Оглавление

Часть 2
Использование TypeScript в блокчейн-приложении

3 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 класс `Person` является *предком*, а `Employee` *потомком* `Person`.

> **ПРИМЕЧАНИЕ** Изнутри 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`.

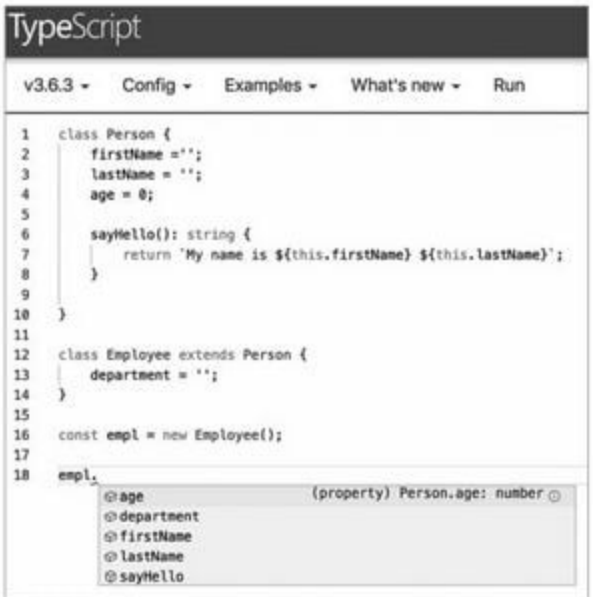

Рис. 3.2. Метод sayHello(), принадлежащий суперклассу, видимый

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.

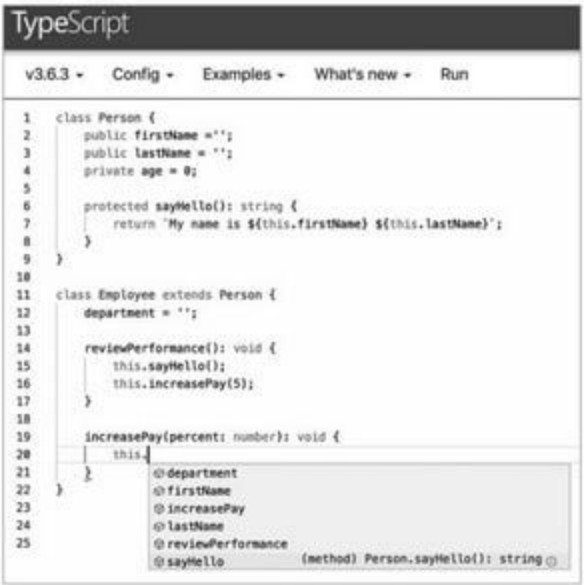

Рис. 3.3. Приватное свойство age невидимо

Рис. 3.4. Многословная версия класса Person