ГЛАВА 1

Вступление

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

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

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

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

БЕЗОПАСНОСТЬ ТИПОВ

Использование типов для предотвращения неверного поведения программ[1].

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

Вот несколько примеров неверного поведения кода:

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

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

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

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

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

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

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

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

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

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

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

Когда именно JavaScript сообщает об ошибке?

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

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

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

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

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

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

Если вы готовы начать путешествие, тогда приступим!

---

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

ГЛАВА 2

TypeScript с высоты птичьего полета

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

Компилятор

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

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

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

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

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

«Причем здесь безопасность кода?» — спросите вы.

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

МОДУЛЬ ПРОВЕРКИ ТИПОВ

Специальная программа, определяющая типобезопасность кода.

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

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

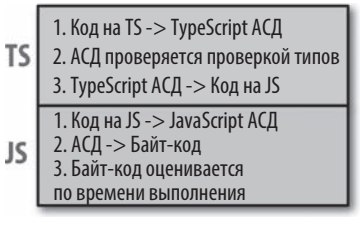

Рис. 2.1. Компиляция и запуск TypeScript

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

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

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

Система типов

В современных языках реализованы различные системы типов.

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

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

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

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

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

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

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

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

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

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

TypeScript vs JavaScript

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

Таблица 2.1. Сравнение систем типов JavaScript и TypeScript

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

Как связываются типы

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

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

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

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

Конвертируются ли типы автоматически

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

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

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

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

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

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

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

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

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

Когда проверяются типы

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

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

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

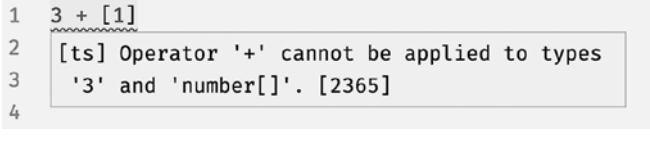

Рис. 2.2. VSCode сообщает об ошибке типа

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