Capítulo 3
Programación orientada a objetos

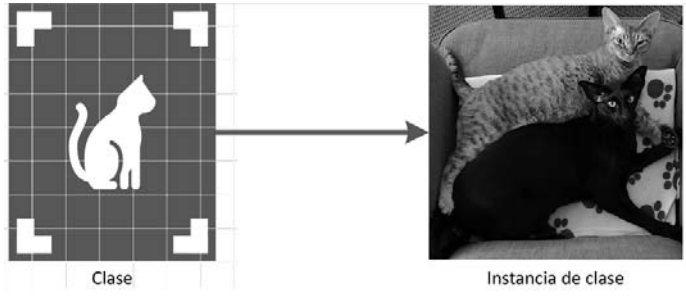
1. Introducción

La programación orientada a objetos (también conocida por las siglas: POO) es uno de los paradigmas de desarrollo más utilizados para la creación de aplicaciones. En los primeros tiempos de la programación, el software se creaba a partir de secuencias de instrucciones que se ejecutaban una tras otra; esto es lo que llamamos programación imperativa. Con el tiempo, los programas informáticos se han vuelto más complejos, lo que ha aumentado considerablemente la dificultad de mantenerlos y hacer que evolucionen. En la década de 1970, *Alan Kay* sentó las bases de la programación orientada a objetos. Este paradigma se hizo popular rápidamente y todavía se utiliza en la actualidad como base de muchos de los programas de software que utilizamos todos los días. Este estilo de programación permite organizar el software usando subconjuntos más pequeños, comúnmente llamados objetos. Cada uno de estos objetos contiene sus propios datos y lógica. Por lo tanto, los objetos son partes autónomas de un programa y tienen interacciones entre sí para que funcione.

TypeScript es uno de los llamados lenguajes *multiparadigma*. Por tanto, permite desarrollar aplicaciones utilizando programación imperativa, orientada a objetos o incluso funcional (consulte el capítulo TypeScript y la programación funcional). Las primeras versiones de TypeScript aportaron muchas capacidades sintácticas en torno a la programación orientada a objetos. Esta es una de las razones por las que el lenguaje se ha vuelto popular entre ciertas comunidades de desarrolladores, en particular C# y Java. Sin embargo, es importante tener en cuenta que, aunque TypeScript es sintácticamente similar a Java y C#, no es un equivalente. En realidad, las capacidades de los objetos de TypeScript siguen el estándar ECMAScript, pero el lenguaje también tiene sus propias particularidades. Estas diferentes capacidades se describirán en detalle en este capítulo.

2. Las clases

El primer concepto importante que se debe aprender al empezar con la programación orientada a objetos es el concepto de clase. Cada objeto debe ser creado por una clase. Esto se puede comparar con las instrucciones de fabricación que contienen toda la información necesaria para la creación de un objeto. Una vez definida, la clase se utiliza en el programa para crear un objeto; esto se denomina «instancia de clase». Puede crear tantos objetos como desee para que un programa funcione.



Posteriormente, los objetos interactuarán entre sí para que el programa funcione.

Para declarar una clase en TypeScript, es necesario usar la palabra clave `class`, darle un nombre y abrir las llaves para definir sus características.

Observación

El concepto de clase no es nuevo en JavaScript. La palabra clave `class` es un azúcar sintáctico basado en la noción de prototipo en el lenguaje (consulte el capítulo Tipos e instrucciones básicas). En ECMAScript 5, es posible definir una clase mediante una función constructora. Esta tiene la particularidad de definir la estructura de los objetos y crearlos. Desde ECMAScript 2015, se prefieren las clases a las funciones constructoras porque estas últimas tienen una sintaxis poco intuitiva.

Sintaxis:

```
class ClassName {
  // ...
}
```

Ejemplo:

```
class Employee {
  // ...
}
```

Observación

La denominación de clases en TypeScript sigue la convención conocida como «PascalCase». Este tipo de denominación especifica que cada palabra que compone el nombre de la clase debe tener su primera letra en mayúscula (ejemplo: `PaySplit`). Las clases y las interfaces son los únicos elementos en TypeScript que utilizan esta convención, lo que las hace fáciles de detectar al leer el código. Para todos los demás elementos, se utiliza la convención de nomenclatura «camelCase». Es similar a la convención «PascalCase», excepto que la primera letra está en minúscula (ejemplo: `firstName`).

Una vez declarada, la clase se puede utilizar para crear nuevas instancias. TypeScript considerará que cada una de estas instancias es del tipo definido por la clase. Para crear una instancia de una clase, se debe usar la palabra clave `new` y asignar la instancia a una variable.

Sintaxis:

```
let/const variable: ClassName = new ClassName();
```

Ejemplo:

```
const employee = new Employee();
```

Si es necesario, se pueden crear varias instancias de la misma clase; cada instancia es completamente independiente de las demás.

Ejemplo:

```
const employee1 = new Employee();
const employee2 = new Employee();

// Log: false
console.log(employee1 === employee2);
```

Con TypeScript, las clases se denominan «*first class citizen*» (objetos de primer orden), por lo que es posible asignarlas en variables.

Sintaxis:

```
let/const ClassName = class {
 // ...
};
```

Cuando una clase está contenida en una variable, se debe usar la palabra clave `new` en ella para crear una instancia de la clase.

Ejemplo:

```
const Employee = class {
  // ...
};

const employee = new Employee();
```

Observación

Esta sintaxis es más particular y, a veces, se utiliza en determinadas implementaciones más complejas. En el resto de este capítulo, los ejemplos de código no se basarán en esta sintaxis.

3. Propiedades

Dado que un objeto debe funcionar de forma autónoma, es necesario que mantenga un estado durante su uso. Este estado está contenido en el objeto en forma de datos. Por tanto, el estado de un programa en programación orientada a objetos corresponderá al conjunto de estados de cada instancia utilizada para hacerlo funcionar.

Para asignar datos a un objeto en TypeScript, se debe utilizar una propiedad. Estas se definen a nivel de clase y deben tiparse. Para declarar una propiedad, es suficiente con agregarla entre las llaves de la clase, dándole un nombre y luego especificando su tipo.

Sintaxis:

```
class ClassName {
  propertyName: type;
}
```

Ejemplo:

```
class Employee {
  firstName: string;
  lastName: string;
}
```

Una vez creada la instancia de una clase, es posible asignar valores a las diferentes propiedades, pero también recuperarlos. Las propiedades están disponibles utilizando un «.» después del nombre de la variable que contiene el objeto.

Ejemplo:

```
let employee = new Employee();
employee.firstName = "Evelyn";
employee.lastName = "Miller";

// Log: Evelyn


console.log(employee.firstName);

// Log: Miller
console.log(employee.lastName);
```

```
employee = new Employee();

// Log: undefined
console.log(employee.firstName);

// Log: undefined
console.log(employee.lastName);
```

Observación

De forma predeterminada, las propiedades se inicializarán con el valor undefined.

Este primer ejemplo plantea un problema de compilación cuando se inicia un proyecto de TypeScript con la configuración básica del compilador (obtenida ejecutando el comando `tsc --init`). De forma predeterminada, el compilador inicializa el proyecto con la opción `--strict` habilitada en el archivo `tsconfig.json`. Esta activa subopciones, incluida la opción `--strictPropertyInitialization`. Este última genera un error cuando los valores de las propiedades no se inicializan al crear una instancia de una clase. En el ejemplo anterior, las propiedades no se inicializan cuando se usa la palabra clave new, por lo que contienen el valor `undefinided`. Esta situación podría deberse a un error por descuido, razón por la cual TypeScript no lo permite de forma predeterminada.

Es posible forzar al compilador a que no informe este error utilizando el símbolo «!» al final del nombre de una propiedad (también llamado *definite assignment assertion operator*).

Ejemplo:

```
class Employee {
  firstName!: string;
  lastName!: string;
}

const employee = new Employee();

// Log: undefined
console.log(employee.firstName);

// Log: undefined
console.log(employee.lastName);
```

Observación

Tenga cuidado de no forzar el compilador sistemáticamente. La mayoría de los errores reportados por el modo estricto de TypeScript son importantes. Forzar la inicialización de propiedades permite, por ejemplo, no obtener error si se utilizan sin valor. Este es un error común en el desarrollo web, especialmente cuando se supone que la propiedad contiene una función (lo que desencadena el error: undefined is not a function).

Asignar un valor a las propiedades también le permite evitar el error de compilación vinculado al modo strict.

Ejemplo:

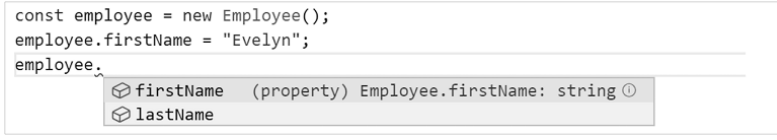
```
class Employee {
  firstName: string = "Evelyn";
  lastName: string = "Miller";
}

const employee = new Employee();

// Log: Evelyn
console.log(employee.firstName);

// Log: Miller
console.log(employee.lastName);
```

Observación

En el resto de este capítulo, la opción `--strict` siempre se considerará activa en los ejemplos de código.

Una vez definidas, las propiedades se pueden encontrar en el entorno de desarrollo mediante el autocompletado.

Ejemplo (Visual Studio Code):



4. Métodos

Los datos contenidos en un objeto se pueden utilizar posteriormente durante la ejecución de reglas lógicas dentro del programa. Es posible escribir estas reglas de forma imperativa o utilizando una función.

Por su parte, la programación orientada a objetos propone definir esta lógica directamente en clases usando métodos. Para definir un método en una clase, es suficiente con agregar una función dentro de ella.

Sintaxis:

```
class ClassName {
  methodName(param1: type, param2: type, ...): type {
    // ...
  }
}
```

Dentro del método, será posible hacer referencia a la instancia actual de la clase usando palabra clave `this`. Esto le permite manipular las propiedades de un objeto o utilizar otro método.

Ejemplo:

```
class Employee {
  firstName!: string;
  lastName!: string;

  getFullName(): string {
    return `${this.firstName} ${this.lastName}`;
  }
}
const employee = new Employee();
employee.firstName = "Evelyn";
employee.lastName = "Miller";

const fullName = employee.getFullName();

// Log: Evelyn Miller
console.log(fullName);
```

Observación

La palabra clave `this` ya se ha analizado anteriormente (consulte el capítulo Tipos e instrucciones básicas). Aunque su uso es diferente en la programación orientada a objetos, las reglas básicas relacionadas con el alcance de `this` se aplican siempre. Cuidado al usarlo en una función devuelta por un método. Siempre es necesario utilizar una función `bind` o flecha para aplicar el alcance correcto y no causar un error al ejecutar la función.

Al igual que las funciones, los métodos pueden tener parámetros los cuales cumplen todas las reglas relativas a las funciones vistas anteriormente (consulte el capítulo Tipos e instrucciones básicas - Funciones).

Ejemplo:

```
class Employee {
  firstName!: string;
  lastName!: string;
  salary!: number;

  increaseSalary(percent: number): void {
    if (percent >= 0 && percent <= 100) {
      const amount = this.salary * (percent / 100);
      this.salary += amount;
    }
  }
}

const employee = new Employee();
employee.firstName = "Evelyn";
employee.lastName = "Miller";
employee.salary = 2000;

employee.increaseSalary(5);

// Log: 2100
console.log(employee.salary);
```

Los parámetros también se pueden declarar opcionalmente añadiendo el carácter «**?**» después del nombre del parámetro (consulte el capítulo Tipos e instrucciones básicas - Funciones).