

Оглавление

Введение	15
Благодарности	17
О книге	18
Для кого эта книга	18
Структура книги	18
О коде.....	20
Об авторах	21
Об обложке.....	22

Часть 1

Основы синтаксиса TypeScript

Глава 1. Знакомство с TypeScript	24
1.1. Зачем программировать в TypeScript.....	24
1.2. Типичные рабочие процессы TypeScript.....	29
1.3. Использование компилятора TypeScript	31
1.4. Знакомство с Visual Studio Code	35
Итоги.....	38
Глава 2. Базовые и пользовательские типы	39
2.1. Объявление переменных с типами	40
2.1.1. Базовые аннотации типов	41
2.1.2. Типы в объявлениях функций.....	45
2.1.3. Объединенный тип	47

2.2. Определение пользовательских типов	50
2.2.1. Использование <code>type</code>	50
2.2.2. Использование классов в качестве пользовательских типов	52
2.2.3. Интерфейсы в качестве пользовательских типов	54
2.2.4. Структурная система типов против номинальной	57
2.2.5. Пользовательские объединения типов	60
2.3. Типы <code>any</code> и <code>unknown</code> , а также пользовательские защиты типов	62
2.4. Мини-проект	64
Итоги	65
Глава 3. Объектно-ориентированное программирование с классами и интерфейсами	67
3.1. Работа с классами	68
3.1.1. Знакомство с наследованием классов	68
3.1.2. Модификаторы доступа <code>public</code> , <code>private</code> , <code>protected</code>	70
3.1.3. Статические переменные и пример <code>Одиночки</code>	73
3.1.4. Метод <code>super()</code> и ключевое слово <code>super</code>	76
3.1.5. Абстрактные классы	78
3.1.6. Перегрузка метода	81
3.2. Работа с интерфейсами	87
3.2.1. Обеспечение выполнения контракта	88
3.2.2. Расширение интерфейсов	90
3.2.3. Программирование через интерфейсы	92
Итоги	96
Глава 4. Перечисления и обобщенные типы	98
4.1. Использование <code>enums</code>	98
4.1.1. Численные значения <code>enums</code>	99
4.1.2. Строковые перечисления	102
4.1.3. Использование перечислений <code>const</code>	105
4.2. Использование обобщений	106
4.2.1. Разъяснение обобщений	106
4.2.2. Создание собственных обобщенных типов	112

4.2.3. Создание обобщенных функций.....	116
4.2.4. Обеспечение возвращаемого типа функции высшего порядка ..	121
Итоги.....	123
Глава 5. Декораторы и продвинутые типы	124
5.1. Декораторы.....	125
5.1.1. Создание декораторов классов.....	127
5.1.2. Создание декораторов методов	133
5.2. Отображенные типы	135
5.2.1. Отображенный тип Readonly	135
5.2.2. Объявление собственных отображенных типов	140
5.2.3. Другие встроенные отображенные типы	141
5.3. Условные типы	144
5.3.1. Ключевое слово infer.....	148
Итоги.....	151
Глава 6. Инструменты	152
6.1. Карты кода.....	153
6.2. Линтер TSLint	156
6.3. Связывание кода с помощью Webpack	159
6.3.1. Связывание JavaScript с помощью Webpack	161
6.3.2. Связывание TypeScript с помощью Webpack	166
6.4. Использование компилятора Babel	171
6.4.1. Использование Babel с JavaScript	175
6.4.2. Использование Babel с TypeScript	177
6.4.3. Использование Babel с TypeScript и Webpack	180
6.5. Инструменты для рассмотрения.....	182
6.5.1. Знакомство с DevO.....	182
6.5.2. Знакомство с псс.....	184
Итоги.....	187
Глава 7. Использование TypeScript и JavaScript в одном проекте	188
7.1. Файлы определений типов	188
7.1.1. Знакомство	189

7.1.2. Файлы определений типов и IDE	191
7.1.3. Shim и определения типов	195
7.1.4. Создание собственных файлов определений типов	196
7.2. Пример TypeScript-приложения, использующего JavaScript-библиотеки	197
7.3. Введение TypeScript в JavaScript-проект	207
Итоги	212

Часть 2

Использование TypeScript в блокчейн-приложении

Глава 8. Разработка собственного блокчейн-приложения	214
8.1. Блокчейн 101	215
8.1.1. Криптографические хеш-функции	217
8.1.2. Из чего состоит блок?	220
8.1.3. Что такое добыча блока?	221
8.1.4. Мини-проект с хешем и повсе	224
8.2. Ваш первый блокчейн	226
8.2.1. Структура проекта	227
8.2.2. Создание примитивного блокчейна	231
8.2.3. Создание блокчейна с доказательством прделанной работы	234
Итоги	237
Глава 9. Разработка узла блокчейна на основе браузера	238
9.1. Запуск блокчейн-веб-приложения	239
9.1.1. Структура проекта	239
9.1.2. Развертывание приложения с помощью прт-сценариев	242
9.1.3. Работа с блокчейн-веб-приложением	243
9.2. Веб-клиент	246
9.3. Добыча блоков	251
9.4. Использование сгурто API для генерации хшей	257
9.5. Самостоятельный блокчейн-клиент	260
9.6. Отладка TypeScript в браузере	263
Итоги	265

Глава 10. Клиент-серверное взаимодействие посредством Node.js, TypeScript и WebSocket	266
10.1. Разрешение конфликтов с помощью правила длиннейшей цепочки.....	267
10.2. Добавление сервера в блокчейн	270
10.3. Структура проекта.....	271
10.4. Файлы конфигураций проекта.....	273
10.4.1. Настройка компиляции TypeScript.....	273
10.4.2. Что находится в package.json	275
10.4.3. Настройка nodemon	276
10.4.4. Выполнение блокчейн-приложения.....	277
10.5. Краткое знакомство с WebSockets.....	284
10.5.1. Сравнение протоколов HTTP и WebSocket	285
10.5.2. Передача данных от сервера Node к простому клиенту	286
10.6. Рассмотрение процессов уведомления.....	291
10.6.1. Рассмотрение кода сервера.....	294
10.6.2. Рассмотрение кода клиента.....	304
Итоги.....	315
Глава 11. Разработка приложений Angular с помощью TypeScript	317
11.1. Генерация и запуск нового приложения с помощью Angular CLI.....	319
11.2. Рассмотрение сгенерированного приложения	322
11.3. Сервисы Angular и внедрение зависимостей	328
11.4. Приложение с внедрением ProductService.....	332
11.5. Программирование через абстракции в TypeScript.....	336
11.6. Начало работы с HTTP-запросами	337
11.7. Начало работы с формами.....	342
11.8. Основы маршрутизации.....	346
Итоги.....	351
Глава 12. Разработка клиента блокчейна на Angular	352
12.1. Запуск блокчейн-приложения Angular.....	352
12.2. Обзор AppComponent.....	355
12.3. Рассмотрение Transaction Form Component.....	359

12.4. Обзор Block Component.....	361
12.5. Обзор сервисов.....	363
Итоги.....	366
Глава 13. Разработка приложений React.js с помощью TypeScript.....	367
13.1. Разработка простейшей веб-страницы при помощи React.....	368
13.2. Генерация и запуск нового приложения с помощью Create React App.....	371
13.3. Управление состоянием компонента.....	377
13.3.1. Добавление состояния в компоненты, основанные на классе.....	377
13.3.2. Использование хуков для управления состоянием.....	379
13.4. Разработка метеоприложения.....	382
13.4.1. Добавление хука состояния в компонент App.....	383
13.4.2. Получение данных при помощи хука useEffect в компоненте App.....	387
13.4.3. Использование свойств.....	393
13.4.4. Как дочерний компонент может передавать данные родителю?.....	399
13.5. Что такое виртуальная DOM?.....	402
Итоги.....	404
Глава 14. Разработка блокчейн-клиента в React.js.....	405
14.1. Запуск клиента и сервера обмена сообщениями.....	406
14.2. Что изменилось в директории lib.....	409
14.3. Умный компонент App.....	411
14.3.1. Добавление транзакции.....	413
14.3.2. Генерация нового блока.....	417
14.3.3. Объяснение хуков useEffect().....	417
14.3.4. Мемоизация с помощью хука useCallback().....	419
14.4. Компонент представления TransactionForm.....	423
14.5. Компонент представления PendingTransactionsPanel.....	427
14.6. Компоненты представления BlocksPanel и BlockComponent.....	429
Итоги.....	431

Глава 15. Разработка приложений Vue.js с помощью TypeScript	433
15.1. Разработка простейшей веб-страницы с помощью Vue.....	434
15.2. Генерация и запуск приложения с помощью Vue CLI.....	438
15.3. Разработка одностраничных приложений с маршрутизацией.....	446
15.3.1. Генерация нового приложения с Vue Router.....	447
15.3.2. Отображение списка товаров в представлении Home.....	451
15.3.3. Передача данных с помощью Vue Router.....	458
Итоги.....	463
Глава 16. Разработка блокчейн-клиента на Vue.js	464
16.1. Запуск клиента и сервера обмена сообщениями.....	465
16.2. Компонент App.....	468
16.3. Компонент представления TransactionForm.....	473
16.4. Компонент представления PendingTransactionsPanel.....	478
16.5. Компоненты представления BlocksPanel и Block.....	480
Итоги.....	484
Эпилог.....	485
Приложение А. Современный JavaScript	486
А.1. Как запускать образцы кода.....	486
А.2. Ключевые слова let и const.....	487
А.2.1. Ключевое слово var и поднятие.....	487
А.2.2. let и const для работы в области блока.....	489
А.3. Шаблонные литералы.....	490
А.3.1. Размеченные шаблонные строки.....	491
А.4. Опциональные параметры и значения по умолчанию.....	493
А.5. Выражения стрелочных функций.....	494
А.6. Оператор остатка (rest).....	496
А.7. Оператор распространения.....	498
А.8. Деструктуризация.....	500
А.8.1. Деструктуризация объектов.....	500
А.8.2. Деструктуризация массивов.....	503

A.9. Классы и наследование	504
A.9.1. Конструкторы	506
A.9.2. Ключевое слово <code>super</code> и функция <code>super()</code>	507
A.9.3. Статические члены класса	509
A.10. Асинхронная обработка	511
A.10.1. Ад обратных вызовов	511
A.10.2. Промисы	512
A.10.3. Разрешение нескольких промисов одновременно	515
A.10.4. <code>async-await</code>	516
A.11. Модули	518
A.11.1. Импорты и экспорты	521
A.12. Транспиляторы	523

Объектно-ориентированное программирование с классами и интерфейсами

В этой главе:

- ✓ Принцип работы наследования классов.
- ✓ Где и для чего используются абстрактные классы.
- ✓ Как интерфейсы могут принудить класс иметь методы с известными сигнатурами, не беспокоясь о деталях реализации.
- ✓ Программирование через интерфейсы.

В главе 2 мы познакомились с использованием классов и интерфейсов для создания пользовательских типов. В текущей главе мы продолжим изучение классов и интерфейсов с позиции объектно-ориентированного программирования (ООП). ООП — это стиль программирования, когда ваши программы фокусируются на обработке объектов вместо составления действий (то есть функций). Конечно же, некоторые из этих функций также будут создавать объекты, но в ООП объекты являются центром всего творения.

Разработчики, работающие с объектно-ориентированными языками, используют интерфейсы как способ обусловить классы конкретными API. Помимо этого, в диалогах программистов вы можете часто услышать фразу «программирование через интерфейсы». В этой главе мы объясним, что она означает. Эта глава является быстрым обзором ООП с использованием TypeScript.

3.1. РАБОТА С КЛАССАМИ

Давайте вспомним, что вы узнали о классах TypeScript в главе 2:

- Вы можете объявлять классы со свойствами, которые в других объектно-ориентированных языках называются *переменными-членами*.
- Как и в JavaScript, классы могут объявлять конструкторы, которые вызываются один раз при инстанцировании.
- Компилятор TypeScript преобразует классы в функции-конструкторы JavaScript, если в качестве целевого синтаксиса указан ES5. Если же указана версия ES6 или более поздняя, то классы TypeScript будут перекомпилированы в JavaScript-классы.
- Если конструктор класса определяет аргументы, использующие такие ключевые слова, как `readonly`, `public`, `protected` или `private`, TypeScript создает свойства класса для каждого аргумента.

Однако это еще не все, что касается классов. В текущей главе мы рассмотрим наследование классов, узнаем, для чего нужны абстрактные классы и модификаторы доступа `public`, `protected` и `private`.

3.1.1. Знакомство с наследованием классов

В реальной жизни каждый человек наследует определенные черты от своих родителей. Сходным образом в мире TypeScript вы можете создать новый класс на основе существующего. Например, можно создать класс `Person` с рядом свойств, а затем создать класс `Employee`, который будет *наследовать* все свойства `Person` и при этом объявлять дополнительные. Наследование — это одна из основных особенностей любого объектно-ориентированного языка, в которой слово `extends` объявляет, что один класс наследует от другого.

Рисунок 3.1 — это скриншот песочницы TypeScript (<http://mng.bz/O9Yw>). Обратите внимание, что мы не использовали явные типы в объявлении свойств класса `Person`. Мы инициализировали свойства `firstName` и `lastName` с пустыми строками и `age` с `0`. Компилятор TypeScript сам выведет типы, основываясь на изначальных значениях.

СОВЕТ В меню настроек песочницы TypeScript опция компилятора `strictPropertyInitialization` включена. Это означает, что если свойство класса не инициализировано в конструкторе класса или во время объявления, то компилятор сообщает об ошибке.

Строка 7 на рис. 3.1 показывает, как вы можете объявить класс `Employee`, расширяющий класс `Person` и объявляющий дополнительное свойство `department`. В строке 11 мы создаем экземпляр класса `Employee`.



Рис. 3.1. Наследование классов в TypeScript

Этот скриншот был сделан после того, как на строке 13 мы ввели `empl,` после которой нажали `Ctrl-пробел`. Статический анализатор TypeScript распознает, что тип `Employee` наследован от `Person`, поэтому он предлагает свойства, определенные в обоих этих классах, `Person` и `Employee`.

В нашем примере класс `Employee` является подклассом `Person`, и наоборот, класс `Person` является суперклассом `Employee`. Иначе вы еще можете сказать, что классе `Person` является *предком*, а `Employee` *потомком* `Person`.

ПРИМЕЧАНИЕ Внутри JavaScript поддерживает *объектное* наследование через прототипы, когда один объект может быть присвоен другому в качестве его прототипа, — это происходит при выполнении. Как только TypeScript-код, использующий наследование, скомпилирован, получившийся JavaScript использует синтаксис прототипного наследования.

В дополнение к свойствам класс может включать *методы* — с помощью которых мы вызываем функции, объявленные внутри классов. При этом метод, объявленный в суперклассе, будет унаследован подклассом, если только он не был объявлен с модификатором доступа `private`, который мы рассмотрим несколько позднее.

Следующая версия класса `Person` показана на рис. 3.2 и включает метод `sayHello()`. (Вы можете найти этот код в песочнице <http://mng.bz/YeNz>.) Как вы

можете видеть в строке 18, статический анализатор включил этот метод в меню автоподстановки.

Вам может стать интересно: есть ли способ контролировать, какие свойства и методы класса будут доступны из других сценариев? Да! И именно для этого используются ключевые слова `private`, `protected` и `public`.

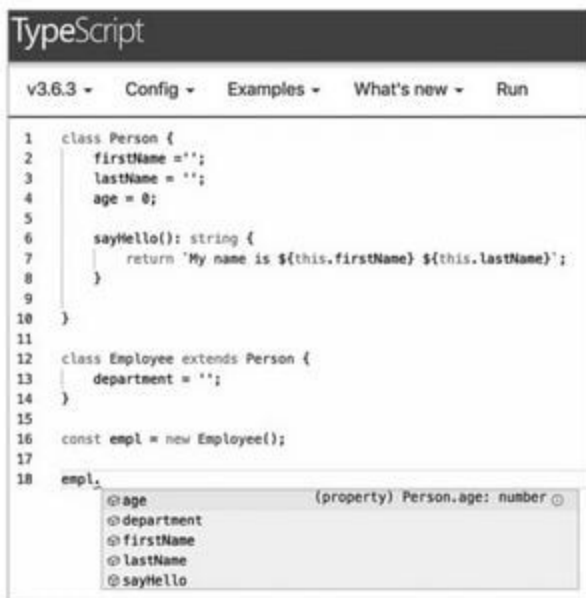


Рис. 3.2. Метод `sayHello()`, принадлежащий суперклассу, видимый

3.1.2. Модификаторы доступа `public`, `private`, `protected`

TypeScript включает ключевые слова `public`, `protected` и `private` для управления доступом к членам класса (свойствам и методам).

- `public` — к членам класса, отмеченным как `public`, можно обратиться как из внутренних методов класса, так и из внешних сценариев. Это уровень доступа по умолчанию, поэтому если вы поместите ключевое слово `public` перед

свойством или методом класса `Person`, приведенном на рис. 3.2, доступность этих членов класса не изменится.

- **protected** — к членам класса, отмеченным как **protected**, можно обратиться либо из внутреннего кода класса, либо из наследников этого класса.
- **private** — члены класса **private** видимы только внутри класса.

ПРИМЕЧАНИЕ Если вы знаете языки вроде Java или C#, то знакомы с ограничением уровня доступа посредством ключевых слов **private** и **protected**. TypeScript же является надмножеством JavaScript, который не поддерживает ключевое слово **private**, поэтому **private**, **protected** (а также **public**) удаляются при компиляции кода. Итоговый JavaScript-код не будет содержать их, поэтому вы можете рассматривать эти ключевые слова просто как помощь при разработке.

На рис. 3.3 показаны модификаторы доступа **protected** и **private**. В строке 15 мы можем обратиться к методу `sayHello()`, принадлежащему **protected**-предку, потому что мы делаем это из его потомка. Но когда мы нажмем **Ctrl-пробел** после **this**, в строке 20, переменная `age` не будет показана в списке автоподстановки, так как объявлена как **private** и доступна только внутри класса `Person`.

Этот образец кода показывает, что подкласс не может обратиться к **private**-члену суперкласса (проверьте это в песочнице <http://mng.bz/07gJ>). Здесь к **private**-членам класса может обратиться только метод из класса `Person`.

Несмотря на то что **protected** члены класса доступны из кода потомка, они недоступны для экземпляра класса. Например, следующий код не скомпилируется и выдаст ошибку «Property 'sayHello' is protected and accessible within class 'Person' and its subclasses». (Свойство `'sayHello'` является **protected** и доступно только внутри класса `'Person'` и его подклассов.)

```
const empl = new Employee(); empl.sayHello(); // ошибка
```

Давайте посмотрим другой пример класса `Person`, имеющий конструктор, два **public**-свойства и одно **private**-свойство, как показано на рис. 3.4 (либо в песочнице <http://mng.bz/KEgX>).

В строке 7 на рис. 3.5 мы создаем экземпляр класса `Person`, передавая изначальные значения свойств его конструктору, который будет присваивать эти значения соответствующим свойствам объекта. На строке 9 мы хотели выводить в консоль значения свойства `firstName` и `age` объекта, но последнее было подчеркнуто красной линией, поскольку `age` является приватным.

Сравните рис. 3.4 и 3.5. На рис. 3.4 класс `Person` явно объявляет три свойства, которые мы инициализируем в конструкторе. На рис. 3.5 класс `Person` не имеет явных объявлений свойств и явных инициализаций в конструкторе.

```

1 class Person {
2   public firstName = '';
3   public lastName = '';
4   private age = 0;
5
6   protected sayHello(): string {
7     return 'My name is ${this.firstName} ${this.lastName}';
8   }
9 }
10
11 class Employee extends Person {
12   department = '';
13
14   reviewPerformance(): void {
15     this.sayHello();
16     this.increasePay(5);
17   }
18
19   increasePay(percent: number): void {
20     this
21   }
22 }
23
24
25

```

@department
 @firstName
 @increasePay
 @lastName
 @reviewPerformance
 @sayHello (method) Person.sayHello(): string

Рис. 3.3. Приватное свойство age невидимо

```

1 class Person {
2   public firstName = '';
3   public lastName = '';
4   private age = 0;
5
6   constructor(firstName: string, lastName: string, age: number) {
7     this.firstName = firstName;
8     this.lastName = lastName;
9     this.age = age;
10  }
11 }

```

Рис. 3.4. Многословная версия класса Person