ls @types/foo`. Простым решением будет обновить ваши зависимости в `@types/foo` либо `@types/bar` так, чтобы они стали совместимыми. Передающиеся зависимости `@types` вроде упомянутых часто становятся причиной проблем. Советы по их избеганию при публикации типов читайте в правиле 51.

К некоторым пакетам, особенно написанным в TypeScript, привязывают их собственные декларации типов. Обычно это отражено в поле `"types"`, содержащемся в их `package.json`, которое указывает на файл `.d.ts`:

```
{
  "name": "left-pad",
  "version": "1.3.0",
  "description": "String left pad",
  "main": "index.js",
  "types": "index.d.ts",
  // ...
}
```

Решает ли это все наши проблемы? Я бы не стал спрашивать, если бы ответ был «да».

Привязка типов решает проблему несовместимости версий, особенно если сама библиотека написана в TypeScript, а декларации типов сгенерированы `tsc`. Но она также имеет и собственные проблемы.

Во-первых, что, если в привязанных типах есть ошибка, которую не исправить аугментацией? Или типы исправно работали после публикации, но с появлением новой версии TypeScript возникла ошибка? С `@types` вы зависите от реализации библиотеки, а не от ее деклараций типов. А при использовании привязанных типов одна плохая декларация типа может помешать использовать последующие версии TypeScript. По мере разработки новых версий TypeScript Microsoft сверяет работоспособность DefinitelyTyped со всеми декларациями типов. Несоответствия сразу исправляются.

Во-вторых, что если ваши типы зависят от деклараций типов другой библиотеки? Обычно это касается `devDependency` (правило 45). Если вы опубликуете свой модуль, а другой пользователь его установит, он не получит ваши `devDependencies`, и возникнут ошибки типов. Можно не делать зависимость прямой, чтобы JavaScript-пользователи не устанавливали модули `@types` без причины (правило 51). Или, публикуя типы на DefinitelyTyped, объявить зависимость типа, чтобы ее получили только ваши TypeScript-пользователи.

В-третьих, что если вам нужно исправить проблему в декларациях типов старой версии вашей библиотеки? Сможете ли вы вернуться и выпустить обновление патча? В DefinitelyTyped есть механизмы, одновременно поддерживающие декларации типов для разных версий одной библиотеки, — это тяжело сделать в проекте самостоятельно.

В-четвертых, насколько вы внимательны к патчам? Вспомните версии `react` и `@types/react` из начала правила. В том случае было в три раза больше обновлений деклараций типов, чем самой библиотеки. Поддерживаемый и развиваемый сообществом DefinitelyTyped способен справиться с таким объемом. В частности, если сопровождающий библиотеки не проверяет наличие патча в течение пяти дней, это делает глобальный сопровождающий. Сможете ли вы поддерживать такой же темп обновления для вашей библиотеки?

Управлять зависимостями в TypeScript нелегко, но грамотно прописанные декларации типов помогут вам понять, как правильно использовать библиотеки. Учитывайте совместимость трех версий.

Если вы занимаетесь публикацией пакетов, то взвесьте все плюсы и минусы привязки деклараций типов вместо их размещения в DefinitelyTyped. Официально рекомендуется привязывать типы, только если библиотека написана на TypeScript. Это исправно работает, так как `tsc` может автоматически генерировать декларации типов (используя опцию `declaration` компилятора). Рукописные декларации типов с большей вероятностью будут содержать ошибки и потребуют больше обновлений. Если вы размещаете декларации типов в DefinitelyTyped, то сообщество поможет вам в их сопровождении.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ В зависимости `@types` задействовано три версии: версия библиотеки, версия `@types` и версия самого TypeScript.
- ✓ При обновлении библиотеки убедитесь, что вы также обновляете соответствующую зависимость `@types`.
- ✓ Взвесьте плюсы и минусы привязки типов в сравнении с публикацией их на DefinitelyTyped. Отдавайте предпочтение привязке, только если ваша библиотека написана в TypeScript.

ПРАВИЛО 47. Экспортируйте все типы, появляющиеся в публичных API

Используя TypeScript, вы, скорее всего, захотите убедиться, что `type` или `interface` не экспортированы. К счастью, инструменты отображения между типами в TypeScript достаточно обширны для того, чтобы в качестве пользователя библиотеки вы почти всегда могли обратиться к нужному типу. Если же вы автор библиотеки, то для начала просто должны экспортировать типы. Как только тип появляется в декларации функции, он эффективно экспортируется. Поэтому вы можете делать элементы явными.

Предположим, вы хотите создать некие секретные неэкспортируемые типы:

```
interface SecretName {
  first: string;
  last: string;
}

interface SecretSanta {
  name: SecretName;
  gift: string;
}

export function getGift(name: SecretName, gift: string): SecretSanta {
  // ...
}
```

Будучи пользователем вашего модуля, я не могу напрямую импортировать `SecretName` или `SecretSanta`, но могу — `getGift`. Однако эти типы появятся в сигнатуре экспортированной функции, и я смогу их извлечь, например, с помощью обобщенных типов `Parameters` и `ReturnType`:

```
type MySanta = ReturnType<typeof getGift>; // SecretSanta
type MyName = Parameters<typeof getGift>[0]; // SecretName
```

Если таким образом вы хотели сохранить подвижность, то ваши карты раскрыты. Вы уже оказались к ним привязаны, поместив их в публичный API. Сделайте пользователям одолжение и экспортируйте их.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Экспортируйте типы, которые в любой форме появляются в любом открытом методе. Пользователи все равно смогут их извлечь, поэтому лучше упростите им задачу.

Правило 48. Используйте TSDoc для комментариев в API

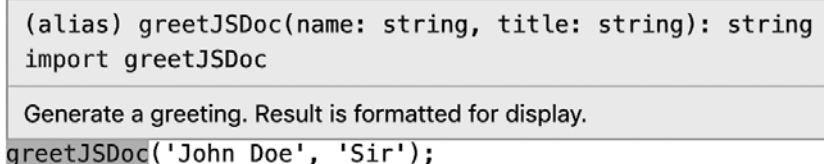
Вот функция TypeScript, генерирующая приветствие:

```
// Создает приветствие. Результат отформатирован для отображения.
function greet(name: string, title: string) {
    return `Hello ${title} ${name}`;
}
```

Автор оставил комментарий, описывающий действия функции. Но документацию, предназначенную для чтения функции пользователями, лучше создавать с помощью комментариев JSDoc:

```
/** Создает приветствие. Результат отформатирован для отображения. */
function greetJSDoc(name: string, title: string) {
    return `Hello ${title} ${name}`;
}
```

Дело в том, что почти все редакторы при вызове функции показывают комментарии JSDoc (рис. 6.1).

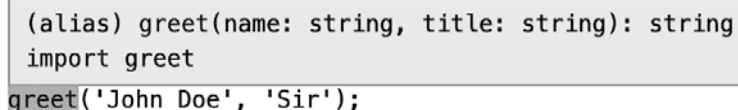


```
(alias) greetJSDoc(name: string, title: string): string
import greetJSDoc

Generate a greeting. Result is formatted for display.
greetJSDoc('John Doe', 'Sir');
```

Рис. 6.1. Комментарии в стиле JSDoc обычно отображаются во всплывающих подсказках в редакторе

Строковые же комментарии в таких случаях не отображаются (рис. 6.2).



```
(alias) greet(name: string, title: string): string
import greet
greet('John Doe', 'Sir');
```

Рис. 6.2. Строковые комментарии обычно не отображаются во всплывающих подсказках

Языковые службы TypeScript также поддерживают эту функцию, и вам не стоит упускать ее преимуществ. Если комментарий описывает публичный API, то он должен быть прописан как JSDoc. В контексте TypeScript такие комментарии иногда называют TSDoc. Вы также можете использовать многие из обычных условных конструкций вроде `@param` и `@returns`:

```
/**
 * Создать приветствие.
 * @param — назвать Name приветствуемого.
 * @param — обратиться к его званию.
 * @returns — приветствие, оформленное для человеческого восприятия.
 */
function greetFullTSDoc(name: string, title: string) {
  return `Hello ${title} ${name}`;
}
```

Это позволит редакторам при написании вызова функции показывать документацию, относящуюся к каждому отдельному параметру (рис. 6.3).

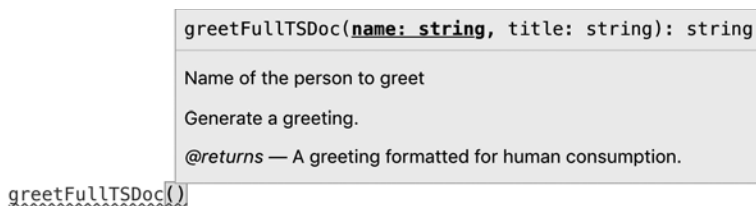


Рис. 6.3. Аннотация `@param` позволяет редактору отображать документацию для текущего параметра по мере его ввода

Вы также можете использовать TSDoc с определениями типов:

```
/** Измерение проведено в определенном месте в конкретное время. */
interface Measurement {
  /** Где было проведено измерение? */
  position: Vector3D;
  /** Когда было проведено измерение? В секундах от точки отсчета. */
  time: number;
  /** Наблюдаемый импульс */
  momentum: Vector3D;
}
```

Рассматривая отдельные поля в объекте `Measurement`, вы получите контекстную документацию (рис. 6.4).

```
const m: Measurement = {
  (property) Measurement.time: number
  When was the measurement made? In seconds since epoch.
  time: (new Date().getTime()) / 1000,
  position: {x: 0, y: 0, z: 0},
  momentum: {x: 1, y: 2, z: 3},
};
```

Рис. 6.4. TSDoc для поля отображается при наведении указателя мыши на это поле в редакторе

Комментарии TSDoc формируются согласно языку разметки (Markdown), поэтому вы вполне можете использовать жирный шрифт, курсив или маркированный список (рис. 6.5).

```
/**
 * Этот интерфейс имеет три свойства:
 * 1. x
 * 2. y
 * 3. z
 */
interface Vector3D {
  x: number;
  y: number;
  z: number;
}
```

```
/**
 * This interface
 * 1. x
 * 2. y
 * 3. z
 */
export interface Vector3D {
  x: number;
  y: number;
  z: number;
}
```

interface Vector3D

This *interface* has **three** properties:

- 1. x
- 2. y
- 3. z

Рис. 6.5. TSDoc-комментарии

Старайтесь избегать написания в документации целых эссе, ведь лучшие комментарии лаконичны.

JSDoc включает некоторые условные конструкции для определения информации типа (`@param {string} name ...`), но вам следует избегать их в угоду типам TypeScript (правило 30).

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Используйте комментарии в формате JSDoc / TSDoc для документирования экспортируемых функций, классов и типов, чтобы редакторы смогли продемонстрировать эту информацию пользователям.
- ✓ Используйте `@param`, `@returns` и язык разметки для форматирования.
- ✓ Избегайте добавления информации типа в документацию (правило 30).

ПРАВИЛО 49. Определяйте тип `this` в обратных вызовах

Ключевое слово `this` является одним из наиболее запутывающих элементов в JavaScript. В отличие от `let` или `const`, чей диапазон ограничен лексически, `this` имеет динамический диапазон. Его значение зависит не от того, как оно было определено, а от того, как было вызвано.

Чаще всего `this` используется в классах, где оно указывает на текущий экземпляр объекта:

```
class C {
  vals = [1, 2, 3];
  logSquares() {
    for (const val of this.vals) {
      console.log(val * val);
    }
  }
}

const c = new C();
c.logSquares();
```

Отсюда логируется:

```
1
4
9
```

Взгляните, что произойдет, если вы попытаете поместить `logSquare` в переменную и вызвать ее:

```
const c = new C();
const method = c.logSquares;
method();
```

Такой вариант при выполнении выдаст ошибку:

```
Uncaught TypeError: Cannot read property 'vals' of undefined
```

Проблема в том, что `c.logSquares()` совершает два действия: вызывает `C.prototype.logSquares` и привязывает значение `this` к `c` функции. Извлекая ссылку на `logSquares`, мы разделяем эти действия, и `this` становится `undefined`.

JavaScript дает вам полный контроль над привязкой `this`. Вы можете решить проблему, используя `call` для явного определения `this`:

```
const c = new C();
const method = c.logSquares;
method.call(c); // снова логирует квадраты значений
```

Привязывать `this` можно к чему угодно. Поэтому в библиотеках значение `this` — это часть API. Даже DOM находит применение для `this`. Например, в обработчике событий:

```
document.querySelector('input')!.addEventListener('change', function(e) {
  console.log(this); // логирует вводный (Input) элемент, который запустил
                     // событие.
});
```

Привязка `this` часто встречается в контекстах обратных вызовов вроде этого. Если же вы, например, хотите определить обработчик `onClick` в классе, то можете попробовать сделать так:

```
class ResetButton {
  render() {
    return makeButton({text: 'Reset', onClick: this.onClick});
  }
  onClick() {
    alert(`Reset ${this}`);
  }
}
```

Когда `Button` вызовет `onClick`, он сообщит, что `"Reset undefined"`. Упс! Как обычно, виновна здесь привязка `this`. Стандартным решением будет создать связанную версию метода в конструкторе:

```
class ResetButton {
  constructor() {
    this.onClick = this.onClick.bind(this);
  }
  render() {
    return makeButton({text: 'Reset', onClick: this.onClick});
  }
  onClick() {
    alert(`Reset ${this}`);
  }
}
```

Определение `onClick() { ... }` задает свойство в `ResetButton.prototype`, которое разделяется всеми экземплярами `ResetButton`. Когда вы привязываете `this.onClick = ...` в конструкторе, он создает свойство с именем `onClick` в экземпляре `ResetButton` с `this`, привязанным к этому экземпляру. Свойство экземпляра `onClick` идет прежде свойства прототипа `onClick` в последовательности поиска, значит, `this.onClick` обращается к связанной функции в методе `render()`.

Иногда удобно использовать сокращение для назначения привязки:

```
class ResetButton {
  render() {
    return makeButton({text: 'Reset', onClick: this.onClick});
  }
  onClick = () => {
    alert(`Reset ${this}`); // "this" всегда относится к экземпляру
                           // ResetButton.
  }
}
```

Здесь мы заменили `onClick` на стрелочную функцию. Она будет определять новую функцию каждый раз, когда `ResetButton` будет построен с `this`, имеющим корректное значение. Взгляните на код JavaScript, который таким образом генерируется:

```
class ResetButton {
  constructor() {
    var _this = this;
    this.onClick = function () {
      alert("Reset " + _this);
    };
  }
}
```

```

    }
    render() {
        return makeButton({ text: 'Reset', onClick: this.onClick });
    }
}

```

При чем здесь TypeScript? Так как привязка `this` — это часть JavaScript, TypeScript ее моделирует. Это означает, что если вы пишете (или типизируете) библиотеку, которая устанавливает значение `this` в обратных вызовах, то это также нужно смоделировать.

Для этого добавьте параметр `this` в обратный вызов:

```

function addKeyListener(
    el: HTMLElement,
    fn: (this: HTMLElement, e: KeyboardEvent) => void
) {
    el.addEventListener('keydown', e => {
        fn.call(el, e);
    });
}

```

Параметр `this` является не просто позиционным аргументом. Если вызвать его с двумя параметрами, получится следующее:

```

function addKeyListener(
    el: HTMLElement,
    fn: (this: HTMLElement, e: KeyboardEvent) => void
) {
    el.addEventListener('keydown', e => {
        fn(el, e);
        // ~ Ожидается 1 аргумент, но получено 2.
    });
}

```

Так даже лучше. TypeScript принудит вас вызывать функцию с корректным контекстом `this`:

```

function addKeyListener(
    el: HTMLElement,
    fn: (this: HTMLElement, e: KeyboardEvent) => void
) {
    el.addEventListener('keydown', e => {
        fn(e);
        // ~~~~ Контекст 'this' типа 'void' не может быть назначен
        //      для 'this' метода в типе 'HTMLElement'.
    });
}

```

Будучи пользователем этой функции, вы можете сослаться на `this` в обратном вызове, получив полную безопасность типов:

```
declare let el: HTMLElement;
addKeyListener(el, function(e) {
  this.innerHTML; // ok, "this" имеет тип HTMLElement.
});
```

Конечно, если вы используете здесь стрелочную функцию, то измените значение `this`, чтобы TypeScript уловил проблему:

```
class Foo {
  registerHandler(el: HTMLElement) {
    addKeyListener(el, e => {
      this.innerHTML;
      // ~~~~~ Свойство 'innerHTML' не существует в типе 'Foo'
    });
  }
}
```

Не забывайте о `this`! Если вы добавите значение `this` в обратные вызовы, то оно станет частью вашего API и нужно будет добавить его в декларации типов.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Разберитесь, как работает привязка `this`.
- ✓ Назначьте тип для `this` в обратных вызовах, если он является частью вашего API.

ПРАВИЛО 50. Лучше условные типы, чем перегруженные декларации

Как бы вы написали декларацию типа для этой функции на JavaScript?

```
function double(x) {
  return x + x;
}
```

В `double` может быть передан либо `string`, либо `number`. Значит, можно использовать тип объединения:

```
function double(x: number|string): number|string;
function double(x: any) { return x + x; }
```

(Эти примеры исходят из концепции TypeScript о перегрузке функции, описанной в правиле 3.)

Хотя эта декларация и верна, ей недостает точности:

```
const num = double(12); // string | number
const str = double('x'); // string | number
```

Когда в `double` передается `number`, она возвращает `number`. Если передается `string` — возвращается `string`. Но декларация упускает этот нюанс и производит типы, с которыми трудно работать.

Вы можете попробовать удержать связь, используя обобщение:

```
function double<T extends number|string>(x: T): T;
function double(x: any) { return x + x; }
```

```
const num = double(12); // тип 12
const str = double('x'); // тип "x"
```

К несчастью, теперь типы стали чрезмерно точными. При передаче типа `string` декларация `double` будет возвращать `string`, и это верно. Но при передаче типа строкового литерала возвращаемым типом будет тот же строковый литерал. Это неверно, так как результатом удвоения (`double`) `' x'` будет `' xx'`, а не `' x'`.

Вместо обобщения можно сделать несколько деклараций типов. При том что TypeScript позволяет писать лишь одну реализацию функции, он дает возможность создавать неограниченное число деклараций типов:

```
function double(x: number): number;
function double(x: string): string;
function double(x: any) { return x + x; }
```

```
const num = double(12); // тип number
const str = double('x'); // тип string
```

Это уже лучше! Но в декларации по-прежнему присутствует баг: она будет работать только со значениями `string` или `number`, но не с теми, которые могут быть либо `string`, либо `number`:

```
function f(x: number|string) {
  return double(x);
    // ~ Аргумент типа 'string | number' не может быть назначен
    //   параметру типа 'string'.
}
```

Такой вызов `double` безопасен и должен возвращать `string|number`. Когда вы перегружаете декларации типов, TypeScript обрабатывает их одну за другой, пока не найдет совпадение. Ошибка, которую вы видите, — это результат провала последней перегрузки (версия со `string`) из-за того, что `string|number` не может быть назначен для `string`.

Хоть вы и могли бы исправить эту проблему, добавив третью перегрузку `string|number`, лучшее решение — использовать *условный тип*. Он является своеобразными `if` (условной конструкцией) в пространстве типов и идеально подходит для подобных ситуаций:

```
function double<T extends number | string>(
  x: T
): T extends string ? string : number;
function double(x: any) { return x + x; }
```

В отличие от обобщения, этот подход имеет более проработанный возвращаемый тип. Вы можете читать условный тип как троичный (`?:`) оператор в JavaScript:

- Если `T` является подмножеством `string` (то есть `string`, строкового литерала или объединения строковых литералов), тогда возвращаемый тип будет `string`.
- В противном случае вернется `number`.

С такой декларацией все наши примеры заработают:

```
const num = double(12); // number
const str = double('x'); // string

// function f(x: string | number): string | number
function f(x: number|string) {
  return double(x);
}
```

Пример `number|string` работает, так как условные типы охватывают больше возможностей, чем объединения. Когда `T` является `number|string`, TypeScript распознает условный тип следующим образом:

```
(number|string) extends string ? string : number
-> (number extends string ? string : number) |
   (string extends string ? string : number)
-> number | string
```

Хотя декларации типов, использующие перегрузку, легче в написании, вариант с использованием условных типов оказывается более корректным, так как сводится к объединению отдельных случаев. Тогда как перегрузки обрабатываются по отдельности, условные типы модуль проверки может анализировать как единое выражение, распространяя их за пределы объединений. Если вы вдруг соберетесь написать перегруженные декларации типов, подумайте: возможно, лучше использовать для этого условные типы.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Применяйте условные типы вместо перегруженных деклараций. Распределяясь по объединениям, условные типы позволяют декларациям поддерживать типы объединений без использования дополнительных перегрузок.

ПРАВИЛО 51. Зеркалируйте типы для разрыва зависимостей

Предположим, вы написали библиотеку для парсинга CSV-файлов. Ее API прост: вы передаете содержимое файла и получаете назад список объектов, отображающих имена колонок в значениях. Для удобства пользователей NodeJS вы позволяете содержимому быть либо `string`, либо `Buffer` NodeJS:

```
function parseCSV(contents: string | Buffer): {[column: string]: string}[] {
  if (typeof contents === 'object') {
    // Это буфер
    return parseCSV(contents.toString('utf8'));
  }
  //...
}
```

Тип `Buffer` определяется по декларациям типов NodeJS, которые необходимо установить:

```
npm install --save-dev @types/node
```

При публикации библиотеки вы прикрепляете к ней эти декларации. Поскольку они зависят от типов NodeJS, вы включите их как `devDependency` (правило 45), и начнутся жалобы от двух групп пользователей:

- Разработчики JavaScript спросят, что это за модули `@types`, от которых они зависят.
- Веб-разработчики TypeScript поинтересуются, почему они зависят от NodeJS.

И они будут правы. Поведение `Buffer` не является основополагающим и касается только тех пользователей, кто уже применяет NodeJS. А декларация в `@types/node` касается только тех пользователей NodeJS, которые применяют TypeScript.

Структурная типизация TypeScript (правило 4) поможет вам выйти из положения. Вместо использования декларации `Buffer` из `@types/node` вы можете написать свою декларацию, содержащую исключительно нужные вам методы и свойства. В этом случае кодировку (`encoding`) поддержит только метод `toString`:

```
interface CsvBuffer {  
    toString(encoding: string): string;  
}  
function parseCSV(contents: string | CsvBuffer):  
    {[column: string]: string}[] {  
    // ...  
}
```

Такой интерфейс существенно меньше полноценного, но при этом он получает из `Buffer` все необходимое. Благодаря совместимости типов вызов `parseCSV` с реальным `Buffer` работает в NodeJS-проекте:

```
parseCSV(new Buffer("column1,column2\nval1,val2", "utf-8")); // ok
```

Если ваша библиотека зависит от типов для другой библиотеки (не от ее реализации), примените зеркалирование нужных вам деклараций, чтобы облегчить пользователям работу с программой.

Если вы зависите от реализации библиотеки, то также можете применить этот подход для избежания зависимости от ее типизаций. Однако по мере

роста зависимости сложность этого подхода возрастает. Если вы копируете большое количество деклараций типов для другой библиотеки, то можете закрепить связь открыто, сделав зависимость `@types` явной.

Эта техника также полезна для разрыва зависимостей между тестами модулей и системами вывода. Смотрите пример `getAuthors` в правиле 4.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Используйте структурную типизацию для разрыва несущественных зависимостей.
- ✓ Не принуждайте JavaScript-пользователей зависеть от `@types`, а веб-разработчиков — от NodeJS.

ПРАВИЛО 52. Тестируйте типы с осторожностью

Вы же не опубликуете код, не написав для него тесты (я надеюсь)? То же касается и деклараций типов. Но как же тестировать типы? Тестирование собственных деклараций — сложная задача. Было бы удобно использовать утверждения типов внутри системы типов с помощью инструментов TypeScript. Но в этом подходе присутствует несколько опасностей, поэтому лучше использовать инструменты вроде `dtlint`, которые инспектируют типы извне.

Допустим, вы написали декларацию типов для функции `map`, предоставленной библиотекой утилит (`Lodash` или `Underscore`):

```
declare function map<U, V>(array: U[], fn: (u: U) => V): V[];
```

Как же проверить, что эта декларация обеспечивает верные типы? (По всей видимости, для реализации нужны отдельные тесты.) Можно написать тестовый файл, вызывающий функцию:

```
map(['2017', '2018', '2019'], v => Number(v));
```

Произойдет довольно грубая проверка ошибок: если декларация `map` выведет лишь один параметр, то ошибка будет обнаружена. Но не кажется ли вам, что здесь чего-то не хватает?

Эквивалент этого теста поведения при выполнении может выглядеть так:

```
test('square a number', () => {
  square(1);
  square(2);
});
```

Да, вы проверите, не выбрасывает ли функция `square` ошибку. Но здесь нет проверок возвращаемых значений, значит, реального тестирования поведения не произойдет и неверная реализация `square` пройдет такой тест.

Этот подход распространен для тестирования файлов деклараций типов из-за простоты копирования существующих тестов модулей для библиотеки. Несмотря на то что он имеет свои преимущества, гораздо лучше провести реальную проверку некоторых типов.

Один из способов — присвоить результат переменной с особым типом:

```
const lengths: number[] = map(['john', 'paul'], name => name.length);
```

Получится именно тот вид перегруженной декларации типа, который мы устранили в правиле 19. Но здесь он дает уверенность, что декларация `map` делает хоть что-то ощутимое с типами. В `DefinitelyTyped` есть много деклараций типов, которые используют такой подход для тестирования.

Но как мы увидим, присваивание для тестирования имеет несколько фундаментальных проблем.

Первая заключается в необходимости создавать именованную переменную, которая, скорее всего, останется неиспользованной. Вы получите рутинный код и будете вынуждены отключить некоторые формы линтинга.

Популярный обходной путь — определить хелпер:

```
function assertType<T>(x: T) {}

assertType<number[]>(map(['john', 'paul'], name => name.length));
```

Так будет устранена проблема с неиспользуемой переменной, но некоторые сюрпризы останутся.

Вторая проблема заключается в том, что вместо проверки равенства двух типов мы проверяем их назначаемость. Например:

```
const n = 12;
assertType<number>(n); // ok
```

Если вы просмотрите символ `n`, то увидите, что его тип на самом деле `12` — тип числового литерала. Это подтип `number`, следовательно, он проходит проверку назначаемости.

Пока все хорошо. Но ситуация омрачается, когда вы начинаете проверять типы объектов:

```
const beatles = ['john', 'paul', 'george', 'ringo'];
assertType<{name: string}[]>(
  map(beatles, name => ({
    name,
    inYellowSubmarine: name === 'ringo'
  }))); // ok
```

Вызов `map` возвращает массив объектов `{name: string, inYellowSubmarine: boolean}`. Он может быть назначен для `{name: string}[]`, но не будем ли мы вынуждены также подтвердить `yellow submarine`? В зависимости от контекста решите, проверять типы на равенство или нет.

Если функция возвращает другую функцию, то назначение вас удивит:

```
const add = (a: number, b: number) => a + b;
assertType<(a: number, b: number) => number>(add); // ok

const double = (x: number) => 2 * x;
assertType<(a: number, b: number) => number>(double); // ok?!
```

Почему второе утверждение «успешно»? В TypeScript функция может быть назначена для типа функции, получающего меньше параметров:

```
const g: (x: string) => any = () => 12; // ok
```

То есть вполне нормально вызывать JavaScript-функцию с бóльшим числом параметров, чем предполагает ее декларация. TypeScript моделирует это поведение, не запрещая его, потому что оно широко распространено в обратных вызовах. Например, обратный вызов Lodash функции `map` получает три параметра:

```
map(array, (name, index, array) => { /* ... */ });
```

Несмотря на то что все три являются доступными, зачастую используется только один или два. Запретив такое назначение, TypeScript обнаружил бы огромное число ошибок в коде JavaScript.

Что же можно сделать? Например, разделить тип функции и протестировать его по частям, используя обобщенные типы `Parameters` и `ReturnType`:

```
const double = (x: number) => 2 * x;
let p: Parameters<typeof double> = null!;
assertType<[number, number]>(p);
// ~ Аргумент типа '[number]' не может
//      быть назначен параметру типа
//      [number, number]'.
let r: ReturnType<typeof double> = null!; assertType<number>(r); // ok
```

Но вот еще одна проблема: `map` устанавливает значение `this` для своего обратного вызова. TypeScript моделирует это поведение (правило 49), поэтому декларация типа тоже должна это делать. Но как ее протестировать?

Стиль наших недавних тестов `map` напоминал черный ящик: мы пропускали массив и функцию через `map` и тестировали итоговый тип, но не детали промежуточных шагов. Их можно протестировать, заполнив функцию обратного вызова и напрямую верифицировав типы ее параметров и `this`:

```
const beatles = ['john', 'paul', 'george', 'ringo'];
assertType<number[]>(map(
  beatles,
  function(name, i, array) {
    // ~~~~~ Аргумент типа '(name: any, i: any, array: any) => any'
    //      не может быть назначен параметру типа '(u: string) => any'.
    assertType<string>(name);
    assertType<number>(i);
    assertType<string[]>(array);
    assertType<string[]>(this);
    // ~~~~ 'this' неявно имеет тип 'any'
    return name.length;
  }
));
```

Теперь обнаружилось несколько проблем с декларацией `map`. Обратите внимание, что, используя не стрелочную функцию, мы смогли протестировать тип `this`:

Вот декларация, которая проходит проверки:

```
declare function map<U, V>(
  array: U[],
```

```
fn: (this: U[], u: U, i: number, array: U[]) => V
): V[];
```

Остается всего одна, но самая главная проблема. Есть завершенный файл декларации типа для нашего модуля, который пройдет даже самые требовательные тесты для `map`, но по качеству он хуже бесполезного:

```
declare module 'overbar';
```

Тип `any` присвоился *всему модулю*. Тесты состоятся, но утратится безопасность типов. Что еще хуже, каждый вызов функции в этом модуле будет тихо производить «заразные» типы `any`. Даже используя `noImplicitAny`, вы все равно будете получать `any` через декларации типов.

Чтобы обнаруживать типы `any` изнутри системы типов, требуются инструменты тестирования деклараций, работающие *вне* модуля проверки.

Для деклараций типов, содержащихся в `DefinitelyTyped`, этим инструментом является `dtlint`. Он работает посредством специально отформатированных комментариев. Вот как с его помощью вы можете написать последний тест для функции `map`:

```
const beatles = ['john', 'paul', 'george', 'ringo'];
map(beatles, function(
  name, // $ExpectType string
  i,    // $ExpectType number
  array // $ExpectType string[]
) {
  this // $ExpectType string[]
  return name.length;
}); // $ExpectType number[]
```

Вместо того чтобы проверять назначаемость, `dtlint` инспектирует тип каждого символа и проводит контекстное сравнение. Аналогично декларации типов проверяются в редакторе, но `dtlint` существенно автоматизирует этот процесс.

У этого подхода есть недостатки: `number|string` и `string|number` отличаются в написании, но являются одним типом, несмотря на то что они могут быть назначены друг другу.

Тестирование деклараций типов — это непростое занятие. Их *нужно* тестировать. Но при этом будьте осторожны и рассмотрите использование инструментов вроде `dtlint`, чтобы избежать проблем.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Тестируя типы, остерегайтесь отличия между равенством и назначаемостью, особенно для типов функций.
- ✓ Для функций тестируйте выведенные типы параметров обратных вызовов. Не забудьте протестировать тип `this`, если он является частью вашего API.
- ✓ Будьте осторожны с `any` в тестах, задействующих типы. Рассмотрите применение инструмента вроде `dtslint` для более серьезной проверки, которая будет меньше подвержена ошибкам.

Написание и запуск кода

Эта глава посвящена разбору ряда сложностей, связанных с написанием и запуском самого кода, а не типов.

ПРАВИЛО 53. Используйте возможности ECMAScript, а не TypeScript

С течением времени взаимосвязь между TypeScript и JavaScript претерпела некоторые изменения. Когда Microsoft начинала работу над TypeScript в 2010 году, вокруг JavaScript витала слава проблемного языка, требующего доработки. Считалось обыденной практикой в JavaScript через фреймворки и транспайлеры добавлять недостающую функциональность вроде классов, декораторов и системы модулей. Ранние версии TypeScript тоже включали в себя собственные версии классов, типов-перечислений (`enum`) и модулей.

Спустя некоторое время TC39 — группа разработчиков стандартов, управляющих JavaScript, добавила новые возможности в ядро языка, но они оказались несовместимыми с версиями их аналогов в TypeScript. Это поставило разработчиков TypeScript в затруднительное положение: внедрять новый стандарт или ломать существующий код?

Выбор пал на второй вариант, и был утвержден принцип: TC39 определяет процесс выполнения, в то время как TypeScript занимается обновлением пространства типов.

Но осталось несколько функциональных возможностей, внедренных до принятия этого решения. Очень важно распознавать и понимать их, так как они не согласуются с остальной частью языка. Я рекомендую избегать

их, чтобы связь между TypeScript и JavaScript оставалась максимально прозрачной.

Тип-перечисление (enum)

Многие языки используют типы-перечисления для моделирования типов, способных принимать небольшой набор значений. TypeScript привносит эту возможность в JavaScript:

```
enum Flavor {  
    VANILLA = 0,  
    CHOCOLATE = 1,  
    STRAWBERRY = 2,  
}  
  
let flavor = Flavor.CHOCOLATE; // тип Flavor  
  
Flavor // автоподстановка показывает: VANILLA, CHOCOLATE, STRAWBERRY  
Flavor[0] // значение "VANILLA"
```

Аргумент в пользу enum гласит, что он обеспечивает больше безопасности и прозрачности, чем просто число. Но у него есть свои причуды. Разные варианты enum ведут себя по-разному:

- Типу enum с числовым значением (как в примере с Flavor выше) можно присвоить любое число, поэтому он небезопасен (такая конструкция разработана, чтобы допускать структуры битовых флагов).
- Тип enum со строковым значением безопасен. Он обеспечивает более прозрачные значения во время выполнения. Однако он не является структурно типизированным в отличие от остальных типов в TypeScript (см. ниже).
- Тип `const enum`, в отличие от enum, полностью исчезает при выполнении. Если в примере с Flavor использовать `const enum`, компилятор перепишет `Flavor.Chocolate` как `0`. Это нарушит поведение, ожидаемое от компилятора, и оставит расхождения в поведении между значениями `string` и `number` в enum.
- Тип `const enum` с флагом `preserveConstEnums` сохраняет `const enum` при выполнении как обычные enum.

Особенно удивительно то, что типы enum со строковыми значениями имеют номинальную типизацию, когда любой другой тип в TypeScript для присвоения использует структурную (правило 4):

```
enum Flavor {
  VANILLA = 'vanilla',
  CHOCOLATE = 'chocolate',
  STRAWBERRY = 'strawberry',
}

let flavor = Flavor.CHOCOLATE; // Тип Flavor
  flavor = 'strawberry';
  // ~~~~~ Тип "'strawberry'" не может быть назначен для типа 'Flavor'.
```

Отсюда возникают сложности при публикации библиотек. Предположим, у вас есть функция, которая получает Flavor:

```
function scoop(flavor: Flavor) { /* ... */ }
```

Поскольку при выполнении Flavor — строка, пользователи JavaScript так ее и вызовут:

```
scoop('vanilla'); // ok в JavaScript
```

Однако пользователям TypeScript потребуется импортировать ее и использовать enum:

```
scoop('vanilla');
  // ~~~~~ "'vanilla'" не может быть назначен параметру
  // ~~~~~ типа 'Flavor'.

import {Flavor} from 'ice-cream';
scoop(Flavor.VANILLA); // ok
```

Двойные ситуации и являются поводом избегать enum со строковыми значениями.

TypeScript предлагает их альтернативу, которая менее распространена в других языках, — объединение типов литералов:

```
type Flavor = 'vanilla' | 'chocolate' | 'strawberry';

let flavor: Flavor = 'chocolate'; // ok
  flavor = 'mint chip';
  // ~~~~~ Тип "'mint chip'" не может быть назначен типу 'Flavor'.
```

Оно предоставляет такую же степень безопасности, как enum, но имеет преимущество более прямого переноса в JavaScript и предлагает эффективную автоподстановку в редакторе:

```
function scoop(flavor: Flavor) {
  if (flavor === 'v
    // автоподстановка здесь предлагает 'vanilla'.
}
```

Более подробно об этом подходе читайте в правиле 33.

Свойства параметра

Общепринятая практика при инициализации класса — присваивать свойства параметру конструктора:

```
class Person {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
}
```

TypeScript для этого предлагает более компактный синтаксис:

```
class Person {
  constructor(public name: string) {}
}
```

Свойство параметра равнозначно предыдущему примеру. Используя этот подход, стоит опасаться нескольких сложностей:

- Это одна из немногих конструкций, генерирующих код при компиляции в JavaScript (еще одна — это `enum`). Чаще всего компиляция только стирает типы.
- Так как параметр используется только в сгенерированном коде, в исходном коде он выглядит как неиспользуемый.
- Смещение свойств, независимо от их отношения к параметру, может скрыть структуру классов.

Например:

```
class Person {
  first: string;
  last: string;
  constructor(public name: string) {
    [this.first, this.last] = name.split(' ');
  }
}
```

У этого класса три свойства (`first`, `last`, `name`), но лишь два из них перечислены перед конструктором. Воспринимать код будет еще сложнее, если конструктор получит и другие параметры.

Если класс состоит только из свойств параметра и не содержит методов, вы можете рассмотреть вариант сделать его `interface` и использовать объектные литералы. Вспомните, что два свойства могут быть назначены друг для друга благодаря структурной типизации (правило 4):

```
class Person {  
  constructor(public name: string) {}  
}  
const p: Person = {name: 'Jed Bartlet'}; // ok
```

В отношении свойств параметра мнения расходятся. В то время как я стараюсь их избегать, другие, наоборот, ценят их за экономию времени. Имейте в виду, что они могут слиться с остальной частью кода TypeScript, сделав ее непонятной. Постарайтесь не скрыть класс, правильно используя свойства, относящиеся и не относящиеся к параметру.

Namespace и директивы с тройными слешами

До появления ECMAScript 2015 в JavaScript не было официальной системы модулей. Различные среды компенсировали этот недостаток по-разному: NodeJS использовал `require` и `module.exports` там, где AMD использовал функцию `define` с обратным вызовом.

TypeScript также заполнил этот пробел собственной системой модулей с ключевым словом `module` и директивами с тройными слешами. После того как ECMAScript 2015 ввел официальную систему модулей, TypeScript добавил `namespace` в качестве синонима `module` для избежания путаницы:

```
namespace foo {  
  function bar() {}  
}  
///  
// <ссылочный путь="other.ts"/>  
foo.bar();
```

Вне деклараций типов директивы с тройными слешами и ключевое слово `module` представляют лишь исторический интерес. Используйте модули ECMAScript 2015 `import` и `export` (правило 58).

Декораторы

Декораторы полезны для аннотирования или модификации классов, методов и свойств. Например, вы можете определить аннотацию `logged`, регистрирующую все вызовы метода в классе:

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  @logged
  greet() {
    return "Hello, " + this.greeting;
  }
}

function logged(target: any, name: string, descriptor: PropertyDescriptor)
{
  const fn = target[name];
  descriptor.value = function() {
    console.log(`Calling ${name}`);
    return fn.apply(this, arguments);
  };
}

console.log(new Greeter('Dave').greet());
// Выводит:
// Вызов приветствия:
// Hello, Dave
```

Изначально эта функция была добавлена для поддержания фреймворка Angular и требовала, чтобы в `tsconfig.json` было включено свойство `experimentalDecorators`. ТС39 до сих пор не стандартизировали реализацию декораторов, поэтому любой использующий их код может дать сбой или стать нестандартным в будущем. Пока вы не используете Angular или другой фреймворк, требующий аннотации, и пока декораторы не стандартизированы, не применяйте их в TypeScript.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ В большинстве случаев вы можете конвертировать TypeScript в JavaScript, удалив все типы из кода.

- ✓ `enum`, свойства параметра, директивы с тройными слешами и декораторы являются историческим исключением из этого правила.
- ✓ Для максимально ясного представления роли TypeScript в вашей кодовой базе я рекомендую избегать применения перечисленных в пункте возможностей.

ПРАВИЛО 54. Проводите итерацию по объектам

Этот код отлично запускается, но TypeScript указывает на ошибку. Почему?

```
const obj = {
  one: 'uno',
  two: 'dos',
  three: 'tres',
};
for (const k in obj) {
  const v = obj[k];
  // ~~~~~ Элемент неявно имеет тип 'any'
  //        потому что тип ... не имеет сигнатуры индекса.
}
```

Инспектирование символов `obj` и `k` проясняет ситуацию:

```
const obj = { /* ... */ };
// const obj: {
//   one: string;
//   two: string;
//   three: string;
// }
for (const k in obj) { // const k: string
  // ...
}
```

`k` имеет тип `string`, но вы пытаетесь указать на объект, чей тип имеет только три конкретных ключа: `'one'`, `'two'` и `'three'`. Для строк допустимо большее число значений, чем три, поэтому возникает ошибка.

Объявив более узкий тип для `k`, мы устраним проблему:

```
let k: keyof typeof obj; // Тип "one" | "two" | "three"
for (k in obj) {
  const v = obj[k]; // ok
}
```

Почему тип `k` в первом примере был выведен как `string`, а не `"one" | "two" | "three"`?

Для понимания стоит взглянуть на несколько иной пример, использующий интерфейс и функцию:

```
interface ABC {
  a: string;
  b: string;
  c: number;
}

function foo(abc: ABC) {
  for (const k in abc) { // const k: string
    const v = abc[k];
    // ~~~~~ Элемент неявно имеет тип 'any',
    //       так как 'ABC' не имеет сигнатуры индекса.
  }
}
```

Ошибка та же. Исправить ее можно с помощью аналогичной декларации типа (`let k: keyof ABC`). Но в этом случае TypeScript прав, указав на ошибку, потому что:

```
const x = {a: 'a', b: 'b', c: 2, d: new Date()};
foo(x); // ok
```

Функция `foo` может быть вызвана с любым значением, доступным для назначения `ABC`, но не только со значением, содержащим свойства `"a"`, `"b"`, `"c"`. Вполне возможно, что значение будет иметь и другие свойства (правило 4). Чтобы это допустить, TypeScript присваивает `k` единственный тип, в котором он может быть уверен, а именно `string`.

Применение здесь декларации `keyof` создаст другую проблему:

```
function foo(abc: ABC) {
  let k: keyof ABC;
  for (k in abc) { // let k: "a" | "b" | "c"
    const v = abc[k]; // тип string | number
  }
}
```

Если тип `"a", "b", "c"` слишком узок для `k`, то `string | number` окажется слишком узок для `v`. В примере выше одно из значений — это `Date`, но оно может быть чем угодно. Типы здесь дают ложное чувство уверенности, которое не подтвердится при выполнении.

Если вы захотите перебрать ключи и значения объекта без ошибок типов, `Object.entries` позволит вам перебрать и те и другие одновременно:

```
function foo(abc: ABC) {  
  for (const [k, v] of Object.entries(abc)) {  
    k // Тип string  
    v // Тип any  
  }  
}
```

Несмотря на то что с этими типами будет тяжело работать, они, по крайней мере, честные.

Вам также стоит остерегаться возможного загрязнения прототипа. Даже в случае с объектным литералом, который вы определяете, цикл `for...in` может произвести дополнительные ключи:

```
> Object.prototype.z = 3; // Не делайте этого!>  
> const obj = {x: 1, y: 2};  
> for (const k in obj) { console.log(k); }  
x  
y  
z
```

Даже в неконкурентной среде не стоит добавлять перечисляемые свойства к `Object.prototype`, но `for...in` производит ключи `string` даже для объектных литералов еще и по другой причине.

Если вы хотите перебрать ключи и значения объекта, используйте либо декларацию `keyof` (`let k: keyof T`), либо `Object.entries`. Первый вариант подходит для констант или ситуаций, в которых известно, что объект не будет иметь дополнительных ключей. Второй вариант предпочтительнее, хотя с ключами и типами значений будет сложнее работать.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Используйте `let k: keyof T` и цикл `for...in` для перебора объектов, когда вы знаете, какими именно будут ключи. Имейте в виду, что любые объекты, которые функция получает в качестве параметров, могут иметь дополнительные ключи.
- ✓ Используйте `Object.entries` для перебора ключей и значений объекта.

ПРАВИЛО 55. Иерархия DOM — это важно

Большинство правил книги равно актуальны для запуска TypeScript на браузере, сервере или мобильном телефоне. А это правило касается только работы в браузере.

Используя JavaScript на браузере, вы всегда будете иметь дело с иерархией DOM. Когда вы используете `document.getElementById` для получения элемента или `document.createElement` для его создания, то всегда подразумеваете конкретный вид элемента, даже если вы недостаточно знакомы с таксономией. Вы вызываете методы и используете желаемые свойства, надеясь на лучшее.

В случае с TypeScript иерархия элементов DOM вполне наглядна. `Node` и `EventTarget` помогут вам проводить отладку ошибок типов и решать, в каких случаях уместны утверждения типов. В связи с тем, что многие API основаны на DOM, это также касается работы с фреймворками вроде React или d3.

Предположим, вы хотите отслеживать движения пользовательского курсора, который перемещается через `<div>`. Вы пишете безобидный на первый взгляд код JavaScript:

```
function handleDrag(eDown: Event) {
  const targetEl = eDown.currentTarget;
  targetEl.classList.add('dragging');
  const dragStart = [eDown.clientX, eDown.clientY];
  const handleUp = (eUp: Event) => {
    targetEl.classList.remove('dragging');
    targetEl.removeEventListener('mouseup', handleUp);
    const dragEnd = [eUp.clientX, eUp.clientY];
    console.log('dx, dy = ', [0, 1].map(i => dragEnd[i] - dragStart[i]));
  }
  targetEl.addEventListener('mouseup', handleUp);
}
const div = document.getElementById('surface');
div.addEventListener('mousedown', handleDrag);
```

Но модуль проверки типов TypeScript обозначает не менее 11 ошибок в этих 14 строках:

```
function handleDrag(eDown: Event) {
  const targetEl = eDown.currentTarget;
  targetEl.classList.add('dragging');
```

```

// ~~~~~ Объект, вероятно, 'null'.
// ~~~~~ Свойство 'classList' не существует в типе
// ~~~~~ 'EventTarget'
const dragStart = [
  eDown.clientX, eDown.clientY];
// ~~~~~ 'clientX' не существует в 'Event'
// ~~~~~ 'clientY' не существует в 'Event'
//
const handleUp = (eUp: Event) => {
  targetEl.classList.remove('dragging');
// ~~~~~ Объект, вероятно, 'null'.
// ~~~~~ Свойство 'classList' не существует в типе
// ~~~~~ 'EventTarget'
  targetEl.removeEventListener('mouseup', handleUp);
// ~~~~~ Объект, вероятно, 'null'.
  const dragEnd = [
    eUp.clientX, eUp.clientY];
    // ~~~~~ Свойство 'clientX'
    // ~~~~~ не существует в 'Event'
    // ~~~~~ свойство 'clientY'
    // ~~~~~ не существует в 'Event'
    console.log('dx, dy = ', [0, 1].map(i => dragEnd[i] - dragStart[i]));
  }
  targetEl.addEventListener('mouseup', handleUp);
// ~~~~~ Объект, вероятно, 'null'
}

const div = document.getElementById('surface');
div.addEventListener('mousedown', handleDrag);
// ~~~ Объект, вероятно, 'null'

```

Что пошло не так? Что это за `EventTarget` и почему все может быть `null`?

Изучение иерархии DOM поможет разобраться с ошибками `EventTarget`. Возьмем некий код HTML:

```
<p id="quote">and <i>yet</i> it moves</p>
```

Если вы откроете JavaScript-консоль браузера и получите ссылку на элемент `p`, то увидите, что он является `HTMLParagraphElement`:

```

const p = document.getElementsByTagName('p')[0];
p instanceof HTMLParagraphElement
// true

```

`HTMLParagraphElement` является подтипом `HTMLElement`, который, в свою очередь, является подтипом `Element`, являющегося подтипом `Node` — подтипа `EventTarget`. Вот некоторые примеры типов по ходу построения иерархии (табл. 7.1).

Таблица 7.1. Типы в иерархии DOM

Тип	Примеры
EventTarget	window, XMLHttpRequest
Node	document, Text, Comment
Element	включает HTMLElements, SVGElements
HTMLElement	<i>,
HTMLButtonElement	<button>

EventTarget — это наиболее обобщенный из типов DOM. Все, что можно с ним делать, — это добавлять слушателей событий, удалять их или диспетчеризировать события. Принимая это во внимание, мы лучше поймем ошибки classList:

```
function handleDrag(eDown: Event) {
  const targetEl = eDown.currentTarget;
  targetEl.classList.add('dragging');
// ~~~~~ Объект, вероятно, 'null'.
// ~~~~~ Свойство 'classList' не существует в типе
// 'EventTarget'.
// ...
}
```

Как подразумевает его имя, EventTarget — это свойство currentTarget, принадлежащее Event. Оно вполне может быть null. У TypeScript нет оснований верить, что оно имеет свойство classList. Хотя на деле EventTarget *может* быть HTMLElement, с точки зрения системы типов ничто не запрещает ему быть window или XMLHttpRequest.

Перемещаясь выше по иерархии, мы подходим к Node. Фрагменты и компоненты, такие как в этом коде HTML, имеют тип Node, не являющийся Element:

```
<p>
  And <i>yet</i> it moves
  <!-- quote from Galileo -->
</p>
```

Самый удаленный элемент — это HTMLParagraphElement. Он включает children и childNodes:

```
> p.children
HTMLCollection [i]
> p.childNodes
NodeList(5) [text, i, text, comment, text]
```

`children` возвращает `HTMLCollection` — массивоподобную структуру, содержащую только дочерние элементы (`<i>yet</i>`). `childNodes` возвращает `NodeList` — массивоподобную коллекцию типов `Node`. Она включает не только типы `Element` (`<i>yet</i>`), но и фрагменты текста ("`And`", "`it moves`") и комментарии ("`quote from Galileo`").

Какова же разница между `Element` и `HTMLElement`? Существует не `HTMLElement`, включающий всю иерархию SVG-тегов. Он называется `SVGElement` и является еще одним типом `Element`. То есть типами тегов `<html>` или `<svg>` являются `HTMLElement` и `SVGElement`.

Иногда эти специализированные классы имеют собственные свойства. `HTMLElement` имеет свойство `src`, а `HTMLInputElement` — свойство `value`. Если вы хотите читать одно из этих свойств из значения, то его тип должен быть достаточно специфичным, чтобы иметь это свойство.

В TypeScript декларации для DOM свободно используют литеральные типы, чтобы предоставить вам как можно более специфичные типы. Например:

```
document.getElementsByTagName('p')[0]; // HTMLParagraphElement
document.createElement('button'); // HTMLButtonElement
document.querySelector('div'); // HTMLDivElement
```

Однако это не всегда возможно, особенно с `document.getElementById`:

```
document.getElementById('my-div'); // HTMLElement
```

Несмотря на то что утверждения типов нежелательны (правило 9), в этой ситуации вы действительно знаете больше, чем TypeScript, поэтому можете их использовать. Не произойдет ничего плохого, пока вы знаете, что `#my-div` является `div`:

```
document.getElementById('my-div') as HTMLDivElement;
```

При включенной опции `strictNullChecks` вам нужно учесть случай, когда `document.getElementById` возвращает `null`. В зависимости от того, насколько это вероятно, вы можете добавить либо инструкцию `if`, либо утверждение (!):

```
const div = document.getElementById('my-div')!;
```

Эти типы не являются характерными для TypeScript. Они сгенерированы из официальной спецификации DOM, что является примером рекомендации из правила 35: когда это возможно, генерировать типы из спецификаций.

Уже многое сказано про иерархию DOM, а что же насчет ошибок `clientX` и `clientY`?

```
function handleDrag(eDown: Event) {
  // ...
  const dragStart = [
    eDown.clientX, eDown.clientY];
    // ~~~~~ 'clientX' не существует в 'Event'
    // ~~~~~ 'clientY' не существует в 'Event'
  // ...
}
```

В дополнение к иерархии типов `Node` и `Element` существует еще иерархия типов `Event`. Документация Mozilla на данный момент содержит не менее 52 типов `Event`.

Основной `Event` является самым обобщенным типом событий. Более специфичные варианты включают:

- `UIEvent` — любой вид события интерфейса пользователя.
- `MouseEvent` — событие, запущенное мышью (в частности, кликом).
- `TouchEvent` — касание на мобильных устройствах.
- `WheelEvent` — вращение колеса прокрутки.
- `KeyboardEvent` — нажатие кнопки.

Проблема в `handleDrag` кроется в том, что события объявлены как `Event`, в то время как `clientX` и `clientY` существуют только в более конкретном типе `MouseEvent`.

Как же можно исправить пример из начала этого правила? TypeScript-декларации для DOM широко используют контекст (правило 26). Поэтому встраивание обработчика `mousedown` даст TypeScript больше информации и устранил многие ошибки. Вы также можете объявить тип параметра как `MouseEvent` вместо `Event`. Вот вариант с применением обеих техник:

```
function addDragHandler(el: HTMLElement) {
  el.addEventListener('mousedown', eDown => {
    const dragStart = [eDown.clientX, eDown.clientY];
```

```

const handleUp = (eUp: MouseEvent) => {
  el.classList.remove('dragging');
  el.removeEventListener('mouseup', handleUp);
  const dragEnd = [eUp.clientX, eUp.clientY];
  console.log('dx, dy = ', [0, 1].map(i => dragEnd[i] - dragStart[i]));
}
el.addEventListener('mouseup', handleUp);
});
}
const div = document.getElementById('surface');
if (div) {
  addDragHandler(div);
}

```

Инструкция `if` в конце обрабатывает вероятность отсутствия элемента `#surface`. Если вы знаете, что этот элемент существует, то можете вместо нее использовать утверждение `(div!)`. `addDragHandler` требует ненулевое значение, поэтому `null` стоит сместить на периферию кода (правило 31).

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ DOM имеет такую иерархию типов, которую в случае с JavaScript вы можете чаще всего игнорировать. Однако эти типы становятся более существенными при работе в TypeScript. Их понимание поможет написать TypeScript-код для браузера.
- ✓ Постарайтесь понять разницу между `Node`, `Element`, `HTMLElement` и `EventTarget`, а также между `Event` и `MouseEvent`.
- ✓ Либо используйте в коде достаточно конкретные типы, либо предоставьте TypeScript дополнительный контекст для их вывода.

ПРАВИЛО 56. Не полагайтесь на `private` при скрытии информации

В JavaScript всегда не хватало возможности делать свойства класса приватными. Обычно в качестве альтернативы используются префиксы с нижним подчеркиванием для полей, которые не являются частью публичного API:

```

class Foo {
  _private = 'secret123';
}

```

Но для пользователей это просто затрудняет доступ к приватным данным, и при желании такой прием можно обойти:

```
const f = new Foo();
f._private; // 'secret123'
```

В TypeScript имеются дополнительные модификаторы полей `public`, `protected` и `private`, которые скрывают информацию лучше:

```
class Diary {
  private secret = 'cheated on my English test';
}

const diary = new Diary();
diary.secret
// ~~~~~ Свойство 'secret' является приватным
//       и доступно только внутри класса 'Diary'.
```

Но `private` является особенностью системы типов и удаляется при выполнении (правило 3). Вот как выглядит этот фрагмент после компиляции в JavaScript (при `target=ES2017`):

```
class Diary {
  constructor() {
    this.secret = 'cheated on my English test';
  }
}
const diary = new Diary();
diary.secret;
```

Указатель `private` удален, и ваш секрет открылся. Условное обозначение `_private` из JavaScript тоже представляет лишь помеху для получения доступа к приватным данным. Используя утверждение типа, вы даже можете получить доступ к приватному свойству внутри TypeScript:

```
class Diary {
  private secret = 'cheated on my English test';
}

const diary = new Diary();
(diary as any).secret // ok
```

Поэтому *не стоит полагаться на `private` для скрытия информации*.

Есть ли более надежная альтернатива? JavaScript предлагает механизмы замыкания, которые можно создавать в конструкторе:

```

declare function hash(text: string): number;

class PasswordChecker {
  checkPassword: (password: string) => boolean;
  constructor(passwordHash: number) {
    this.checkPassword = (password: string) => {
      return hash(password) === passwordHash;
    }
  }
}

const checker = new PasswordChecker(hash('s3cret'));
checker.checkPassword('s3cret'); // возвращает true

```

В JavaScript нет способа получить доступ к переменной `passwordHash` вне конструктора `PasswordChecker`. Но и здесь есть несколько негативных моментов: каждый метод, который `passwordHash` использует, должен быть определен в `PasswordChecker`. Поэтому требуются копии методов для каждого экземпляра класса, которые повышают затраты памяти. Они мешают другим экземплярам того же класса получать приватные данные. Механизмы замыкания могут быть неудобны, но они однозначно эффективны.

Более современный способ решить эту задачу — использовать приватные поля, являющиеся стандартом JavaScript, который будет утвержден к моменту выхода этой книги. Чтобы сделать поле приватным и для модуля проверки, и для среды выполнения, добавьте к нему префикс `#`:

```

class PasswordChecker {
  #passwordHash: string;

  constructor(passwordHash: number) {
    this.#passwordHash = passwordHash;
  }

  checkPassword(password: string) {
    return hash(password) === this.#passwordHash;
  }
}

const checker = new PasswordChecker(hash('s3cret'));
checker.checkPassword('secret'); // возвращает false
checker.checkPassword('s3cret'); // возвращает true

```

Свойство `#passwordHash` недоступно вне класса. Его отличие от техники замыкания в том, что оно все равно доступно из методов класса и других экземпляров того же класса. Для целевых объектов ECMAScript, не име-

ющих встроенной поддержки приватных полей, используется запасной вариант с применением `WeakMaps`, который оставляет данные приватными.

В заключение добавлю, что если вас интересует *безопасность*, а не простая инкапсуляция, то есть и другие требующие изучения вопросы. Например, модификации для встроенных прототипов и функций.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Модификатор доступа оказывается эффективным лишь для системы типов. Он не работает в среде выполнения и может быть проигнорирован утверждением. Не надейтесь, что он сохранит данные скрытыми.
- ✓ Для более надежной защиты информации используйте механизмы замыкания или приватные поля с префиксом `#`.

ПРАВИЛО 57. Используйте карты кода для отладки

Когда вы запускаете исходный код TypeScript, фактически он представляет собой скомпилированный исходный код JavaScript, полученный посредством минификатора, компилятора или препроцессора. Хорошо, когда код достаточно прозрачен и выполняется на TypeScript без необходимости править его версию на JavaScript.

Но иногда приходится проводить отладку кода. По умолчанию отладчики работают с исполняемым кодом и не учитывают пройденный им процесс компиляции. Поскольку JavaScript является популярным целевым языком, поставщики браузеров разработали для его отладки карты кода. Они отображают позиции и символы в сгенерированном файле согласно соответствующим позициям и символам оригинального кода. При этом карты поддерживает большинство браузеров и IDE. Если вы еще не используете их, то многое теряете!

Предположим, вы создали небольшой скрипт для добавления на HTML-страницу кнопки, которая увеличивается при нажатии:

```
function addCounter(el: HTMLElement) {  
  let clickCount = 0;  
  const button = document.createElement('button');  
  button.textContent = 'Click me';  
  button.addEventListener('click', () => {
```

```

    clickCount++;
    button.textContent = `Click me (${clickCount})`;
  });
  el.appendChild(button);
}

addCounter(document.body);

```

Если вы загрузите его в браузер и запустите отладчик, то увидите сгенерированный JavaScript. Он не сильно отличается от своего исходного варианта, поэтому отладка не составит труда (рис. 7.1).

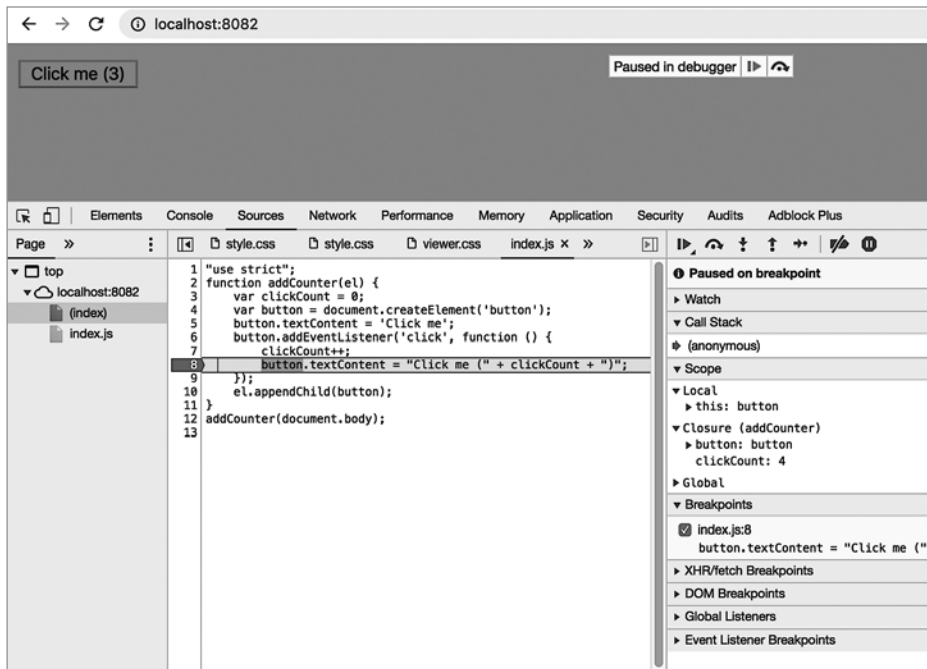


Рис. 7.1. Отладка сгенерированного JavaScript с помощью инструментов разработчика Chrome. Здесь сгенерированный JavaScript очень похож на TypeScript

Давайте сделаем страницу повеселее, запросив интересные факты о каждом числе с ресурса numbersapi.com:

```

function addCounter(el: HTMLElement) {
  let clickCount = 0;
  const triviaEl = document.createElement('p');

```

```

const button = document.createElement('button');
button.textContent = 'Click me';
button.addEventListener('click', async () => {
  clickCount++;
  const response = await fetch(`http://numbersapi.com/${clickCount}`);
  const trivia = await response.text();
  triviaEl.textContent = trivia;
  button.textContent = `Click me (${clickCount})`;
});
el.appendChild(triviaEl); el.appendChild(button);
}

```

Если вы снова откроете отладчик, то увидите, что сгенерированный код усложнился (рис. 7.2).

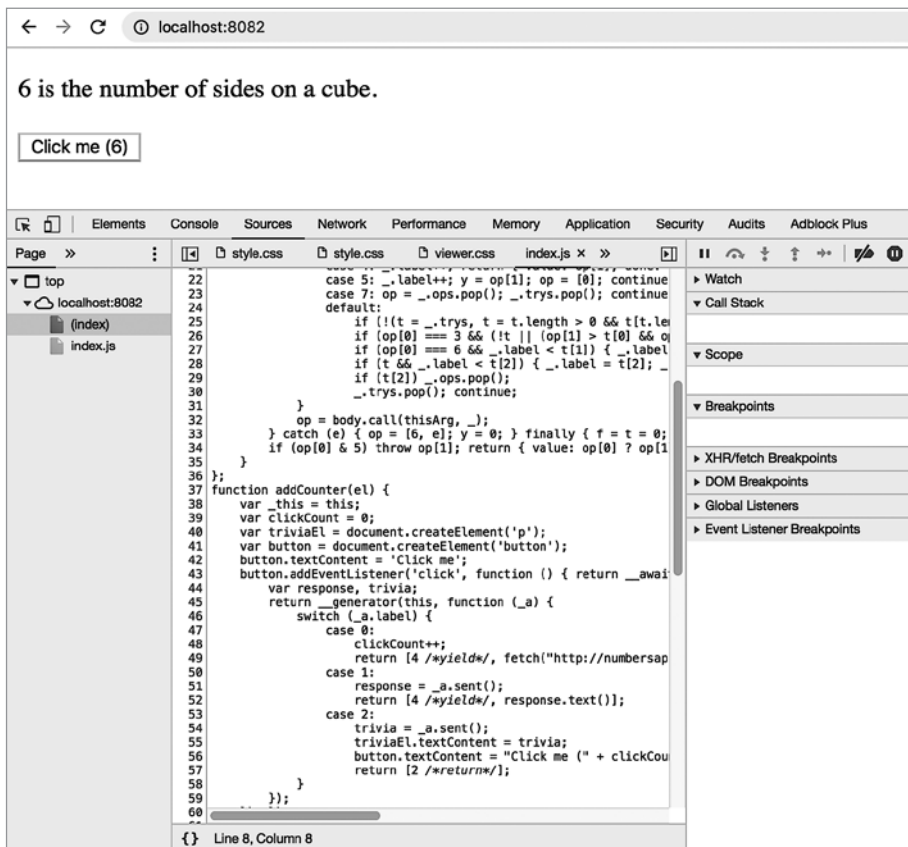


Рис. 7.2. Здесь компилятор TypeScript сгенерировал JavaScript, который не очень похож на исходный код TypeScript. Это сделает отладку сложнее

Для поддержки `async` и `await` в старых браузерах TypeScript переписал обработчик событий как конечную машину. Поведение кода будет тем же, но он не будет близок своему оригиналу.

Вот здесь-то и помогает карта кода. Чтобы TypeScript создал ее, вам нужно активировать опцию `sourceMap` в `tsconfig.json`:

```
{
  "compilerOptions": {
    "sourceMap": true
  }
}
```

Теперь при запуске `tsc` сгенерируются два выходных файла для каждого файла `.ts`, `.js` и `.js map`. Последний будет являться картой кода.

При ее формировании в отладчике браузера появится новый файл `index.ts`. Вы сможете обозначить в нем точки прерывания и инспектировать переменные, как и планировали (рис. 7.3).

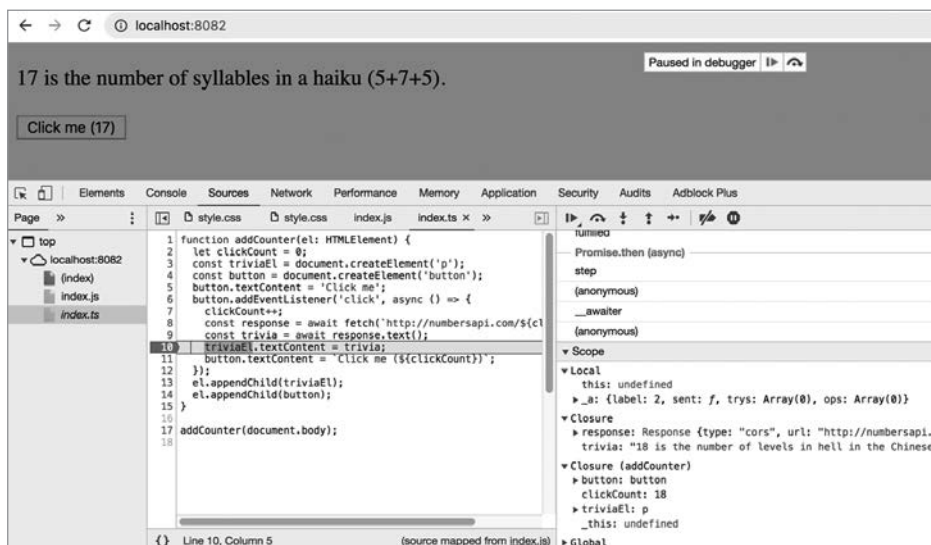


Рис. 7.3. При наличии исходной карты вы можете работать с исходным кодом TypeScript в своем отладчике, а не со сгенерированным JavaScript

Имейте в виду, что `index.ts`, напечатанный курсивом, появляется в списке файлов слева. Это говорит о том, что он не является «реальным» файлом —

он был добавлен не веб-страницей, а через карту кода. В зависимости от настроек `index.js.map` будет содержать либо ссылку на `index.ts` (браузер загрузит его по сети), либо встроенную копию (запросов не потребуется).

Однако есть несколько опасных моментов:

- Бандлер или минификатор, которые вы можете использовать в TypeScript, способны генерировать собственные карты кода. Для получения лучших результатов отладки добейтесь отображения оригинального кода, а не его конечного JavaScript-варианта. Если в вашем бандлере есть встроенная поддержка TypeScript, то все должно работать как надо. Если же нет, то, вероятно, вам понадобится отыскать нужные флаги, чтобы заставить бандлер читать входные данные карт кода.
- Остерегайтесь размещения карт кода в продакшене. Браузер не будет загружать их, если не открыт отладчик, и пользователи не увидят разницы. Но если карта кода содержит встроенную копию оригинального кода, то в ней может присутствовать содержимое, которое вы не собирались разглашать. Действительно ли пользователям по всему миру стоит видеть ваши придиричивые комментарии или URL-адреса внутреннего баг-трекера?

С помощью карт кода также можно осуществить отладку программы NodeJS. Обычно это делается посредством редактора или через подключение к вашему процессу Node из отладчика браузера. Подробнее об этом написано в документации Node.

Модуль проверки способен отловить множество ошибок, прежде чем вы запустите код, но не сравнится с хорошим отладчиком. Используйте карты кода для получения лучших результатов отладки в TypeScript.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Не осуществляйте отладку сгенерированного JavaScript-кода. Используйте карты кода для отладки TypeScript-кода в среде выполнения.
- ✓ Убедитесь, что ваши карты кода полностью отображают запускаемый вами исходный код.
- ✓ При определенных настройках карты кода могут содержать встроенную копию оригинального кода. Не опубликуйте их случайно.

Перенос данных в TypeScript

Вы уже слышали о том, что TypeScript хорош. Вы также наверняка знаете из горького опыта, что поддержание 15-летней библиотеки JavaScript размером в 10 000 строк дается трудно. Однако можно перенести ее в TypeScript.

В этой главе представлены некоторые советы по миграции проектов из JavaScript в TypeScript без утраты их работоспособности и вложенных усилий.

Быстрая миграция доступна только для небольших баз кода, в то время как более крупные переносятся поэтапно. В правиле 60 этот процесс описан подробно. При длительных миграциях важно отслеживать прогресс и убеждаться, что вы не возвращаетесь к пройденному. Так вы сможете почувствовать инерцию и необратимость изменений (правило 61).

Перенос крупных проектов не обязательно будет легким, но принесет огромный рабочий потенциал. Исследование, проведенное в 2017 году, выявило, что 15 % багов, исправленных в JavaScript-проектах на GitHub, могли быть предотвращены с помощью TypeScript¹. По результатам шестимесячного ретроспективного исследования проектов в Airbnb количество багов, подвластных TypeScript, составило 38 %². Используйте эти данные, предлагая TypeScript в своей организации, а в правиле 59 мы рассмотрим эксперименты с TypeScript до переноса данных.

Поскольку эта глава существенно затрагивает JavaScript, многие образцы кода либо являются чистым JavaScript (не предполагают прохождения модуля проверки типов), либо проверяются с урезанными настройками (например, с выключенной опцией `noImplicitAny`).

¹ Z. Gao, C. Bird and E. T. Barr, «Типизировать или не типизировать: подсчет обнаруживаемых багов в JavaScript», ICSE 2017. <http://earlbarr.com/publications/typestudy.pdf>.

² Brie Bunge, «Внедрение TypeScript с масштабированием», JSConf Hawaii 2019. <https://youtu.be/P-J9Eg7hJwE>.

ПРАВИЛО 58. Пишите современный JavaScript

В дополнение к проверке кода на безопасность типов TypeScript компилирует его в любую версию JavaScript, вплоть до самой давней ES3 1999 года. Поскольку TypeScript — это надмножество последней версии JavaScript, вы можете использовать `tsc` в качестве транспайлера. Это позволит из современной версии JavaScript получить старую, но более широко поддерживаемую.

С другой стороны, это означает, что когда вы конвертируете базу JavaScript-кода в TypeScript, не возникает проблемы с внедрением всех возможностей последней версии JavaScript. Так как TypeScript разработан для взаимодействия с последними версиями JavaScript, модернизирование вашего JS-кода послужит отличным первым шагом к освоению TypeScript.

Занимаясь написанием современного JavaScript, вы совершенствуете свое понимание TypeScript.

Это правило является небольшой экскурсией по особенностям современного JavaScript, которые были введены в ES2015 (ES6) и после. Содержащийся здесь материал более развернуто освещен в других книгах и на онлайн-ресурсах. Поэтому если какие-либо упомянутые темы покажутся вам незнакомыми, можете самостоятельно их изучить. TypeScript становится особенно полезным, когда вы изучаете новые возможности языка вроде `async` и `await` — он понимает эти возможности лучше вас и может содействовать в правильном их применении.

Наиболее важные особенности TypeScript — это модули ECMAScript и классы ES2015.

Используйте модули ECMAScript

До появления версии ECMAScript 2015 не существовало стандартного способа деления кода на модули. Для этого использовалось множество решений, начиная с тегов `<script>`, ручной конкатенации и сборочных файлов до инструкций `require` в стиле `node.js` или обратных вызовов `define` в стиле AMD. В TypeScript даже имелась собственная система модулей (правило 53).

Сегодня существует единый стандарт деления кода на модули ECMAScript `import` и `export`. Если ваша база JavaScript-кода все еще состоит из одного файла и вы используете конкатенацию или одну из других систем модулей, то настало время переключиться на модули ES. Для этого может потребо-

ваться установить инструмент **webpack** или **ts-node**. TypeScript лучше всего работает с модулями ES, и их внедрение упростит миграцию, в частности, благодаря возможности переносить по одному модулю за раз (правило 61).

Детали будут отличаться в зависимости от настроек, но если вы используете CommonJS:

```
// CommonJS
// a.js
const b = require('./b');
console.log(b.name);

// b.js
const name = 'Module B';
module.exports = {name};
```

эквивалент с ES-модулем будет выглядеть так:

```
// ECMAScript module
// a.ts
import * as b from './b';
console.log(b.name);

// b.ts
export const name = 'Module B';
```

Используйте классы вместо прототипов

В JavaScript реализована гибкая объектная модель, основанная на прототипах. Но разработчики часто ею пренебрегали в угоду более жесткой модели, основанной на классах, которая была закреплена в ES2015 вместе с ключевым словом **class**.

Если в вашем коде прототипы применены в простом виде, то переключитесь на классы. Вместо:

```
function Person(first, last) {
  this.first = first;
  this.last = last;
}

Person.prototype.getName = function() {
  return this.first + ' ' + this.last;
}

const marie = new Person('Marie', 'Curie');
const personName = marie.getName();
```

напишите:

```
class Person {
  first: string;
  last: string;

  constructor(first: string, last: string) {
    this.first = first;
    this.last = last;
  }

  getName() {
    return this.first + ' ' + this.last;
  }
}

const marie = new Person('Marie', 'Curie');
const personName = marie.getName();
```

У TypeScript есть сложности с версией прототипа `Person`, но при этом ему достаточно минимума аннотаций для понимания версии, основанной на классах. Если вы пока не знакомы с синтаксисом, TypeScript поможет правильно его применить.

Для кода, использующего старый вариант классов, языковые службы предлагают быструю «Функцию конвертации в класс ES2015», которая ускоряет процесс (рис. 8.1).

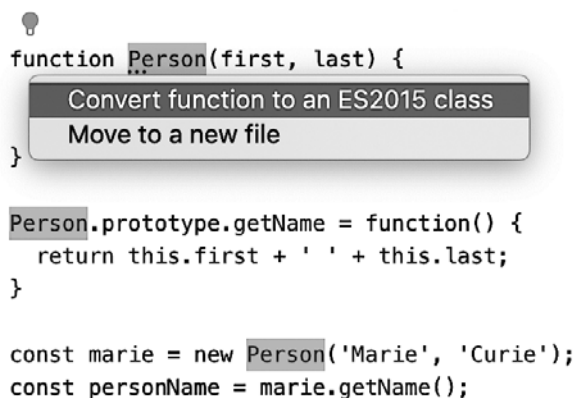


Рис. 8.1. Служба языка TypeScript предлагает быстрое решение для преобразования классов более старых стилей в классы ES2015

Используйте `let` и `const` вместо `var`

Ключевое слово `var` имеет замысловатые правила видимости. Если вам интересно больше узнать о них, то прочитайте «Эффективный JavaScript». Однако лучше всего избегать этого слова и использовать `let` и `const`. Их диапазон видимости ограничивается одним блоком, а их работа интуитивно понятна.

TypeScript и здесь вам поможет. Если замена `var` на `let` выдает ошибку, то вы наверняка делаете то, чего делать не стоит.

Вложенные инструкции функций также имеют правила видимости, подобные `var`:

```
function foo() {  
  bar();  
  function bar() {  
    console.log('hello');  
  }  
}
```

Когда вы вызываете `foo()`, логируется `hello`, так как определение `bar` оказывается наверху в `foo`. Это удивительно! Здесь предпочтительнее использовать выражение функции `const bar = () => { ...}`.

Используйте `for...of` или методы массива вместо `for(;;)`

В классическом JavaScript для итерации по массиву вы использовали стиль C:

```
for (var i = 0; i < array.length; i++) {  
  const el = array[i];  
  // ...  
}
```

В современном JavaScript вместо этого вы можете использовать `for...of`:

```
for (const el of array) {  
  // ...  
}
```

Этот вариант менее подвержен опечаткам и не вводит переменную индекса. Если же вам нужна такая переменная, то можете использовать `forEach`:

```
array.forEach((el, i) => {  
  // ...  
});
```

Избегайте использования конструкции `for...in` для итерации по массивам, так как это приведет к нежелательным сюрпризам (правило 16).

Используйте стрелочные функции вместо функций-выражений

Так как правила видимости ключевого слова `this` отличны от других переменных, то оно является одним из самых путающих аспектов в языке JavaScript:

```
class Foo {  
  method() {  
    console.log(this);  
    [1, 2].forEach(function(i) {  
      console.log(this);  
    });  
  }  
}  
const f = new Foo();  
f.method();  
// печатает Foo, undefined, undefined в строгом режиме  
// печатает Foo, window, window (!) в нестрогом режиме
```

Вам нужно, чтобы `this` ссылалась на соответствующий экземпляр любого класса, в котором вы находитесь. Стрелочные функции в этом помогут, удерживая значение `this` вне их области:

```
class Foo {  
  method() {  
    console.log(this);  
    [1, 2].forEach(i => {  
      console.log(this);  
    });  
  }  
}  
const f = new Foo();  
f.method();  
// всегда печатает Foo, Foo, Foo
```

Помимо использования более простой семантики, стрелочные функции более сжаты, поэтому их стоит применять везде, где это возможно. Более подробно о привязке `this` читайте в правиле 49. При включенной опции `noImplicitAny` (или `strict`) TypeScript поможет сделать привязку `this` правильно.

Используйте компактные объектные литералы и деструктурирующее назначение

Вместо:

```
const x = 1, y = 2, z = 3;
const pt = {
  x: x,
  y: y,
  z: z
};
```

вы можете просто написать:

```
const x = 1, y = 2, z = 3;
const pt = { x, y, z };
```

Кроме краткости, это обеспечит согласованное именование переменных и свойств, которое точно оценят будущие читатели кода (правило 36).

Для возвращения объектного литерала из стрелочной функции заключите его в скобки:

```
['A', 'B', 'C'].map((char, idx) => ({char, idx}));
// [ { char: 'A', idx: 0 }, { char: 'B', idx: 1 }, { char: 'C', idx: 2 } ]
```

Для свойств, чьи значения являются функциями, существует краткий вариант записи:

```
const obj = {
  onClickLong: function(e) {
    // ...
  },
  onClickCompact(e) {
    // ...
  }
};
```

Обратное преобразование компактных объектных литералов — это деструктуризация объекта. Вместо:

```
const props = obj.props;
const a = props.a;
const b = props.b;
```

можно написать так:

```
const {props} = obj;  
const {a, b} = props;
```

или так:

```
const {props: {a, b}} = obj;
```

В последнем примере переменными становятся только `a` и `b`, но не `props`.

При деструктуризации вы можете назначить значение по умолчанию. Вместо:

```
let {a} = obj.props;  
if (a === undefined) a = 'default';
```

напишите так:

```
const {a = 'default'} = obj.props;
```

Вы также можете деструктурировать массивы, что особенно полезно в случае с кортежными типами:

```
const point = [1, 2, 3];  
const [x, y, z] = point;  
const [, a, b] = point; // игнорирует первый
```

Деструктуризация может быть применена и в параметрах функций:

```
const points = [  
  [1, 2, 3],  
  [4, 5, 6],  
];  
points.forEach(([x, y, z]) => console.log(x + y + z));  
// логирует 6, 15
```

Как и в случае с компактным синтаксисом объектных литералов, деструктуризация является краткой и содействует формированию согласованных имен переменных. Пользуйтесь этим!

Используйте параметры функций по умолчанию

В JavaScript все параметры функции являются опциональными:

```
function log2(a, b) {  
  console.log(a, b);  
}  
log2();
```

Вывод:

```
undefined undefined
```

Это зачастую используется для применения в параметрах значений по умолчанию:

```
function parseNum(str, base) {  
    base = base || 10;  
    return parseInt(str, base);  
}
```

В современном JavaScript вы можете назначать значения по умолчанию непосредственно в списке параметров:

```
function parseNum(str, base=10) {  
    return parseInt(str, base);  
}
```

Помимо повышения лаконичности это проясняет, что `base` является опциональным параметром. При миграции в TypeScript параметры по умолчанию имеют еще и другое преимущество — они помогают модулю проверки типов выводить тип параметра без необходимости его аннотирования (правило 19).

Используйте `async` и `await` вместо чистых промисов или обратных вызовов

Правило 25 поясняет, почему `async` и `await` являются предпочтительными, но сама их суть в том, что они упрощают код, предотвращают появление багов и помогают типам пройти весь ваш асинхронный код.

Вместо любого из этих вариантов:

```
function getJSON(url: string) {  
    return fetch(url).then(response => response.json());  
}  
function getJSONCallback(url: string, cb: (result: unknown) => void) {  
    // ...  
}
```

напишите так:

```
async function getJSON(url: string) {  
    const response = await fetch(url);  
    return response.json();  
}
```

Не добавляйте 'use strict' в TypeScript

В ES2015 был введен строгий режим, чтобы определить некоторые подозрительные паттерны как более наглядные ошибки. Включить этот режим можно, внося 'use strict' в код:

```
'use strict';
function foo() {
  x = 10; // выбрасывает в строгом режиме, определяет глобальную
          // переменную в нестрогом.
}
```

Если вы никогда не использовали строгий режим в кодовой базе JavaScript, то попробуйте. Ошибки, которые при этом обнаружатся, вероятно, будут также обнаружены и компилятором TypeScript.

Но при переносе нет особого смысла оставлять 'use strict' в исходном коде, потому что проверки работоспособности, проводимые TypeScript, строже тех, которые предлагает этот режим.

Тем не менее есть некоторая ценность 'use strict' для кода JavaScript, который генерирует tsc. Если вы активируете опцию `alwaysStrict` или `strict`, TypeScript произведет парсинг кода в строгом режиме и поместит 'use strict' в его итоговый JavaScript-вариант.

Коротко говоря, вместо 'use strict' в TypeScript лучше использовать `alwaysStrict`.

Это лишь немногие из огромного числа возможностей JavaScript, которые доступны для использования в TypeScript. TC39 очень активен, и новые возможности добавляются ежегодно. Команда TypeScript на данный момент нацелена на внедрение любых возможностей, которые достигают третьей стадии процесса стандартизации (всего их четыре). Проверьте наличие обновлений в репозитории TC39 на GitHub¹. На момент написания книги предложения пайплайна и декораторов имеют серьезный потенциал влияния на TypeScript.

¹ <https://github.com/tc39/proposals>.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ TypeScript позволяет писать современный JavaScript независимо от того, какая у вас среда выполнения. Используйте преимущества тех дополнительных возможностей, которые при этом появляются. Помимо улучшения самой базы кода это поможет TypeScript лучше понимать код.
- ✓ Используйте TypeScript для изучения языковых особенностей вроде классов, деструктуризации и ключевых слов `async` и `await`.
- ✓ Не озадачивайтесь `'use strict'`, потому что TypeScript строже.
- ✓ Заглядывайте в репозиторий TC39 на GitHub, а также проверяйте описание последних изменений TypeScript, чтобы знать все новейшие возможности языка.

ПРАВИЛО 59. Используйте `// @ts-check` и JSDoc для экспериментов в TypeScript

До начала конвертации исходных JavaScript файлов в TypeScript (правило 60) вы можете поэкспериментировать с проверкой типов, чтобы предварительно увидеть все потенциальные проблемы. Для этого можно использовать TypeScript-директиву `// @ts-check`. Она дает задачу модулю проверки типов проанализировать отдельный файл и сообщить обо всех обнаруженных в нем проблемах. Вы можете воспринимать ее как упрощенный вариант проверки типов, который даже проще, чем проверка с выключенной опцией `noImplicitAny` (правило 2).

Вот как это работает:

```
// @ts-check
const person = {first: 'Grace', last: 'Hopper'};
2 * person.first
// ~~~~~ Правая часть арифметической операции должна иметь тип
// 'any', 'number', 'bigint' или тип enum
```

TypeScript выводит тип `person.first` как `string`, следовательно, `2 * person.first` имеет ошибку типа, аннотация не требуется.

В то время как эта проверка может вскрывать очевидные ошибки или функции, вызываемые со слишком большим числом аргументов, директива `// @ts-check` стремится обнаружить несколько специфичных видов ошибок.

Необъявленные глобальные переменные

Если ими являются определяемые вами символы, тогда объявите их с `let` или `const`. Если же они являются символами, определяемыми где-либо еще (например, в теге `<script>` HTML-файла), создайте декларации типов для их описания.

Например, для такого JavaScript-кода:

```
// @ts-check
console.log(user.firstName);
// ~~~~ Невозможно найти имя 'user'.
```

вы можете создать файл с названием `types.d.ts`:

```
interface UserData {
  firstName: string;
  lastName: string;
}
declare let user: UserData;
```

Одно только создание этого файла может устранить проблему. Еще вам может понадобиться явно импортировать его ссылкой с тремя слешами:

```
// @ts-check
/// <путь ссылки="./types.d.ts" />
console.log(user.firstName); // ok
```

Файл `types.d.ts` станет основой деклараций типов вашего проекта.

Неизвестные библиотеки

Если вы применяете стороннюю библиотеку, TypeScript должен об этом знать. Например, вы используете jQuery, чтобы установить размер HTML-элемента. С директивой `// @ts-check` TypeScript обозначит ошибку:

```
// @ts-check
$('#graph').style({'width': '100px', 'height': '100px'});
// ~ Невозможно найти имя '$'.
```

Решением будет установить декларации типов для jQuery:

```
$ npm install --save-dev @types/jquery
```

Теперь ошибка относится конкретно к jQuery:

```
// @ts-check
$('#graph').style({'width': '100px', 'height': '100px'});
// ~~~~~ Свойство 'style' не существует в типе
//       'jQuery<HTMLElement>'
```

На деле должно быть `.css`, а не `.style`.

// @ts-check позволяет воспользоваться преимуществами деклараций TypeScript для популярных библиотек без необходимости переноса. Это одно из главных преимуществ этой директивы.

Проблемы с DOM

Предполагая, что вы пишете код, выполняющийся в браузере, TypeScript, скорее всего, обозначит проблемы, связанные с обработкой элементов DOM. Например:

```
// @ts-check
const ageEl = document.getElementById('age');
ageEl.value = '12';
// ~~~~~ Свойство 'value' не существует в типе 'HTMLElement'.
```

Проблема в том, что свойство `value` имеет только тип `HTMLInputElement`, но `document.getElementById` возвращает более общий тип `HTMLElement` (правило 55). Если вам известно, что элемент `#age` на самом деле является элементом `input`, тогда используйте утверждение (правило 9). Но это все еще файл JS, значит, вы не можете написать `as HTMLInputElement`. Утвердите тип с помощью JSDoc:

```
// @ts-check
const ageEl = /** @type {HTMLInputElement} */(document.getElementById('age'));
ageEl.value = '12'; // ok
```

Если вы наведете курсор на `ageEl` в редакторе, то увидите, что TypeScript теперь считает его `HTMLInputElement`. Будьте осторожны, печатая аннотацию типа JSDoc `@type` — после комментария необходимы скобки.

Это ведет к еще одному виду ошибки, которая вскрывается // @ts-check:

Неверный JSDoc-комментарий

Если в вашем проекте уже есть JSDoc-комментарии, TypeScript начнет их проверку, когда вы включите директиву `// @ts-check`. Если вы ранее использовали систему вроде Closure Compiler, в которой эти комментарии служили для безопасности типов, тогда существенных сложностей возникнуть не должно. Однако вы можете столкнуться с некоторыми сюрпризами, если эти комментарии носили рекомендательный характер:

```
// @ts-check
/**
 * Получает размер элемента в пикселах.
 * @param {Node} Элемент el
 * @return {{w: number, h: number}} Size (Размер)
 */
function getSize(el) {
  const bounds = el.getBoundingClientRect();
  // ~~~~~ Свойство 'getBoundingClientRect'
  // ~~~~~ не существует в типе 'Node'.
  return {width: bounds.width, height: bounds.height};
  // ~~~~~ Тип '{ width: any; height: any; }' не может
  // ~~~~~ быть присвоен типу '{ w: number; h: number; }'
}
```

Первая проблема заключается в неверном понимании DOM: `getBoundingClientRect()` определен в `Element`, а не в `Node`. Следовательно, тег `@param` требует обновления. Вторая проблема — это несовпадение свойств, определенных в теге `@return`, с реализацией. По всей видимости, оставшаяся часть проекта использует свойства `width` и `height`, следовательно, тег `@return` также должен быть обновлен.

Вы можете использовать JSDoc для постепенного добавления аннотаций типов в проект. Языковая служба TypeScript предложит вывод таких аннотаций в качестве быстрого исправления кода там, где это будет очевидно из контекста (рис. 8.2).

```
function double(val) {
  return 2 * val;
}
```

В результате получится корректная JSDoc-аннотация:

```
// @ts-check
/**
 * @param {number} val
```

```
*/
function double(val) {
    return 2 * val;
}
```

```
// @ts-check
```

```
function double(val) {
    return 2 *
}
```

(parameter) val: any

Parameter 'val' implicitly has an 'any' type, but a better type may be inferred from usage. ts(7044)

Quick Fix...

Infer parameter types from usage

Рис. 8.2. Языковые службы TypeScript предлагают быстрое решение для определения типов параметров

Но директива `// @ts-check` не всегда работает столь хорошо. Вот пример:

```
function loadData(data) {
    data.files.forEach(async file => {
        // ...
    });
}
```

Если вы используете быстрое исправление для аннотации `data`, то столкнетесь со следующим:

```
/**
 * @param {{
 *   files: { forEach: (arg0: (file: any) => Promise<void>) => void; };
 * }} data
 */
function loadData(data) {
    // ...
}
```

Здесь структурная типизация дала сбой (правило 4). Хотя технически функция будет работать с любым видом объекта, имеющего метод `forEach` с такой сигнатурой, но намерение заключалось в том, чтобы параметр был `{files: string[]}`.

JSDoc-аннотации и директивы `// @ts-check` расширят ваши возможности и не потребуют изменения существующего инструментария. Но все же лучше ими не увлекаться. Шаблоны комментариев не всегда уместны, так как логика легко может заплутать в JSDoc-пучине. TypeScript же лучше работает с файлами `.ts`, чем с `.js`. Однако `// @ts-check` может быть очень полезна для экспериментов с типами и поиска начальных ошибок, особенно в проектах, где в большом количестве использованы JSDoc-аннотации.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Добавляйте `// @ts-check` в шапку JavaScript-файла, чтобы включить проверку типов.
- ✓ Находите распространенные ошибки. Изучите, как объявлять глобальные переменные и добавлять декларации типов для сторонних библиотек.
- ✓ Используйте аннотации JSDoc для утверждений типов и лучшего их вывода.
- ✓ Не тратьте много времени, стараясь идеально типизировать код с помощью JSDoc. Помните, что конечная цель — это конвертация в `.ts`.

ПРАВИЛО 60. Используйте `allowJs` для совмещения TypeScript и JavaScript

Чем крупнее проект, тем дольше процесс его конвертирования в TypeScript. И постепенный перенос данных требует условий для сосуществования TypeScript и JavaScript.

Их обеспечивает опция компилятора `allowJs`, которая помогает файлам JavaScript и TypeScript импортировать друг друга. Для файлов JS этот режим допускает очень многое. Пока вы не включите `// @ts-check` (правило 59), вам будут представлены только синтаксические ошибки в качестве наиболее тривиальной демонстрации TypeScript как надмножества JavaScript.

Хотя опция `allowJs` и не предполагает обнаружения ошибок, она дает возможность ввести TypeScript в цепочку сборки, прежде чем приступить

к изменению кода. И это хорошо, так как вам понадобится возможность проводить тестирование при конвертации модулей в TypeScript (правило 61).

Удобно, если бандлер включает интегрированный TypeScript или имеет доступный плагин. Например, с помощью `browserify` вы запускаете `npm install --save-dev tsify` и добавляете его в качестве плагина:

```
$ browserify index.ts -p [ tsify --noImplicitAny ] > bundle.js
```

Большинство инструментов для модульного тестирования также имеют подобную опцию. Например, работая с инструментом `jest`, вы устанавливаете `ts-jest` и пропускаете исходные файлы TypeScript через него, определив файл `jest.config.js` следующим образом:

```
module.exports = {
  transform: {
    '^.+\\.tsx?$': 'ts-jest',
  },
};
```

Если речь идет о стандартной цепочке сборки, то задача окажется сложнее. Однако всегда есть резервный вариант. Когда вы определите опцию `outDir`, TypeScript будет генерировать чистые исходники JavaScript в директорию, параллельную дереву исходного кода, через которую будет запущена текущая цепочка сборки. Вам может потребоваться настроить вывод JavaScript-кода, производимого TypeScript, так чтобы он точнее совпадал с оригинальным JavaScript-кодом. Это делается посредством определения опций `target` и `module`.

Добавление TypeScript в процесс сборки и тестирования может оказаться не самым приятным делом, но с этого этапа начинается уверенная миграция кода.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Используйте опцию компилятора `allowJs` для совмещения JavaScript и TypeScript в процессе переноса проекта.
- ✓ Убедитесь в работоспособности тестов и цепочки сборки в TypeScript, прежде чем начинать полномасштабную миграцию.

ПРАВИЛО 61. Конвертируйте модуль за модулем, восходя по графу зависимостей

Вот вы и освоили современный JavaScript, преобразовав проект для использования модулей и классов ECMAScript (правило 58). Вы интегрировали TypeScript в цепочку сборки и провели все тесты (правило 60). Пора перейти к самой веселой части — преобразованию JavaScript в TypeScript. Но с чего же начать?

Добавляя типы в модуль, вы, скорее всего, будете получать ошибки типов в других модулях, зависящих от него. Поэтому стоит преобразовать каждый модуль отдельно, согласно графу зависимостей, — начиная с листов (модулей, не зависящих от других) и продвигаясь вверх к корню графа.

Вначале для миграции выбирайте модули, которые от вас не зависят (только вы от них). Как правило, это подразумевает установку модулей `@types`. Например, если вы используете библиотеку утилит `Lodash`, то запустите `npm install --save-dev @types/lodash`. Эти декларации помогут типам перемещаться по коду и выявлять ошибки в использовании библиотек.

Если ваш код вызывает внешние API, то вам может понадобиться добавить их декларации на ранней стадии. Хотя эти вызовы могут происходить в любой части кода, здесь работает принцип движения по графу зависимостей, поскольку вы зависите от API, а они от вас нет. Через API-вызовы проходит множество типов, которые сложно вывести с помощью контекста. Если у вас есть возможность использовать спецификацию API, генерируйте типы, исходя из нее (правило 35).

При миграции собственных модулей можете визуализировать граф зависимостей. На рис. 8.3 приведен пример, составленный с помощью инструмента `madge`, изображающий JavaScript-проект среднего размера.

Нижняя часть этого графа представляет круговую зависимость между `utils.js` и `tickers.js`. Есть много других модулей, которые зависят от этих двух, но они сами зависят только друг от друга. Это достаточно распространенный шаблон — большинство проектов будут иметь некий род модуля утилит внизу графа зависимостей.

В процессе миграции кода следует держать фокус на добавлении типов, а не на рефакторинге. Если мы говорим о старом проекте, то вы, скорее всего, заметите некие странные моменты и захотите их исправить. Не отвлекайтесь. Ближайшая цель — преобразовать проект в TypeScript, а не улучшить

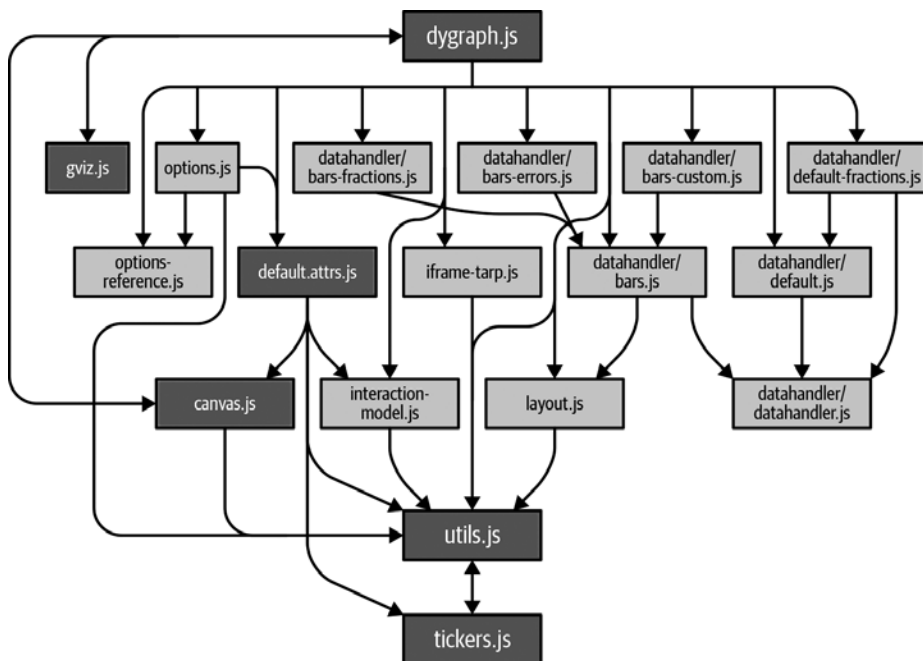


Рис. 8.3. Граф зависимостей для среднего JavaScript-проекта. Стрелки указывают зависимости. Темные сегменты обозначают, что модуль задействован в круговой зависимости

его структуру. Просто запишите замеченные вами проблемы и составьте список для будущего рефакторинга.

Есть несколько распространенных ошибок, с которыми вы столкнетесь при конвертации. Некоторые из них были рассмотрены в правиле 59, но есть и другие:

Необъявленные члены класса

Классы в JavaScript не требуют объявления их членов, но в TypeScript это необходимо. При переименовании файла классов `.js` в `.ts` могут появляться ошибки в свойствах, на которые ведут ссылки:

```
class Greeting {
  constructor(name) {
    this.greeting = 'Hello';
    // ~~~~~ Свойство 'greeting' не существует в типе 'Greeting'
    this.name = name;
  }
}
```

```

    // ~~~~ Свойство 'name' не существует в типе 'Greeting'
  }
  greet() {
    return this.greeting + ' ' + this.name;
    // ~~~~~~ ~~~~ Свойство ... не существует.
  }
}

```

Есть эффективный способ быстрого устранения этой проблемы (рис. 8.4).

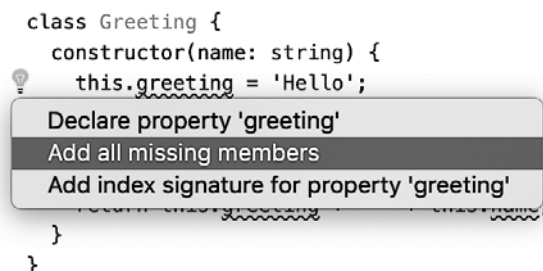


Рис. 8.4. Быстрое исправление добавления объявлений для отсутствующих членов особенно полезно при преобразовании класса в TypeScript

Это добавит декларации для недостающих членов, основываясь на их использовании:

```

class Greeting { greeting: string;
  name: any; constructor(name) {
    this.greeting = 'Hello';
    this.name = name;
  }
  greet() {
    return this.greeting + ' ' + this.name;
  }
}

```

TypeScript смог верно понять тип `greeting`, но не тип `name`. После такого исправления просмотрите список свойств и исправьте типы `any`.

Если вы первый раз видите полный перечень свойств класса, то может возникнуть некоторый шок. Когда я конвертировал основной класс в `digraph.js` (корневой модуль на рис. 8.1), то обнаружил, что он имеет не менее 45 переменных членов. Миграция в TypeScript предусматривает способ выявить подобные плохие структуры, которые прежде были скрыты.

Сложно обосновать плохой дизайн, когда приходится на него смотреть. Но не спешите сразу делать рефакторинг. Обратите внимание на отклонения и подумайте о том, как это можно исправить в дальнейшем.

Значения с изменяющимися типами

TypeScript обозначит проблемы в подобном коде:

```
const state = {};  
state.name = 'New York';  
    // ~~~~ Свойство 'name' не существует в типе '{}'.  
state.capital = 'Albany';  
    // ~~~~ Свойство 'capital' не существует в типе '{}'.  

```

Эта тема раскрыта в правиле 23. Вы можете освежить ее в памяти, когда столкнетесь с подобной ошибкой. Если требуются незначительные исправления, то можно просто собрать объект полностью:

```
const state = {  
    name: 'New York',  
    capital: 'Albany',  
}; // ok
```

Если все сложнее, то самое время использовать преобразование типа:

```
interface State {  
    name: string;  
    capital: string;  
}  
const state = {} as State;  
state.name = 'New York'; // ok  
state.capital = 'Albany'; // ok
```

Потом это надо будет исправить (правило 9), но в данный момент такая уловка поможет вам продолжить миграцию.

Если вы использовали JSDoc и `// @ts-check` (правило 59), имейте в виду, что при переносе в TypeScript вы можете утратить безопасность типов.

Например, TypeScript обозначает ошибку в этом JavaScript-коде:

```
// @ts-check  
/**  
 * @param {number} num  
 */
```

```
function double(num) {
    return 2 * num;
}

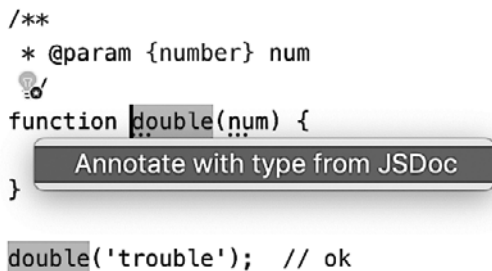
double('trouble');
// ~~~~~ Аргумент типа '"trouble"' не может быть присвоен
//          параметру типа 'number'.
```

При конвертации в TypeScript JSDoc и директива `// @ts-check` перестают выполняться. Это означает, что `num` неявно имеет тип `any` и ошибка не возникает:

```
/**
 * @param {number} num
 */
function double(num) {
    return 2 * num;
}

double('trouble'); // ok
```

К счастью, есть быстрый способ исправить это, переместив аннотации JSDoc-типов в типы TypeScript. Если в коде присутствует JSDoc, сделайте следующее (рис. 8.5).



```
/**
 * @param {number} num
 */
function double(num) {
    double('trouble'); // ok
```

Рис. 8.5. Быстрое исправление для копирования аннотаций JSDoc в аннотации типа TypeScript

Как только вы скопировали аннотации типов в TypeScript, обязательно удалите их из JSDoc во избежание повторов (правило 30):

```
function double(num: number) {
    return 2 * num;
}
```

```
double('trouble');  
// ~~~~~ Аргумент типа '"trouble"' не может быть присвоен  
// параметру типа 'number'.
```

Эта проблема будет также обнаружена при включении `noImplicitAny`, но добавить типы можно уже сейчас.

Переносите тесты в последнюю очередь. Они должны находиться на вершине вашего графа зависимостей (поскольку код от них не зависит). Также очень полезно убедиться, что тесты продолжают успешно выполняться в процессе миграции, несмотря на то что вы их не изменяли.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Начинайте миграцию с добавления `@types` для сторонних модулей и внешних API-вызовов.
- ✓ Проводите миграцию модулей согласно графу зависимостей снизу вверх. Первый модуль, как правило, будет неким кодом утилит. Используйте визуализацию графа для отслеживания прогресса.
- ✓ Не спешите делать рефакторинг кода при обнаружении проблем в его структуре. Составьте список идей относительно дальнейшего рефакторинга, но не отклоняйтесь от процесса переноса.
- ✓ Учитывайте распространенные ошибки, возникающие в процессе миграции. При необходимости скопируйте JSDoc-аннотации для сохранения безопасности типов после конвертации.

ПРАВИЛО 62. Не считайте миграцию завершенной, пока не включите `noImplicitAny`

Преобразование всего проекта в `.ts` — это крупное достижение. Но на этом работа не заканчивается. Следующий этап подразумевает включение опции `noImplicitAny` (правило 2). TypeScript-код с выключенной `noImplicitAny` может утаить реальные ошибки, допущенные в декларациях типов.

Например, вы использовали быстрое исправление «Добавить все недостающие члены» для добавления деклараций свойств в класс (правило 61). В итоге образовался тип `any`, и вы хотите это исправить:

```
class Chart {
  indices: any;

  // ...
}
```

Похоже, что `indices` должен быть массивом чисел, и вы вставляете соответствующий тип:

```
class Chart {
  indices: number[];

  // ...
}
```

Новых ошибок нет, и вы двигаетесь далее. К несчастью, вы ошиблись, и `number[]` — неверный тип. Вот часть кода из другой части класса:

```
getRanges() {
  for (const r of this.indices) {
    const low = r[0]; // Тип any
    const high = r[1]; // Тип any
    // ...
  }
}
```

Однозначно, что тип `number[] []` или `{number, number} []` будет более подходящим. Вас не удивляет, что допускается индексация в `number`? Воспринимайте это как пример того, насколько ненадежен может быть TypeScript без опции `noImplicitAny`.

Если же ее включить, то это допущение становится ошибкой:

```
getRanges() {
  for (const r of this.indices) {
    const low = r[0];
    // ~~~ Элемент неявно имеет тип 'any', потому что
    //      тип 'Number' не имеет сигнатуры индекса.
    const high = r[1];
    // ~~~~ Элемент неявно имеет тип 'any', потому что
    //       тип 'Number' не имеет сигнатуры индекса.
    // ...
  }
}
```

Хорошей стратегией по включению `noImplicitAny` будет ее установка в локальном клиенте и переход к исправлению ошибок. Их число, сообщаемое

модулем проверки, даст уверенное ощущение прогресса. Вы сможете делать коммит исправлений в типах, не делая коммит файла `tsconfig.json` до устранения всех ошибок.

Есть еще много рукояток, за которые вы можете потянуть для повышения строгости проверки типов. Самая существенная — это `"strict": true`. Однако `noImplicitAny` является наиболее важной. С ее помощью ваш проект способен получить большую часть преимуществ TypeScript даже без активации других опций вроде `strictNullChecks`. Дайте каждому участнику вашей команды возможность привыкнуть к TypeScript, прежде чем вводить более строгие настройки.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Не считайте миграцию завершенной, пока не включите `noImplicitAny`. Упрощенная проверка типов может не обнаружить реальных ошибок в декларациях.
- ✓ Исправляйте ошибки типов постепенно, прежде чем активировать `noImplicitAny`. Дайте вашей команде возможность достаточно освоить TypeScript и лишь потом включайте более строгие проверки.

Об авторе

Дэн Вандеркам является главным разработчиком ПО в Sidewalk Labs. Ранее работал над программой по визуализации генома в Школе медицины Икана Медицинского центра Маунт-Синай, а также над поисковыми возможностями Google, используемыми миллионами людей (запросы: «закат в Нью Йорке» или «население Франции»). Дэн имеет богатый опыт разработки проектов с открытым исходным кодом и является соорганизатором семинаров NYC TypeScript.

В свободное от программирования время он любит покорять скалы и играть в бридж. Его статьи можно найти на портале Medium и сайте danvk.org. Получил степень бакалавра computer science в Университете Райса в Хьюстоне, Техас. Проживает в Бруклине, Нью-Йорк.

Об обложке

На обложке книги — красноклювый буйволоковый скворец (*Buphagus erythrorhynchus*). Ареал вида охватывает Эфиопию и Сомали, Кению, Танзанию, Малави, Замбию, Ботсвану, Зимбабве, юг Мозамбика и северо-восточную часть ЮАР; но можно сказать, что эти птицы держатся рядом с пастбищными животными, на которых они проводят почти всю свою жизнь.

Красноклювые скворцы — родственники скворцов и майн, но при этом представляют собой отдельное и самостоятельное семейство. Птица длиной 18–20 см. Спина темно-коричневая. Брюхо охристого цвета. Ноги черные, с крепкими когтями. Самая приметная часть — эффектный красный клюв, иногда с желтым кончиком на надклювье.

На образ жизни птицы влияет то, как она находит пищу: красноклювые скворцы питаются клещами и другими паразитами животных. Часто сидят на антилопах (куду и импала), а также на зебрах, жирафах, буйволах и носорогах (кроме слонов). Добывать пищу птицам помогает плоский клюв, которым они проникают в густую шерсть животных, и острые когти. Жесткий хвост помогает удерживать равновесие. В период ухаживаний птицы также сидят на животном-хозяине и покидают его только во время гнездования. Птицы выращивают трех птенцов в гнезде рядом со стадами животных, чтобы можно было прокормить себя и своих детенышей.

Отношения птиц с животными-хозяевами когда-то рассматривались как четкий и классический пример мутализма (взаимовыгодного взаимодействия между видами). Но недавние исследования показали, что привычки кормления скворцов не оказывают существенного влияния на паразитную нагрузку хозяев. Кроме этого, выяснилось, что птицы трудятся над тем, чтобы держать раны животных открытыми, что позволяет скворцам питаться их кровью.

Красноклювые буйволоковые скворцы широко распространены по ареалу обитания, однако использование пестицидов является для них угрозой. Многие из животных, изображенных на обложках книг O'Reilly, находятся под угрозой исчезновения; все они важны для мира.

Иллюстрация на обложке — Хосе Марзан. За основу взята черно-белая гравюра из *Elements of Ornithology*.

Дэн Вандеркам
Эффективный TypeScript: 62 способа улучшить код

Перевел с английского *Д. Акуратер*

Заведующая редакцией	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>М. Колесников</i>
Ведущий редактор	<i>К. Тульцева</i>
Литературный редактор	<i>А. Руденко</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>Н. Викторова, М. Молчанова</i>
Верстка	<i>Л. Егорова</i>

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: 04.2020. 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 27.03.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 23,220. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».

142300, Московская область, г. Чехов, ул. Полиграфистов, 1.

Сайт: www.chpk.ru. E-mail: marketing@chpk.ru

Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электrozаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:

тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanova@piter.com
Подробная информация здесь: <http://www.piter.com/page/avtoru>

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: тел./факс: (812) 703-73-73; e-mail: sales@piter.com

Заказ книг для вузов и библиотек:

тел./факс: (812) 703-73-73, гоб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: на сайте www.piter.com; тел.: (812) 703-73-74, гоб. 6216;
e-mail: books@piter.com

Вопросы по продаже электронных книг: тел.: (812) 703-73-74, гоб. 6217;
e-mail: kuznetsov@piter.com

ВАША УНИКАЛЬНАЯ КНИГА

Хотите издать свою книгу? Она станет идеальным подарком для партнеров и друзей, отличным инструментом для продвижения вашего бренда, презентом для памятных событий! Мы сможем осуществить ваши любые, даже самые смелые и сложные, идеи и проекты.

МЫ ПРЕДЛАГАЕМ:

- издать вашу книгу
- издание книги для использования в маркетинговых активностях
- книги как корпоративные подарки
- рекламу в книгах
- издание корпоративной библиотеки

Почему надо выбрать именно нас:

Издательству «Питер» более 20 лет. Наш опыт — гарантия высокого качества.

Мы предлагаем:

- услуги по обработке и доработке вашего текста
- современный дизайн от профессионалов
- высокий уровень полиграфического исполнения
- продажу вашей книги во всех книжных магазинах страны

Обеспечим продвижение вашей книги:

- рекламой в профильных СМИ и местах продаж
- рецензиями в ведущих книжных изданиях
- интернет-поддержкой рекламной кампании

Мы имеем собственную сеть дистрибуции по всей России, а также на Украине и в Беларуси. Сотрудничаем с крупнейшими книжными магазинами. Издательство «Питер» является постоянным участником многих конференций и семинаров, которые предоставляют широкую возможность реализации книг.

Мы обязательно проследим, чтобы ваша книга постоянно имелась в наличии в магазинах и была выложена на самых видных местах.

Обеспечим индивидуальный подход к каждому клиенту, эксклюзивный дизайн, любой тираж.

Кроме того, предлагаем вам выпустить электронную книгу. Мы разместим ее в крупнейших интернет-магазинах. Книга будет сверстана в формате ePub или PDF — самых популярных и надежных форматах на сегодняшний день.

Свяжитесь с нами прямо сейчас:

Санкт-Петербург – Анна Титова, (812) 703-73-73, titova@piter.com

Москва – Сергей Клебанов, (495) 234-38-15, klebanov@piter.com