Capítulo 3
Programación orientada a objetos

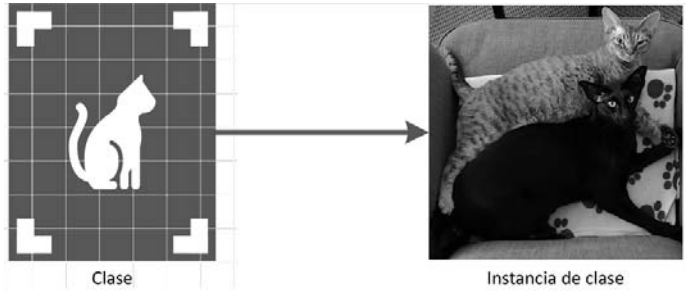
1. Introducción

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:

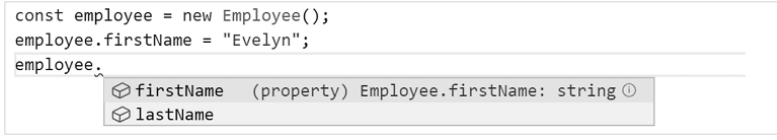
```
class Employee {
  firstName: string = "Evelyn";
  lastName: string = "Miller";
}

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).

Capítulo 3
Descubrir el JSX

1. Introducción a JSX

JSX (*JavaScript XML*) es una extensión de sintaxis utilizada en React para describir la interfaz de usuario en forma de código JavaScript. Permite mezclar código JavaScript con etiquetas HTML, lo que hace que la creación de interfaces de usuario en React sea más legible. Es importante sentirse cómodo con esta sintaxis lo antes posible.

Cuando utilizas JSX, puedes escribir elementos de React como si estuvieras escribiendo HTML, pero en realidad, el JSX se transpila en código JavaScript puro antes de ser interpretado por el navegador.

He aquí un ejemplo sencillo de como se ve JSX:

```
import React from 'react';
const MiComponente = () => {
 return <h1>¡Bienvenido a mi aplicación React!</h1>;
};
```

En este ejemplo, hemos utilizado JSX para crear un elemento h1 que contiene el mensaje de bienvenida. Esto permite escribir código de interfaz de usuario de forma declarativa y sin llamar a `React.createElement` cada vez.

Ventajas de JSX

Claridad y legibilidad del código

Al utilizar el JSX nos acercamos al lenguaje natural utilizado para estructurar una página web. El JSX facilita la comunicación entre desarrolladores y mejora la mantenibilidad del código, porque es más fácil entender la estructura de la interfaz de usuario.

Composición de componentes facilitada

Con el JSX se pueden anidar fácilmente componentes unos dentro de otros. Podemos almacenarlos en variables, pasarlos a funciones o incluso añadir código JavaScript en su interior. Esto permite crear componentes complejos formándolos a partir de componentes más pequeños y reutilizables. Componer componentes es uno de los principios fundamentales de React, y el JSX facilita este enfoque.

Integración de JavaScript dinámico

El JSX permite incorporar expresiones JavaScript directamente en el código de tipo HTML. Esto le permite interactuar dinámicamente con los datos y generar contenidos de interfaz de usuario basados en el estado de la aplicación. Puede utilizar expresiones JavaScript dentro de llaves `{ }` para evaluar variables, realizar operaciones y mucho más.

Validación estática

Una de las ventajas de JSX es que se comprueba estáticamente durante la compilación. Esto significa que los errores de sintaxis o estructura dentro de JSX se detectan de antemano, antes de que se ejecute la aplicación. Esto permite evitar muchos de los errores que podrían producirse durante la ejecución, haciendo que el proceso de desarrollo sea más seguro y fluido.

Integración fácil de bibliotecas de terceros

Como JSX se parece a HTML, es más fácil integrar bibliotecas de componentes de terceros. Por ejemplo, los desarrolladores de React pueden utilizar bibliotecas de componentes de interfaz de usuario, herramientas de gestión de formularios y otras bibliotecas populares simplemente como lo harían dentro de un proyecto HTML estándar.

Transpilación a JavaScript estándar

El JSX debe transpilarse a JavaScript estándar antes de ser ejecutado por los navegadores. Esto permite a los desarrolladores de utilizar las funcionalidades más reciente de React todo manteniendo la compatibilidad con navegadores antiguos. Las herramientas como Babel se han utilizado durante mucho tiempo para transpilar el JSX a JavaScript estándar, aunque han surgido alternativas. No es necesario entender cómo funcionan las herramientas de transpilación, pero es importante recordar que JSX no es un estándar web y que tiene que pasar por una etapa antes de ser entendido por los navegadores.

Así es como se transpila JSX a JavaScript:

```
const mensaje = <h1>¡Bienvenido a mi aplicación React!</h1>;
```

Código JavaScript transpilado:

```
const message = React.createElement("h1", null, "¡Bienvenido a
mi aplicación React!");
```

El JSX se transforma en llamadas a la función `React.createElement`, que crea elementos React utilizando las propiedades pasadas como argumentos.

Con una herramienta como Vite, que vimos anteriormente, no es necesario de configurar nada. JSX se transpilará automáticamente. Del mismo modo, es posible utilizar las funcionalidades modernas de JavaScript sin preocuparse por la compatibilidad del navegador. Las funciones flecha, por ejemplo, pueden convertirse en funciones estándar si su navegador de destino no las soporta. Consulte la documentación de Vite si necesita garantizar la compatibilidad con navegadores antiguos como Internet Explorer.

2. Sintaxis y elementos JSX

En JSX, puedes utilizar una sintaxis similar a HTML para describir la interfaz de usuario de tu aplicación React. Sin embargo, hay algunas diferencias clave a tener en cuenta.

2.1 Elementos JSX

En JSX, usted puede utilizar etiquetas para crear elementos React. Las etiquetas deben corresponder a componentes React definidos o a etiquetas HTML nativas. Los elementos JSX pueden ser anidados unos dentro de otros, al igual que en HTML.

Ejemplo de etiquetas JSX

```
const miElemento = <div>
   <h1>Título</h1>
   <p>Contenido del párrafo</p>
</div>;
```

Los elementos JSX son los bloques de construcción básicos para crear la interfaz de usuario. Parecen etiquetas HTML, pero en realidad son objetos JavaScript que representan componentes React o elementos DOM.

La sintaxis es similar a la de HTML, lo que las hace más familiares para los desarrolladores web.

Ejemplos de elementos JSX

```
const element1 = <div>Contenido div</div>;
const element2 = <h1>Título</h1>;
const element3 = <MiComponente />;
```

En este ejemplo, la variable `element1` contiene un elemento JSX que representa una etiqueta `div`, `element2` contiene una etiqueta `h1` y `element3` contiene un componente React llamado `MiComponente`.

También puede utilizar atributos en las etiquetas JSX, igual que en HTML.

2.2 Atributos JSX

En JSX, usted puede utilizar atributos para configurar los elementos de la interfaz de usuario de manera similar a HTML. Los atributos permiten personalizar el comportamiento y la apariencia de los elementos React. A continuación le explicamos cómo puede utilizar los atributos JSX en sus componentes:

Ejemplo de atributos JSX

```
const miElemento = <input type="text" placeholder="Introduce tu nombre" />;
const nombre = "Juan";
const miElemento2 = <h1>Bienvenido, {nombre} !</h1>;
```

En este ejemplo, `miElemento` es un elemento JSX que representa una etiqueta input con atributos `type` y `placeholder`. El valor del atributo type es una cadena de caracteres `"text"`, mientras que el valor del atributo placeholder es otra cadena de caracteres `"Introduce tu nombre"`.

Por último, `miElemento2` es un elemento JSX que representa una etiqueta `h1` que utiliza una expresión JavaScript (`{nombre}`) para mostrar el nombre dinámico del usuario en función de la variable `nombre`.

2.2.1 Atributos booleanos

Los atributos booleanos en JSX son ligeramente diferentes a los de HTML. En JSX, puede especificar atributos booleanos sin valor, lo que los convierte true por defecto.

Ejemplo de atributo booleano JSX

```
const miElemento = <input type="checkbox" checked />;
```

En este ejemplo, el atributo `checked` se especifica sin valor, lo que significa que se evalúa como `checked={true}`. Esto marca la casilla por defecto.

2.2.2 Atributos personalizados

Los atributos personalizados (*data attributes*) son una función de HTML que permite almacenar información adicional directamente en una etiqueta, utilizando una sintaxis que suele empezar por `data-`. Se puede acceder a estos atributos a través de JavaScript.

Por ejemplo, considere un elemento de lista HTML que represente un producto en una tienda online:

```
<li data-producto-id="123" data-producto-categoría="electrónica">
Tabletas gráficas</li>
```

Dentro de su script JavaScript, usted podría luego acceder a esta información de la siguiente manera:

```
let produit = document.querySelector('li');
console.log(produit.dataset.productId); // Muestra: 123
console.log(produit.dataset.productCategory); // Muestra: electrónica
```

En el contexto de React, el uso de data attributes es menos común, principalmente porque React favorece la gestión de datos a través del estado (*state*) y las propiedades (*props*) de los componentes, que descubriremos en breve. Estas últimas ofrecen un mecanismo más estructurado e integrado para manipular los datos dentro de sus componentes.

Sin embargo, los data attributes todavía pueden encontrar su lugar en una aplicación React, especialmente cuando se trata de integrar scripts de terceros o manipular elementos DOM directamente. Por lo general, estos escenarios se evitan, pero pueden ser necesarios en determinados casos.

Por ejemplo, es posible que desee rastrear información adicional para herramientas de análisis o seguimiento, donde un data attribute podría ser útil. Si volvemos a nuestro ejemplo de un producto en una lista:

```
function Producto({ id, categoria, nombre }) {
  return (
    <li data-product-id={id} data-product-category={categoria}}>
      {nom}
    </li>
    );
}
// Uso del componente Producto con atributos de datos
<Producto id="123" categoria="electronica" nombre="Tableta grafica" />
```

En este fragmento de código, definimos un componente Product que recibe algunos props y utiliza data attributes para añadir información adicional al elemento li. Aunque esta no es una práctica común en las aplicaciones React, puede ser útil en situaciones específicas donde la interacción directa con el DOM es necesaria o preferible. El concepto de props se introducirá en detalle en el capítulo Fundamentos de React.

2.3 Elementos y fragmentos de React

Un elemento React es un objeto JavaScript que representa un componente o elemento JSX. Tiene tres propiedades: el tipo (nombre de la etiqueta o componente), los atributos y los hijos (los elementos o texto que hay dentro).

JSX fomenta el retorno de un único elemento por componente. Esto significa que si desea devolver varios elementos, deben estar envueltos en un único elemento padre.

Los fragmentos permiten englobar varios elementos sin añadir un nodo DOM adicional. Son útiles cuando necesitas devolver varios elementos sin encapsularlos dentro de una etiqueta padre.

Ejemplo de utilización de fragmentos

```
import React from 'react';

const MiComponente = () => {
  return (
    <>
      <h1>Título</h1>
      <p>Contenido del párrafo</p>
    </>
  );
```

Volveremos a los fragmentos en detalle más adelante en este capítulo, en la sección Los fragmentos JSX.

2.4 Comentarios JSX

Los comentarios desempeñan un papel esencial en el desarrollo: permiten a los desarrolladores añadir notas, explicaciones y recordatorios al código. En JSX, también puedes incluir comentarios para documentar el código y facilitar la comprensión y el mantenimiento del desarrollo.

Para añadir comentarios a JSX, puede encerrarlos entre `{/* */}`.

Ejemplo de comentario JSX

```
const miElemento = (
  <div>
    {/* Esto es un comentario */}
    <h1>Título</h1>
  </div>
);
```

Sintaxis de comentarios JSX

JSX, al ser una extensión de sintaxis para JavaScript, tiene una forma distinta de integrar comentarios en el código.

De hecho, para insertar comentarios en JSX, utilizamos una sintaxis similar a la utilizada para los bloques de comentarios en JavaScript. Se trata de encapsular los comentarios entre `{/*y */}`. Este método le permite integrar los comentarios directamente en su código JSX, garantizando que sus notas y aclaraciones permanezcan visibles sin afectar al funcionamiento de su código.

Ejemplo de comentario en línea en JSX

```
function App() {
  return (
    <div>
      {/* Esto es un comentario incrustado en JSX, no será
visible en el DOM */}
      ¡Bienvenido a mi aplicación React!
    </div>
  );
}
```

En este ejemplo, el comentario se coloca entre `{/*` y `*/}`, haciéndolo visible únicamente en el código fuente, sin ningún impacto en la renderización final dentro del navegador.

2.5 Uso de JavaScript dentro de JSX

Para incorporar expresiones JavaScript dentro de JSX, puede utilizar llaves `{ }`.Esto le permite interactuar dinámicamente con los datos y generar elementos de la interfaz de usuario de manera condicional.

Ejemplo de uso de JavaScript en JSX

```
const usuario = {
 nombre: "Juan",
 edad: 30,
};
const miElemento = <h1>¡Bienvenido, { usuario.nombre} ! Tienes
{usuario.edad} años.</h1>;
```

JSX ofrece una sintaxis familiar y expresiva para crear elementos de interfaz de usuario en React. Puedes utilizar etiquetas, atributos e incluso expresiones JavaScript para generar dinámicamente el contenido de tu aplicación.

3. Construir una interfaz con componentes

Dominar React implica el arte de ensamblar componentes reutilizables para crear componentes más grandes y complejos, con el fin de orquestar una aplicación completa. Este método se destaca por su sencillez y eficiencia, promoviendo un código limpio y mantenible.

Es imposible tener una regla precisa sobre el tamaño que deben tener los componentes. Lo que hay que tener en cuenta es que un componente debe ser responsable de una tarea. Una vez más, esta regla difiere de un proyecto a otro.