ГЛАВА 1

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? Поддерживаются ли в нем типы с неопределенными значениями (nullable), функции `any` и утиная типизация?

TypeScript — необычный язык. Он не выполняется в интерпретаторе (как Ruby или Python) и не компилируется в низкоуровневый язык (как Java или C). Программа транспилируется в другой высокоуровневый язык — JavaScript. Поэтому выполняться будет именно JavaScript, а не TypeScript. Очень важно понимать связь TypeScript с JavaScript, и это поможет вам стать более эффективным разработчиком на TypeScript.

Система типов в TypeScript также имеет свои особенности, о которых нужно знать. Далее мы рассмотрим систему типов подробнее, а здесь выделим лишь самые важные моменты.

Стоит прочитать эту главу, даже если вы уже написали немало TypeScript-кода. Она поможет сформировать правильную картину того, что собой представляет TypeScript и как работает его система типов; кроме того, она поможет избавиться от некоторых ошибочных представлений, о которых вы, возможно, и не подозревали.

Правило 1. Разберитесь, как TypeScript связан с JavaScript

Каждому, кто достаточно долго пользовался TypeScript, наверняка попадалось выражение: «TypeScript — это надмножество JavaScript» или «TypeScript — это типизированное расширение JavaScript». Но что конкретно имеется в виду? Какая на самом деле связь между TypeScript и JavaScript? Важно хорошо понимать это, чтобы эффективно использовать JavaScript.

A является «надмножеством» *B*, если каждый элемент *B* также входит в *A*. В синтаксическом смысле TypeScript действительно надмножество JavaScript.

Любая программа JavaScript, не содержащая синтаксических ошибок, также является программой TypeScript. Вполне вероятно, что модуль контроля типов в TypeScript пометит некоторые проблемные места в коде, но это независимая задача. TypeScript все равно обработает ваш код и сгенерирует JavaScript. (Это еще одно ключевое свойство их взаимоотношений. Подробно рассмотрим его в правиле 3.)

Файлы TypeScript имеют расширение .ts вместо расширения .js для JavaScript[1]. Но это вовсе не означает, что TypeScript — совершенно новый язык. Поскольку TypeScript — это надмножество JavaScript, код в файлах .js уже является кодом TypeScript. Переименование файла main.js в main.ts не изменит этого обстоятельства.

Это очень удобно, если вы проводите миграцию существующей кодовой базы JavaScript на TypeScript. Ведь вам не придется переписывать свой код на другой язык, чтобы начать пользоваться TypeScript с его дополнительными преимуществами. Это бы не сработало, если бы вы решили переписать код JavaScript на язык вроде Java. Плавный процесс миграции кода — это одна из лучших особенностей TypeScript. Эту тему более подробно мы рассмотрим в главе 10.

Все программы JS являются программами TS, но обратное справедливо не всегда: есть программы TS, которые не являются программами JS. Дело в том, что в TypeScript добавлен дополнительный синтаксис определения типов. (Также в нем добавляются другие фрагменты синтаксиса, главным образом по историческим причинам, — правило 53.)

К примеру, рабочая программа TypeScript может выглядеть так:

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

Но при попытке запустить ее через среду выполнения JavaScript, вы получите ошибку:

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

SyntaxError: Unexpected token :
```

Часть `: string` является аннотацией типа, относящейся к специфике TypeScript. Стоит вам использовать ее хотя бы один раз, как вы немедленно выйдете за рамки базового JavaScript (рис. 1.1).

[1] Вам также могут встретиться расширения .tsx, .jsx, .mts, .mjs и некоторые другие. Все они относятся к файлам TypeScript и JavaScript.

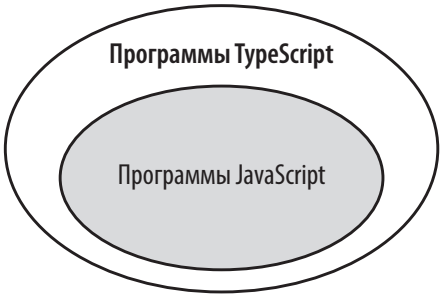

Рис. 1.1. Весь код JavaScript является кодом TypeScript, но не весь код TypeScript — код JavaScript

Это не означает, что TS бесполезен для базовых JS-программ: еще как полезен! Например, следующая программа JS:

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

выдает ошибку при попытке запустить ее:

```
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'. Вы имели в виду 'toUpperCase'?
```

В этой ситуации не нужно сообщать TypeScript, что `city` имеет тип `string`. Он вывел его, опираясь на исходное значение. Вывод типов — это ключевая часть TS, и в главе 3 рассказано о том, как правильно использовать его.

Одна из целей системы типов в TypeScript — обнаружение кода, который выдаст исключение при выполнении, без его фактического запуска. Поэтому иногда говорят, что в TS используется «статическая» система типов. Модуль проверки типов не всегда может обнаружить код, приводящий к выводу исключений, но приложит все усилия.

Даже если код не выдает исключение, возможно, он все еще делает не то, что вам нужно. TypeScript также старается обнаружить подобные проблемы. Например, следующая программа JavaScript:

```
const states = [
  {name: 'Alabama', capital: 'Montgomery'},
  {name: 'Alaska',  capital: 'Juneau'},
  {name: 'Arizona', capital: 'Phoenix'},
  // ...
];
```

```
for (const state of states) {
  console.log(state.capitol);
}
```

выдаст такой результат:

```
undefined
undefined
undefined
```

Что здесь пошло не так? Это рабочая JS-программа (а значит, и TS-программа). Она запустилась без сообщений об ошибках, но сделала совсем не то, что от нее ожидали. Даже без добавления аннотаций типов модуль их проверки способен обнаружить место ошибки и 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'?
}
```

Действительно, `capital` нужно было написать через «а».

Мало того что TypeScript умеет отлавливать ошибки даже при отсутствии аннотирования типов — он сделает это еще эффективнее, если вы проведете аннотирование. Дело в том, что прописанные типы явно сообщают TS о ваших *намерениях*, что позволяет ему легче обнаруживать места кода, им не соответствующие. Например, в обратной ситуации с опечаткой `capital`/`capitol`:

```
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'?
}
```

Подсказка, которая оказалась полезной 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 явным 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`.
  //                Вы имели в виду `capital`?
  // ...
];
for (const state of states) {
  console.log(state.capital);
}
```

Теперь ошибка соответствует проблеме, а предложенное решение корректно. Явно обозначая свои намерения, 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'},
  // ...
];
```

Такая динамика будет часто встречаться при использовании модуля проверки типов: чем больше информации вы ему предоставите, тем больше проблем он сможет найти.

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

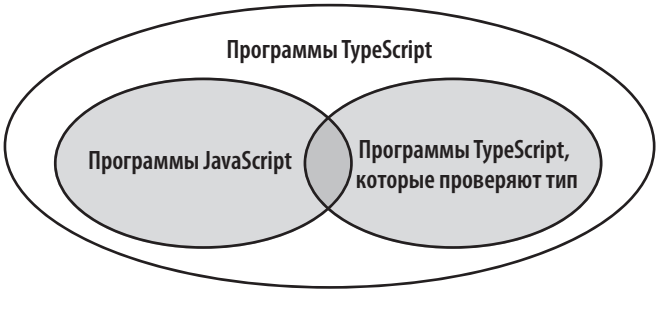

Рис. 1.2. Все программы JavaScript являются программами TypeScript. Тем не менее лишь некоторые программы JavaScript (и TypeScript) проходят проверку типов

Если утверждение «TypeScript — это надмножество JavaScript» кажется вам ошибочным, это может быть связано с тем, что вы подразумеваете третью категорию программ на этой диаграмме. На практике это самый актуальный вариант повседневного использования TypeScript. Как правило, при использовании TypeScript вы стремитесь к тому, чтобы ваш код проходил все проверки типов.

Система типов TypeScript *моделирует* процесс выполнения JavaScript. Это может принести сюрпризы, если вы привыкли к языку с более строгими проверками на стадии 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 оба сомнительны и вызвали бы ошибки выполнения во многих других языках. Но в данном случае точно моделируется поведение JavaScript, где результатом вычисления выражений будет строка `"23"`.

Но в TypeScript все же есть 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; // Дает результат 7 в JS
//        ~~~~ Значение 'null' не может здесь использоваться
const b = [] + 12; // Дает результат '12' в JS
//        ~~~~~~~ Оператор '+' не может применяться к типам ...
alert('Hello', 'TypeScript'); // выдает "Hello".
// ~~~~~~~~~~~~ Ожидается 0 или 1 аргумент, но получено 2.
```

Базовый принцип системы типов TypeScript заключается в моделировании поведения JavaScript при выполнении программы. 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 решает, когда моделировать поведение JavaScript при выполнении, а когда выйти за его рамки? В конечном итоге это дело вкуса. Используя TypeScript, вы доверяетесь команде его разработчиков. Может, вам просто нравится складывать `null` и `7` или `[]` и `12`? Или вызывать функции с лишними аргументами? Если так, то тогда TypeScript не для вас.

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 properties of undefined (reading 'toUpperCase')
```

TypeScript предполагает, что произошло обращение к массиву с выходом за его границу, но это не так. Результатом стало исключение.

Ошибки также ускользают от обнаружения при использовании типа any, который рассматривается в правиле 5, а еще подробнее — в главе 5.

Основная причина исключений кроется в том, что представление TypeScript о типе значения (статический тип) расходится с реальным. Система типов, которая гарантирует точность своих статических типов, называется *надежной* или *безопасной* (sound). Система типов TypeScript надежной определенно не является, да это и не планировалось. В правиле 48 рассматриваются другие возможные причины ненадежности (unsoundness).

Если для вас важна надежность, обратите внимание на другие языки: Reason, PureScript или Dart. Хотя они предоставляют больше гарантий безопасности на стадии выполнения, за все приходится платить: чтобы убедить их модули проверки типов в правильности вашего кода, придется приложить больше усилий, и ни один из них не является надмножеством JavaScript, что повышает сложность миграции.

СЛЕДУЕТ ЗАПОМНИТЬ

- TypeScript — это надмножество JavaScript. Все программы JavaScript — синтаксически правильные программы TypeScript, но не все программы TypeScript — синтаксически правильные программы JavaScript.
- В TypeScript добавляется система типов, которая моделирует поведение JavaScript на стадии выполнения, а также обнаруживает код, который при выполнении выдаст исключение.
- Возможны случаи, когда код проходит проверку типов, но при выполнении выдает ошибку.
- В TypeScript запрещены некоторые формально допустимые, но сомнительные конструкции JavaScript, например вызов функций с неправильным количеством аргументов.
- Аннотации типов сообщают TypeScript ваши намерения и помогают отличать правильный код от неправильного.

Правило 2. Выбирайте нужные опции в TypeScript

Пройдет ли этот код проверку типов?

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

Это невозможно определить заранее — все зависит от того, какие именно опции вы используете. На момент написания книги в компиляторе TypeScript их насчитывалось более сотни.

Их можно задать в командной строке:

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

А также через файл конфигурации tsconfig.json:

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

Лучше использовать файл конфигурации, чтобы ваши коллеги понимали, а вспомогательные инструменты могли интерпретировать то, как вы собираетесь использовать TypeScript. Для создания файла запустите `tsc --init`.

Многие из опций конфигурации определяют, откуда берутся исходные файлы и какой результат должен генерироваться на выходе. Несколько отдельных элементов управляют основными аспектами 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. Чтобы эффективно использовать его, нужно понимать самые важные из них: `noImplicitAny` и `strictNullChecks`.

noImplicitAny

`noImplicitAny` управляет тем, что должен делать TypeScript, если ему не удается определить тип переменной. Следующий код будет работать, если `noImplicitAny` присвоено значение `off`:

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

Если вы наведете курсор на элемент `add` в редакторе, 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. Это полезный инструмент, но пользоваться им стоит осторожно. Тема `any` намного подробнее рассматривается в правиле 5 и главе 5.

Такая конструкция называется «*неявным* `any`», потому что слово `any` нигде не указывается явно, но вы все равно работаете с рискованными типами `any`. В следующем фрагменте это приведет к ошибке при включении опции `noImplicitAny`:

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

Такие ошибки можно устранить явным указанием объявления типа `: any` или более конкретного типа:

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

TypeScript особенно 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 без `noImplicitAny` покажется вам совершенно другим языком.

Каждый новый проект следует начинать с включения этой опции, чтобы прописывать типы в процессе написания кода. Это поможет TypeScript обнаруживать проблемы, улучшит читаемость кода, а вашу работу сделает более комфортной (см. правило 6).

Отключение `noImplicitAny` уместно только при миграции JS-проекта в TypeScript (глава 10). Но даже в таком случае это должно быть временной мерой, и опцию следует включить как можно скорее. TypeScript без `noImplicitAny` оказывается на удивление слабо защищенным. В правиле 83 рассказано, какие проблемы при этом могут возникнуть.

strictNullChecks

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

Следующий код действителен, если `strictNullChecks` присвоено значение `off`:

```
const x: number = null; // OK, null считается допустимым числом
```

Этот же код выдаст ошибку при включенном режиме `strictNullChecks`:

```
const x: number = null;
//    ~ Тип 'null' не может быть присвоен типу 'number'.
```

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

Если вы намерены использовать `null`, то исправить ошибку можно, выражая ваши намерения более явно:

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

Если вы не хотите разрешать `null`, отследите, откуда взялось это значение, и добавьте проверку или утверждение:

```
const statusEl = document.getElementById('status');
statusEl.textContent = 'Ready';
// ~~~~~ 'statusEl' is possibly 'null'.

if (statusEl) {
  statusEl.textContent = 'Ready';  // OK, null исключено.
}
statusEl!.textContent = 'Ready';  // OK, мы проверили, что el отлично от null.
```

Подобное использование команды `if` называется «сужением» или «уточнением» типа. Этот паттерн рассматривается в правиле 22. «!» в последней строке называется «утверждением недопустимости `null`». Утверждения типов находят свое место в TypeScript, но также могут приводить к исключениям на стадии выполнения. В правиле 9 объясняется, где следует (или не следует) использовать утверждения типов.

Опция `strictNullChecks` полезна для обнаружения ошибок, связанных со значениями `null` и `undefined`, но очень усложняет язык. Если вы начинаете новый проект и при этом уже использовали TypeScript, включите режим `strictNullChecks`. Но если вы новичок или переносите код из JavaScript, то, возможно, предпочтете оставить этот режим выключенным. Разумеется, нужно включить `noImplicitAny`, прежде чем включать `strictNullChecks`.

Если вы предпочитаете работать без `strictNullChecks`, следите за риском неприятных ошибок вроде «undefined is not an object». Каждая такая ошибка подсказывает, что стоит подумать о включении более жестких проверок. Активация проверки создаст еще больше сложностей по мере роста проекта, поэтому не затягивайте с решением о ее включении. В большей части кода TypeScript используется режим `strictNullChecks`, и вам стоит последовать этому примеру.

Другие опции

Есть много других опций, влияющих на семантику языка (например, `noImplicitThis` и `strictFunctionTypes`), но они менее значимы по сравнению с упомянутыми `noImplicitAny` и `strictNullChecks`. Для активации всех проверок нужно включить опцию `strict`. При включении `strict` TypeScript успешно обнаруживает большую часть ошибок, так что это должно быть вашей целью.

Также доступно несколько опций, которые можно назвать «более строгими, чем `strict`». Вы можете установить их, чтобы TypeScript еще активнее действовал при поиске ошибок в вашем коде. Одна из этих опций — `noUncheckedIndexedAccess` — помогает обнаруживать ошибки при обращениях к объектам и массивам. Например, следующий код не содержит ошибок типов в режиме `--strict`, но при попытке выполнения кода выдается исключение:

```
const tenses = ['past', 'present', 'future'];
tenses[3].toUpperCase();
```

С включенным режимом `noUncheckedIndexedAccess` будет обнаружена ошибка:

```
const tenses = ['past', 'present', 'future'];
tenses[3].toUpperCase();
// ~~~~~~ Возможно, объект содержит 'undefined'.
```

Но даром ничто не дается. Многие допустимые обращения будут помечены как потенциально неопределенные:

```
tenses[0].toUpperCase();
// ~~~~~~ Возможно, объект содержит 'undefined'.
```

В одних проектах TypeScript этот режим используется, в других нет. Как минимум следует знать о его существовании. Эту опцию подробно рассмотрим в правиле 48.

Разберитесь, какие опции используете. Если ваш коллега показывает свой пример TypeScript-кода с ошибками, а вы не можете воспроизвести их, то убедитесь, что ваши настройки компиляторов совпадают.

СЛЕДУЕТ ЗАПОМНИТЬ

- Компилятор TypeScript поддерживает различные опции, влияющие на основные аспекты функционирования языка.
- Для настройки конфигурации лучше использовать tsconfig.json, а не командную строку.
- Включите `noImplicitAny`, если только не переводите проект JavaScript на TypeScript.
- Используйте `strictNullChecks` для предотвращения таких ошибок выполнения, как «undefined is not an object».
- Активированная опция `strict` обеспечивает самую тщательную проверку, которую только может выполнить TypeScript.

Правило 3. Помните, что генерация кода не зависит от типов

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

- преобразует новейшую версию TypeScript/JavaScript в более старую версию JavaScript, которая работает во всех браузерах (транспиляция);
- проверяет код на наличие ошибок типов.

Как ни странно, эти действия полностью независимы друг от друга. Проверка типов не влияет на код JavaScript, генерируемый TypeScript. Поскольку выполняться будет именно JS, это означает, что ваши типы не смогут повлиять на то, как выполняется код.