Оглавление

Предисловие

Язык TypeScript и его компилятор имели огромный успех с момента своего выхода в конце 2012 года. Они быстро завоевали прочные позиции в сообществе разработчиков на JavaScript и продолжают набирать силу. Многие масштабные проекты на JavaScript, в том числе проекты Adobe, Mozilla и Asana, приняли решение перевести свою кодовую базу с JavaScript на TypeScript. В 2014 году команды Microsoft и Google объявили, что Angular 2.0 будет разрабатываться с использованием TypeScript, тем самым объединяя языки AtScript от Google и TypeScript от Microsoft в один.

Такое крупномасштабное отраслевое внедрение TypeScript показывает ценность данного языка, гибкость компилятора и повышение производительности, которое может быть достигнуто с помощью его богатого набора инструментов разработки. Помимо отраслевой поддержки, стандарты ECMAScript 6 и ECMAScript 7 становятся все ближе и ближе к публикации, а TypeScript предоставляет способ использования функций этих стандартов в наших приложениях сегодня.

Написание приложений на TypeScript стало еще более привлекательным благодаря большой коллекции файлов объявлений, созданных сообществом TypeScript. Эти файлы беспрепятственно интегрируют широкий спектр существующих фреймворков JavaScript в среду разработки TypeScript, что повышает производительность, принося с собой заблаговременное обнаружение ошибок и расширенные функции IntelliSense.

Однако язык JavaScript не ограничивается веб-браузерами. Теперь мы можем писать на JavaScript на стороне сервера, управлять приложениями для мобильных телефонов с помощью JavaScript и даже управлять микроустройствами, разработанными для интернета вещей, используя JavaScript. Поэтому все эти цели JavaScript достижимы для разработчика, пишущего на TypeScript, потому что TypeScript генерирует JavaScript.

Для кого эта книга

Эта книга представляет собой руководство по языку TypeScript, которое начинается с базовых понятий, а затем основывается на этих знаниях, чтобы представить более продвинутые возможности языка и фреймворки. Предварительных знаний JavaScript не требуется, хотя предполагается, что читатель обладает предварительными навыками программирования. Если вы хотите изучать TypeScript, эта книга предоставит вам все необходимые знания и навыки для работы с любым проектом на TypeScript. Если вы уже являетесь опытным разработчиком на JavaScript или TypeScript, то эта книга выведет ваши навыки на новый уровень. Вы узнаете, как использовать TypeScript со множеством современных фреймворков, и выберете лучший фреймворк, соответствующий требованиям вашего проекта. Вы будете исследовать методы разработки через тестирование, изучите стандартные шаблоны проектирования и узнаете, как собрать полностью готовое к использованию приложение на TypeScript.

О чем рассказывается в этой книге

`Глава 1` «*Инструменты TypeScript и параметры фреймворков*» подготавливает почву для начала разработки на TypeScript. В ней рассматриваются преимущества использования TypeScript в качестве языка и компилятора, а затем рассматривается создание полной среды разработки с использованием ряда популярных интегрированных сред разработки.

`Глава 2` «*Типы, переменные и методы функций*» знакомит читателя с языком TypeScript, начиная с базовых типов и их аннотаций, а затем переходит к обсуждению переменных, функций и расширенных возможностей языка.

`Глава 3` «*Интерфейсы, классы и наследование*» основана на работе из предыдущей главы и знакомит с объектно-ориентированными концепциями и возможностями интерфейсов, классов и наследования и затем показывает эти концепции в работе через шаблон проектирования Factory.

`В главе 4` «*Декораторы, обобщения и асинхронные функции*» обсуждаются более продвинутые языковые возможности декораторов и обобщений, прежде чем приступить к понятиям асинхронного программирования. Здесь показано, как язык TypeScript поддерживает эти асинхронные функции с помощью промисов и использования конструкций `async await`.

`Глава 5` «*Файлы объявлений и строгие опции компилятора*» рассказывает читателю о создании файла объявлений для существующего тела кода JavaScript, а затем перечисляет некоторые из наиболее распространенных синтаксисов, используемых при написании файлов объявлений, в виде шпаргалки. Затем обсуждаются строгие настройки компилятора, доступные для него, где они должны использоваться и какие преимущества дают.

Глава 6 «*Сторонние библиотеки*» рассказывает читателю, как использовать файлы объявлений из репозитория DefinLYTyped в среде разработки, а затем показывает, как написать код на TypeScript, совместимый с тремя популярными фреймворками JavaScript – Backbone, AngularJS (версия 1) и ExtJS.

В главе 7 «*Фреймворки, совместимые с TypeScript*» рассматриваются популярные фреймворки, которые полностью интегрированы в язык TypeScript. В ней исследуется парадигма MVC, а затем сравнивается, как данный шаблон проектирования реализован в Backbone, Aurelia, Angular 2 и React. Пример программы, которая использует ввод на основе форм, реализован в каждом их этих фреймворков.

Глава 8 «*Разработка через тестирование*» начинается с обсуждения того, что такое разработка через тестирование, а затем читатель проходит через процесс создания различных типов модульных тестов. Используя библиотеку Jasmine, вы увидите, как применять управляемые данными тесты и как тестировать асинхронную логику. Глава заканчивается обсуждением модулей, выполняющих тестирование, составлением отчетов о тестировании и использованием серверов непрерывной интеграции.

В главе 9 «*Тестирование фреймворков, совместимых с TypeScript*» показано, как протестировать пример приложения, созданного с использованием каждого из совместимых с TypeScript фреймворков. Здесь стратегия тестирования разбивается на тесты модели, тесты представления и тесты контроллера и показаны различия между стратегиями тестирования этих платформ.

В главе 10 «*Модуляризация*» рассказывается, что такое модули, как их можно использовать и о двух типах генерации модулей, которые поддерживает компилятор TypeScript: CommonJS и AMD, после чего показано, как можно применять модули с загрузчиками модулей, включая Require и SystemJS. Затем в этой главе подробно рассматривается использование модулей в Node для создания приложения Express. Наконец, обсуждается использование модулей в бессерверной среде при помощи функций AWS Lambda.

В главе 11 «*Объектно-ориентированное программирование*» обсуждаются концепции объектно-ориентированного программирования, а затем показано, как упорядочить компоненты приложения в соответствии с принципами ООП. Затем подробно рассматривается реализация передовых объектно-ориентированных практик, показывающих, как можно использовать шаблоны проектирования State и Mediator для управления сложными взаимодействиями пользовательского интерфейса.

В главе 12 «*Внедрение зависимости*» рассматриваются концепции размещения служб и внедрения зависимостей, а также способы их использования для решения типичных проблем разработки приложений. Затем показано, как реализовать простой фреймворк внедрения зависимостей, используя декораторы.

В главе 13 «*Создание приложений*» исследуются основные строительные блоки разработки веб-приложений, показывая, как интегрировать сервер Express и сайт на Angular, а также все важные механизмы авторизации, которые должен иметь любой сайт, с подробным обсуждением JWT-токенов. Наконец, в этой главе показано, как интегрировать социальный логин (способ авторизации с использованием существующего аккаунта в социальных сетях, таких как Google или Facebook).

В главе 14 «*Переходим к практике*» создается одностраничное приложение с использованием Angular и Express, объединяя все концепции и компоненты, собранные в книге, в одно приложение. Эти концепции включают в себя разработку через тестирование, государство и шаблоны State и Mediator, проектирование и использование конечных точек Express REST, принципы объектно-ориентированного проектирования и модуляризацию. В этой главе также рассматриваются общие методы при использовании наблюдаемых для обработки большинства типов взаимодействия REST API.

Чтобы получить максимальную отдачу от этой книги

Вам понадобится компилятор TypeScript и какой-нибудь редактор. Компилятор TypeScript доступен для Windows, MacOS и Linux в качестве плагина Node. В главе 1 «*Инструменты TypeScript и параметры фреймворков*» описана настройка среды разработки.

Загрузите файлы с примерами кода

Вы можете загрузить файлы с примерами кода для этой книги, используя свою учетную запись на сайте www.packt.com. Если вы приобрели эту книгу в другом месте, то можете посетить веб-сайт www.packt.com/support и зарегистрироваться, чтобы получить файлы по электронной почте.

Можно загрузить файлы с кодом, выполнив следующие действия.

1. Войдите или зарегистрируйтесь на сайте www.packt.com.
2. Выберите вкладку **ПОДДЕРЖКА**.
3. Нажмите **Загрузки кода и Ошибки**.
4. Введите название книги в поле поиска и следуйте инструкциям на экране.

Как только файл будет загружен, пожалуйста, убедитесь, что вы распаковали или извлекли папку, используя последнюю версию:

- WinRAR/7-Zip для Windows;
- Zipeg/iZip/UnRarX для Mac;
- 7-Zip/PeaZip для Linux.

Пакет кода для книги также размещен на GitHub по адресу `https://github.com/PacktPublishing/Mastering-TypeScript-3`. В случае обновления кода он будет обновлен в существующем репозитории GitHub.

У нас также есть другие пакеты кода из нашего богатого каталога книг и видео, доступных на `https://github.com/PacktPublishing`. Посмотрите их!

Скачать цветные изображения

Мы также предоставляем файл PDF с цветными изображениями скриншотов/диаграмм, используемых в этой книге. Вы можете скачать его по адресу: `http://www.packtpub.com/sites/default/files/downloads/9781789536706_ColorImages.pdf`.

Используемые условные обозначения

В этой книге используется ряд текстовых обозначений.

`Текст кода`: указывает кодовые слова в тексте, имена таблиц базы данных, имена папок, имена файлов, расширения файлов, пути, фиктивные URL-адреса, пользовательский ввод и маркеры Twitter. Пример: «Этот файл `gruntfile.js` необходим для настройки всех задач Grunt».

Блок кода выглядит следующим образом:

```
test = this is a string
test = 1
test = function (a, b) {
   return a + b;
}
```

Когда мы хотим обратить ваше внимание на определенную часть блока кода, соответствующие строки или элементы выделяются жирным шрифтом:

```
declare function describe(
   description: string,
   specDefinitions: () => void
): void;
```

Любой ввод или вывод командной строки записывается следующим образом:

```
npm install -g typcript
```

Bold: указывает на новый термин, важное слово или слова, которые вы видите на экране. Например, слова в меню или диалоговых окнах появляются в тексте следующим образом. Вот пример: «Выберите **Системная информация** на панели **администрирования**».

Предупреждения или важные заметки выглядят так.

Связаться с нами

Мы всегда рады письмам от наших читателей.

Общая обратная связь: если у вас есть вопросы касательно какого-либо аспекта этой книги, укажите ее название в теме вашего сообщения и напишите нам по адресу `customercare@packtpub.com`.

Ошибки: хотя мы и позаботились о точности контента, ошибки случаются. Если вы нашли ошибку в этой книге, мы будем благодарны, если вы сообщите нам об этом. Пожалуйста, посетите `www.packt.com/submit-errata`, выберите свою книгу, нажмите на ссылку Errata Submission Form и введите данные.

Пиратство: если вы обнаружите какие-либо незаконные копии наших изданий в любой форме в интернете, мы будем благодарны, если вы предоставите нам адрес местонахождения или название веб-сайта. Пожалуйста, свяжитесь с нами по `copyright@packt.com`, приведя ссылку на материал.

Если вы заинтересованы в том, чтобы стать автором: если есть тема, в которой вы разбираетесь, и вы заинтересованы в написании или публикации книги, посетите сайт `author.packtpub.com`.

Отзывы

Пожалуйста, оставьте отзыв. После того как вы прочитали и использовали эту книгу, почему бы не оставить отзыв на сайте, на котором вы ее приобрели? Затем потенциальные читатели смогут увидеть и использовать ваше объективное мнение, чтобы принять решения о покупке, а мы в Packt сможем понять, что вы думаете о наших продуктах, и наши авторы смогут увидеть ваши отзывы о своей книге. Спасибо!

Для получения дополнительной информации о Packt, пожалуйста, посетите `packt.com`.

Глава 1

Инструменты TypeScript и параметры фреймворков

JavaScript – это поистине вездесущий язык: чем больше вы смотрите, тем больше JavaScript работает в самых неожиданных местах. Почти каждый веб-сайт, который вы посещаете в современном мире, использует JavaScript, чтобы сделать сайт более отзывчивым, более читабельным или более привлекательным в использовании. Даже традиционные настольные приложения выходят в сеть. Там, где нам когда-то требовалось скачать и установить программу для создания диаграммы или написания документа, теперь у нас есть возможность делать все это в интернете, в рамках своего скромного браузера.

Это сила JavaScript. Она позволяет нам переосмыслить способ использования интернета, но также позволяет переосмыслить то, как мы используем веб-технологии. Например, Node дает возможность JavaScript запускаться на стороне сервера, предоставляя целые крупномасштабные веб-сайты с обработкой сеансов, балансировкой нагрузки и взаимодействием с базой данных. Однако эта перемена в концепции относительно веб-технологий – только начало.

Apache Cordova – это полноценный веб-сервер, который работает как собственное приложение для мобильных телефонов. Это означает, что мы можем создать приложение для мобильного телефона с использованием HTML, CSS и JavaScript, а затем взаимодействовать с акселерометром телефона, службами геолокации или хранилищем файлов. Таким образом, благодаря Cordova JavaScript и веб-технологии переместились в сферу собственных приложений для мобильных телефонов.

Аналогичным образом такие проекты, как *Kinoma*, используют JavaScript для управления устройствами для **интернета вещей** (**IoT**), работающими на крошечных микропроцессорах, встроенных во все виды устройств. **Espruino** – чип микроконтроллера, специально предназначенный для запуска JavaScript. Таким образом, JavaScript теперь может управлять микропроцессорами на встроенных устройствах.

Настольные приложения также могут быть написаны на JavaScript и взаимодействовать с файловой системой с помощью таких проектов, как *Electron*. Это позволяет выполнить однократную запись и последующий запуск в любой операционной системе из коробки. На самом деле два самых популярных редактора исходного кода, Atom и Visual Studio Code, были созданы с использованием *Electron*.

Поэтому изучение JavaScript означает, что у вас есть возможность создавать веб-сайты, приложения для мобильных телефонов, настольные приложения и приложения IoT для запуска на встроенных устройствах.

Язык JavaScript не сложен для изучения, но он создает проблемы при написании больших и непростых программ, особенно в команде, работающей над одним проектом. Одна из основных проблем заключается в том, что JavaScript является интерпретируемым языком, а следовательно, у него нет шага компиляции. Проверка всего написанного кода на предмет мелких ошибок означает, что вы должны запустить его в интерпретаторе. Также традиционно было трудно реализовать принципы ООП в исходном формате на языке, а для создания подходящего, удобного в сопровождении и понятного кода на JavaScript требуется большая осторожность и дисциплина. Для программистов, которые переходят с других сильно типизированных объектно-ориентированных языков, таких как Java, C # или C ++, JavaScript может показаться совершенно чужой средой, особенно при работе с его более ранними версиями.

TypeScript устраняет этот пробел. Это сильно типизированный объектно-ориентированный язык, использующий компилятор для генерации JavaScript. Компилятор будет выявлять ошибки в кодовой базе еще до запуска в интерпретаторе. TypeScript также позволяет использовать хорошо известные методы ООП и шаблоны проектирования для создания приложений на JavaScript. Имейте в виду, что сгенерированный JavaScript – это просто JavaScript, и поэтому он будет работать везде, где может работать JavaScript, – в браузере, на сервере, мобильном устройстве, на рабочем столе или даже во встроенном устройстве.

Эта глава состоит из двух основных разделов. Первый раздел представляет собой краткий обзор ряда преимуществ использования TypeScript, а второй раздел посвящен настройке среды разработки TypeScript.

Если вы опытный программист на TypeScript и уже настроили среду разработки, можете пропустить эту главу. Если вы никогда прежде не работали с TypeScript и выбрали эту книгу, потому что хотите понять, что может делать TypeScript, читайте дальше.

В этой главе мы рассмотрим следующие темы.

Преимущества TypeScript:

- компиляция;
- сильная типизация;
- интеграция с популярными библиотеками JavaScript;
- инкапсуляция;
- методы доступа public и private.

Настройка среды разработки:

- компиляция на основе Node;

- Visual Studio Code;
- Visual Studio 2017;
- WebStorm;
- другие редакторы и Grunt.

Что такое TypeScript?

TypeScript – это язык программирования, разработанный *Андерсом Хейлсбергом* (Anders Hejlsberg), создателем языка C #. Это результат оценки языка JavaScript и того, что можно сделать, чтобы помочь разработчикам при написании кода на JavaScript. TypeScript включает в себя компилятор, который преобразует код, написанный на TypeScript, в JavaScript. Его красота – в своей простоте. Мы можем взять существующий JavaScript, добавить несколько ключевых слов TypeScript тут и там и преобразовать свой код в сильно типизированную объектно-ориентированную кодовую базу с проверкой синтаксиса. Добавив шаг компиляции, мы можем проверить, что написали надежный код, который будет вести себя так, как мы хотели.

TypeScript генерирует JavaScript – все так просто. Это означает, что везде, где может использоваться JavaScript, TypeScript может применяться для генерации того же JavaScript, но с использованием проверок во время компиляции, чтобы гарантировать, что не нарушены определенные правила. Наличие этих дополнительных проверок еще до того, как мы запустим JavaScript, значительно экономит время, особенно там, где существуют большие команды разработчиков или где полученный код JavaScript публикуется как библиотека.

TypeScript также включает в себя языковую службу, которая может использоваться такими инструментами, как редакторы кода, чтобы помочь понять, как следует применять функции и библиотеки JavaScript. Эти редакторы могут затем автоматически предоставить программисту подсказки кода и советы по использованию данных библиотек.

Язык TypeScript, его компилятор и связанные с ним инструменты помогают разработчикам на JavaScript работать более продуктивно, быстрее находить ошибки и помогать друг другу понять, как использовать их код. Это дает нам возможность использовать протестированные концепции ООП и шаблоны проектирования в нашем коде JavaScript в очень простой и понятной форме. Давайте попробуем понять, как это происходит.

JavaScript и ECMAScript

JavaScript как язык существует уже давно. Первоначально разработанный как язык для поддержки HTML в одном веб-браузере, он вдохновил множество клонов языка, каждый со своими реализациями. В конце концов, был введен глобальный стандарт, позволяющий веб-сайтам поддерживать несколько браузеров. Язык, определенный в этом стандарте, называется **ECMAScript**.

Каждый интерпретатор JavaScript должен предоставлять функции и свойства, соответствующие стандарту ECMAScript. Стандарт ECMAScript, который был опубликован в 1999 году, официально назывался **ECMA-262**, **3-е издание**, но стал называться просто **ECMAScript 3**. Эта версия JavaScript получила широкое распространение и стала основой для взрывной популярности и роста интернета в том виде, в котором мы его знаем.

С популярностью языка и растущим использованием вне веб-браузера стандарт ECMAScript неоднократно пересматривался и обновлялся. К сожалению, время, которое требуется между предложением новых языковых функций и ратификацией нового стандарта для их покрытия, может быть довольно длительным. Даже когда публикуется новая версия стандарта, веб-браузеры принимают эти стандарты только с течением времени, а также могут реализовывать фрагменты стандарта раньше других.

Поэтому, прежде чем выбирать, какой стандарт принять, важно понять, какие браузеры или, точнее, какой механизм времени выполнения необходимо будет поддерживать. Чтобы поддержать эти решения, есть ряд справочных сайтов, которые перечисляют поддержку в так называемой таблице совместимости.

В настоящее время на выбор предлагается три основные версии ECMAScript: ES3, ES5 и недавно ратифицированная ES6. ES3 существует уже давно, и почти любой веб-браузер будет поддерживать ее. ES5 поддерживается большинством современных веб-браузеров. ES6 является последней версией стандарта и на сегодняшний день самым крупным обновлением языка до настоящего времени. Она впервые вводит в язык классы, облегчая реализацию объектно-ориентированного программирования.

Компилятор TypeScript имеет параметр, который может переключаться между различными версиями стандарта ECMAScript. В настоящее время TypeScript поддерживает ES3, ES5 и ES6. Когда компилятор запускает ваш TypeScript, он будет генерировать ошибки компиляции, если код, который вы пытаетесь компилировать, не соответствует этому стандарту. Команда Microsoft взяла на себя обязательство следовать стандартам ECMAScript в любых новых версиях компилятора TypeScript, поэтому по мере принятия новых редакций язык TypeScript и компилятор будут следовать их примеру.

Преимущества TypeScript

Чтобы получить представление о преимуществах TypeScript, давайте очень кратко рассмотрим некоторые моменты, которые TypeScript дает в таблице:

- компиляцию;
- сильную типизацию;
- интеграцию с популярными библиотеками JavaScript;
- инкапсуляцию;
- методы доступа public и private.

Компиляция

Одна из самых любимых черт JavaScript – отсутствие шага компиляции. Просто измените свой код, обновите браузер, а интерпретатор позаботится обо всем остальном. Нет необходимости ждать какое-то время, пока компилятор не закончит работу, чтобы запустить свой код.

Хотя можно рассматривать это как преимущество, есть много причин, по которым вы захотите использовать шаг компиляции. Компилятор может найти глупые ошибки, такие как пропущенные скобки или запятые. Он также может найти другие более невнятные ошибки, такие как использование одной кавычки (`'`) там, где должна использоваться двойная кавычка (`"`). Каждый разработчик JavaScript может рассказать страшные истории о часах, потраченных на поиск ошибок в своем коде, только для того, чтобы узнать, что он пропустил случайную закрывающую скобку `}` или простую запятую.

Использование шага компиляции в своем рабочем процессе действительно упрощает работу при управлении большой базой кода. Существует старая поговорка, в которой говорится, что мы должны рано выходить из строя и громко выходить из строя, а компилятор будет очень громко кричать на самой ранней возможной стадии, когда будут обнаружены ошибки. Это означает, что любая фиксация изменений в исходном коде будет свободна от ошибок, обнаруженных компилятором.

При внесении изменений в большую кодовую базу нам также необходимо убедиться, что мы не нарушаем существующую функциональность. В большой команде это часто означает использование ветвления и слияния репозитория исходного кода. Выполнение шага компиляции до, во время и после слияния из одной ветки в другую дает нам дополнительную уверенность в том, что мы не совершили ошибок или что в процессе автоматического слияния также не было допущено ошибок.

Если команда разработчиков использует процесс непрерывной интеграции, сервер **непрерывной интеграции** (**CI**) может быть ответственным за создание и развертывание всего сайта, а затем за выполнение набора модульных и интеграци-

онных тестов для вновь зарегистрированного кода. Мы можем сэкономить часы, которые уходят на сборку и тестирование, гарантируя отсутствие синтаксических ошибок в коде, прежде чем приступать к развертыванию и запуску тестов.

Наконец, как упоминалось ранее, компилятор TypeScript может быть настроен на вывод ES3, ES5 или ES6 JavaScript. Это означает, что мы можем ориентироваться на разные версии среды выполнения из одной и той же базы кода.

Сильная типизация

JavaScript не сильно типизирован. Это очень динамичный язык, поскольку он позволяет объектам изменять свои свойства и поведение на лету. В качестве примера рассмотрим приведенный ниже код:

```
var test = "this is a string;
console.log('test=' + test);

test = 1;
console.log('test=' + test);

test = function (a, b) {
   return a + b;
}

console.log('test=' + test);
```

В первой строке этого фрагмента объявлена переменная с именем `test` и ей присвоено строковое значение. Чтобы убедиться в этом, мы записали значение в консоль. Затем мы присваиваем числовое значение переменной `test` и снова записываем ее значение в консоль. Обратите внимание, однако, на последний фрагмент кода. Мы назначаем функцию, которая принимает два параметра для переменной `test`. Если мы запустим этот код, то получим следующие результаты:

```
test = this is a string
test = 1
test = function (a, b) {
   return a + b;
}
```

Здесь ясно видны изменения, которые мы вносим в переменную `test`. Она меняется, переходя от строкового значения к числовому, а затем становится функцией.

Изменение типа переменной во время выполнения может быть очень опасным занятием и привести к бесчисленным проблемам. Вот почему традиционные объектно-ориентированные языки обеспечивают сильную типизацию. Другими словами, они не позволяют природе переменной изменяться после объявления.

Хотя весь приведенный выше код является допустимым кодом JavaScript – и может быть оправдан, – довольно легко видно, как это может привести к ошибкам во время выполнения. Представьте, что вы были ответственны за написание библиотечной функции для добавления двух чисел, а затем еще одного разработчика, который непреднамеренно переназначил вашу функцию, вместо того чтобы вычитать эти числа.

Подобного рода ошибки можно легко обнаружить в нескольких строках кода, но находить и исправлять их по мере расширения вашей базы кода и команды разработчиков будет все труднее.

Синтаксический сахар

TypeScript представляет очень простой синтаксис для проверки типа объекта во время компиляции. Этот синтаксис известен как синтаксический сахар, или, более формально, аннотация типов. Рассмотрим приведенную ниже версию нашего исходного кода JavaScript, написанного на TypeScript:

```
var test: string = "this is a string";
test = 1;
test = function(a, b) { return a + b; }
```

Обратите внимание, что в первой строке этого фрагмента мы ввели двоеточие `:` и ключевое слово `string` между нашей переменной и ее присваиванием. Этот синтаксис аннотации типа означает, что мы устанавливаем тип нашей переменной как тип `string` и что любой код, который не придерживается этих правил, приведет к ошибке компиляции. Запуск предыдущего кода через компилятор TypeScript вызовет две ошибки:

```
hello.ts(3,1): error TS2322: Type 'number' is not assignable to
type 'string'.
hello.ts(4,1): error TS2322: Type '(a: any, b: any) => any' is
not assignable to type 'string'.
```

Первая ошибка довольно очевидна. Мы указали, что переменная `test` является строкой, и поэтому попытка присвоить ей номер вызовет ошибку компиляции. Вторая ошибка похожа на первую и, по сути, говорит о том, что мы не можем присвоить функцию строке.

Таким образом, компилятор TypeScript вводит сильную или статическую типизацию в нашем коде JavaScript, предоставляя нам все преимущества сильно типизированного языка. Поэтому TypeScript описывается как расширенный вариант JavaScript. Мы рассмотрим это более подробно в главе 2 *«Типы, переменные и методы функций»*.

Определение типов для популярных библиотек JavaScript

Как мы уже видели, TypeScript имеет возможность аннотировать JavaScript и вносить сильную типизацию в опыт разработки на JavaScript. Но как можно сильно типизировать существующие библиотеки JavaScript? Ответ на удивление прост: путем создания файла определения. TypeScript использует файлы с расширением `.d.ts` как своего рода заголовочный файл, по аналогии с такими языками, как C ++, для наложения сильной типизации на существующие библиотеки JavaScript. Эти файлы определений содержат информацию, которая описывает каждую доступную функцию и/или переменные, а также связанные с ними аннотации типов.

Давайте быстро посмотрим, как будет выглядеть определение. В качестве примера рассмотрим функцию из популярного фреймворка для модульного тестирования Jasmine под названием `describe`:

```
var describe = function(description, specDefinitions) {
   return jasmine.getEnv().describe(description, specDefinitions);
};
```

Обратите внимание на то, что у функции `describe` есть два параметра – `description` и `specDefinitions`. Но JavaScript не говорит нам, что это за переменные. Нам нужно взглянуть на документацию по Jasmine, чтобы выяснить, как вызвать эту функцию: если мы перейдем на страницу `http://jasmine.GitHub.io/2.0/introduction.html`, то увидим пример использования этой функции:

```
describe("A suite", function () {
   it("contains spec with an expectation", function () {
      expect(true).toBe(true);
   });
});
```

Таким образом, из документации явствует, что первый параметр является строкой, а второй – это функция. Но в JavaScript нет ничего, что заставляло бы нас соответствовать этому API. Как упоминалось ранее, мы могли бы легко вызвать эту функцию с двумя числами или непреднамеренно переключить параметры, отправив сначала функцию, а затем строку. Очевидно, мы начнем получать ошибки среды выполнения, если сделаем это, но TypeScript, используя файл определений, может генерировать ошибки времени компиляции, прежде чем мы попытаемся запустить этот код.

Давайте посмотрим на фрагмент файла определения `jasmine.d.ts`:

```
declare function describe(
   description: string,
```

```
    specDefinitions: () => void
): void;
```

Это определение TypeScript для функции `describe`. Во-первых, `declare function describe` говорит нам, что мы можем использовать функцию под названием describe, но реализация этой функции будет обеспечиваться в среде выполнения.

Ясно, что параметр `description` сильно типизирован, чтобы стать строкой (`string`), а параметр `specDefinitions` сильно типизирован, чтобы стать функцией (`function`), которая возвращает `void`. TypeScript использует двойные скобки `()` для объявления функций и стрелку для отображения типа возвращаемого значения функции. Следовательно, `() => void` – это функция, которая ничего не возвращает. В конечном итоге функция describe сама вернет void.

Представьте, что наш код должен попытаться передать функцию в качестве первого параметра и строку в качестве второго параметра (явно нарушая определение этой функции), как показано в приведенном ниже примере:

```
describe(() => { /* function body */}, "description");
```

В этом случае TypeScript выдаст следующую ошибку:

```
hello.ts(11,11): error TS2345: Argument of type '() => void'
is not assignable to parameter of type 'string'.
```

Эта ошибка говорит о том, что мы пытаемся вызвать функцию describe с недопустимыми параметрами. Мы рассмотрим файлы определений более подробно в последующих главах, но этот пример ясно показывает, что TypeScript будет генерировать ошибки, если мы попытаемся использовать внешние библиотеки JavaScript неправильно.

DefinitelyTyped

Вскоре после выхода TypeScript *Борис Янков* запустил GitHub-репозиторий для размещения файлов определений под названием DefiniteTyped (`http://definitetyped.org`). Этот репозиторий стал первым портом захода для интеграции внешних библиотек в TypeScript и в настоящее время содержит определения для более чем 1600 библиотек JavaScript. Рост этого сайта и скорость создания типов определений для множества библиотек JavaScript показывают популярность TypeScript.

Инкапсуляция

Одним из фундаментальных принципов объектно-ориентированного программирования является инкапсуляция, возможность определять данные, а также набор функций, которые могут работать с этими данными, в единый компонент.

Большинство языков программирования обладает концепцией класса для этой цели – предоставляя способ определения шаблона для данных и связанных функций.

Для начала давайте посмотрим на простое определение класса TypeScript:

```
class MyClass {
   add(x, y) {
      return x + y;
   }
}

var classInstance = new MyClass();
var result = classInstance.add(1,2);
console.log('add(1,2) returns ${result}');
```

Этот код довольно прост для чтения и понимания. Мы создали класс `MyClass` с простой функцией `add`. Чтобы использовать его, мы просто создаем его экземпляр класса и вызываем функцию `add` с двумя аргументами.

JavaScript, предшествующий ES6, не имеет оператора класса, но вместо этого использует функции для воспроизведения функциональности классов. Инкапсуляция через классы достигается с помощью либо шаблона прототипа, либо шаблона замыкания. Понимание прототипов и шаблона замыкания и их правильное использование считаются фундаментальным навыком при написании кода на JavaScript корпоративного уровня.

Замыкание – это, по сути, функция, которая ссылается на независимые переменные. Это означает, что переменные, определенные в функции замыкания, запоминают среду, в которой они были созданы. Это позволяет JavaScript определять локальные переменные и предоставлять инкапсуляцию. Запись определения `MyClass` в предыдущем коде с использованием замыкания в JavaScript будет выглядеть примерно так:

```
var MyClass = (function () {
   // самовызывающаяся функция – это среда,
   // которую замыкание запомнит;
   function MyClass() {
      // MyClass – это внутренняя функция,
      // замыкание;
   }
      MyClass.prototype.add = function (x, y) {
      return x + y;
   };
   return MyClass;
})();
```

```
var classInstance = new MyClass();
var result = classInstance.add(1, 2);
console.log("add(1,2) returns " + result);
```

Мы начнем с переменной `MyClass` и назначим ее функции, которая выполняется немедленно, – обратите внимание на `}) ();` синтаксис в нижней части определения замыкания. Этот синтаксис является распространенным способом написания кода на JavaScript во избежание утечки переменных в глобальное пространство имен. Затем мы определяем новую функцию с именем `MyClass` и возвращаем ее внешней вызывающей функции. После этого мы используем ключевое слово `prototype`, чтобы добавить новую функцию в определение `MyClass`. Эта функция называется `add` и принимает два параметра, возвращая их сумму.

Последние несколько строк предыдущего фрагмента кода показывают, как использовать это замыкание в JavaScript. Создайте экземпляр типа замыкания, а затем выполните функцию add. Выполнение этого кода будет регистрировать **`add(1,2) returns 3`** в консоль, как и ожидалось.

Сравнив код JavaScript с кодом TypeScript, легко увидеть, насколько просто выглядит TypeScript по сравнению с эквивалентным JavaScript. Помните, мы упоминали, что программисты на JavaScript могут легко потерять фигурную `{` или обычную `(` скобку? Посмотрите на последнюю строку в определении замыкания: `});` на обнаружение одной такой скобки может уйти несколько часов отладки.

Классы TypeScript генерируют замыкания

JavaScript, как показано ранее, на самом деле является результатом определения класса TypeScript. Итак, TypeScript действительно генерирует замыкания за вас.

О добавлении концепции классов в язык JavaScript говорилось годами, и в настоящее время она является частью **ECMAScript 6th Edition**. Microsoft взяла на себя обязательство следовать стандарту ECMAScript в компиляторе TypeScript, как и когда эти стандарты публикуются.

Методы доступа public и private

Еще один принцип ООП, который используется в инкапсуляции, – это концепция сокрытия данных, то есть способность иметь открытые и закрытые переменные. Закрытые переменные должны быть скрыты для пользователя определенного класса, поскольку должны использоваться только самим классом. Случайное раскрытие этих переменных может легко привести к ошибкам во время выполнения.

К сожалению, в JavaScript нет встроенного способа объявления переменных закрытыми. Хотя эту функциональность можно эмулировать с помощью замыканий, многие программисты на JavaScript просто используют символ подчеркива-

ния (_) для обозначения закрытой переменной. Хотя во время выполнения, если вы знаете имя закрытой переменной, вы можете легко присвоить ей значение. Рассмотрим приведенный ниже код:

```
var MyClass = (function() {
   function MyClass() {
      this._count = 0;
   }
   MyClass.prototype.countUp = function() {
      this._count ++;
   }
   MyClass.prototype.getCountUp = function() {
      return this._count;
   }
   return MyClass;
}());var test = new MyClass();
test._count = 17;
console.log("countUp : " + test.getCountUp());
```

Переменная `MyClass` на самом деле является замыканием с функцией-конструктором, функцией `countUp` и функцией `getCountUp`. Предполагается, что переменная `_count` является закрытой переменной-членом, которая используется только в рамках замыкания. Использование соглашения об именах подчеркивания дает пользователю этого класса некоторое представление о том, что переменная является закрытой, но JavaScript все равно позволит вам манипулировать переменной `_count`. Посмотрите на вторую последнюю строку фрагмента кода. Мы явно устанавливаем значение `_count` равным `17`, что разрешено JavaScript, но не желательно для автора класса. Результатом этого кода будет **`countUp: 17`**.

TypeScript, однако, использует ключевые слова `public` и `private`, которые можно использовать для переменных членов класса. Попытка получить доступ к переменной члена класса, которая была помечена как private, приведет к ошибке времени компиляции. Как пример этого предыдущий код может быть написан на TypeScript следующим образом:

```
class CountClass {
   private _count: number;
   constructor() {
      this._count = 0;
   }
   countUp() {
      this._count ++;
   }
   getCount() {
      return this._count;
   }
}
```

```
var countInstance = new CountClass() ;
countInstance._count = 17;
```

Здесь, во второй строке нашего фрагмента кода, мы объявили закрытую переменную-член с именем `_count`. Опять же, у нас есть конструктор, функция `countUp` и функция `getCount`. Если мы скомпилируем этот файл, компилятор выдаст ошибку:

```
hello.ts(39,15): error TS2341: Property '_count' is private and
only accessible within class 'CountClass'.
```

Эта ошибка появляется, потому что мы пытаемся получить доступ к закрытой переменной `_count` в последней строке кода.

Поэтому компилятор TypeScript помогает нам придерживаться открытых и закрытых методов доступа, выдавая ошибку компиляции, когда мы непреднамеренно нарушаем данное правило.

Помните, однако, что эти методы доступа являются всего лишь функцией времени компиляции и не влияют на сгенерированный код JavaScript. Вам необходимо помнить об этом, если вы пишете библиотеки JavaScript, которые будут использоваться третьими сторонами. Обратите внимание, что по умолчанию компилятор TypeScript по-прежнему генерирует выходной файл JavaScript, даже при наличии ошибок компиляции. Однако этот параметр можно изменить, чтобы компилятор TypeScript не генерировал JavaScript, если есть ошибки компиляции.

Интегрированные среды разработки TypeScript

Цель данного раздела – научить вас работать со средой TypeScript, чтобы вы могли редактировать, компилировать, запускать и отлаживать свой код, написанный на TypeScript. TypeScript был выпущен как проект с открытым исходным кодом и включает в себя как вариант для Windows, так и вариант для Node. Это означает, что компилятор будет работать в Windows, Linux, macOS и любой другой операционной системе, которая поддерживает Node. В средах Windows можно установить Visual Studio, которая зарегистрирует `tsc.exe` (компилятор TypeScript) в нашем каталоге `c:\Program Files`, либо можно использовать Node. В средах Linux и macOS нам нужно будет использовать Node.

В этом разделе мы рассмотрим следующие среды разработки:

- компиляцию на основе Node;
- Visual Studio Code;

- Visual Studio 2017;
- WebStorm;
- использование Grunt.

Компиляция на основе Node

Самая простая среда разработки TypeScript состоит из простого текстового редактора и компилятора TypeScript на основе Node. Перейдите на сайт Node (`https://nodejs.org`) и следуйте инструкциям по установке Node на выбранной вами операционной системе.

Как только Node будет установлен, можно установить TypeScript, просто набрав:

```
npm install -g typescript
```

Эта команда вызывает диспетчер пакетов Node (`npm`) для установки TypeScript в качестве глобального модуля (опция `-g`), который сделает его доступным независимо от того, в каком каталоге мы сейчас находимся. После установки TypeScript мы можем отобразить текущую версию компилятора, набрав следующее:

```
tsc -v
```

На момент написания этой главы версия компилятора TypeScript – 3.3.3, и поэтому вывод этой команды такой:

```
Version 3.3.3
```

Давайте теперь создадим файл TypeScript с именем `hello.ts` со следующим содержимым:

```
console.log('hello TypeScript');
```

Из командной строки мы можем использовать TypeScript для компиляции этого файла в файл JavaScript, выполнив следующую команду:

```
tsc hello.ts
```

Как только компилятор TypeScript завершит свою работу, он сгенерирует файл `hello.js` в текущем каталоге. Мы можем запустить этот файл с помощью Node, набрав:

```
node hello.js
```

После этого в консоль будет выведено:

```
hello TypeScript
```