Parte 3
Aproximación a la POO en JavaScript

Capítulo 3-1
Enfoque orientado a "objetos" en JavaScript

1. Introducción

Aunque la implementación del modelo de programación orientada a objetos (POO) no esté tan completa en JavaScript como en C++ o Java, JavaScript ofrece los mecanismos principales gestionados por estos lenguajes.

Recordemos los conceptos más importantes de la POO:

- Encapsulación: reunión de un conjunto de propiedades (parte de tratamiento de datos) y de funciones, también llamadas métodos (parte de procesamientos), dentro de un objeto tipo (quizás es más correcto hablar de clase), con la posibilidad de crear (instanciar) objetos a partir de esta clase.
- Herencia: posibilidad de "fabricar" una nueva clase a partir de una clase existente; esta nueva clase hereda las propiedades y métodos de la clase padre (se pueden añadir nuevas propiedades/métodos a la nueva clase).
- Polimorfismo: un método del mismo nombre asociado a varias clases puede tener comportamientos diferentes para algunas de estas clases.

2. Programación orientada a objetos a través de ejemplos

JavaScript siempre ha sido una pieza esencial en los desarrollos web, principalmente para la programación del lado "cliente", es decir, del lado del navegador. Habitualmente, sin sumergirse de lleno en el lenguaje, los desarrolladores producen código JavaScript de calidad mediocre, contentándose con adaptar el código fuente recuperado de sitios web y manipulando los conceptos POO lo menos posible.

En paralelo, han aparecido un gran número de librerías JavaScript y su uso permite producir aplicaciones de mejor calidad. El dominio de estas librerías supone tener conocimientos básicos de POO en JavaScript.

Por tanto, el objetivo de la exposición que sigue es presentarle lo que hay que saber sobre este asunto. Los conceptos se van a explicar a través de una serie de ejemplos.

Algunos lectores que ya tengan una importante experiencia en otros lenguajes POO (PHP 5, Java, C++...) al principio se pueden sentir incómodos con los aspectos específicos de la POO en JavaScript, la POO por prototipado.

2.1 Secuencia 1: Declaración de los objetos JavaScript de manera "Inline"

Se trata de la manera más sencilla de declarar un objeto en JavaScript.

```
/* Declaración inline de un objeto JavaScript */
/* NB: Esta técnica no permite la herencia a partir del objeto
más adelante */
var Adicion = {
    x: 5,
    y: 10,
    calculo: function()
    {
        return this.x + this.y;
    }
};

/* Uso del objeto Adicion */
document.write("Suma: " + Adicion.calculo());
```

El resultado obtenido de la ejecución será el siguiente:

```
Suma = 15
```

En este tipo de declaración de objeto, es posible prever la especificación de atributos (propiedades) y también de métodos.

La palabra clave `this` sirve para indicar que se está haciendo referencia a los atributos del objeto en sí mismo.

Está claro que con este tipo de declaración no será posible reutilizar este tipo de definición para crear un objeto de las mismas características (o parecidas). Por tanto, este método se utilizará poco (o nada) porque no permite la herencia. No se preocupe, porque volveremos sobre esto más en detalle, a lo largo de esta exposición.

© Editions ENI - All rights reserved

2.2 Secuencia 2: Creación de objetos JavaScript con un constructor

También es posible crear nuestros objetos JavaScript con un constructor (concepto bien conocido en los lenguajes POO). En JavaScript, será suficiente con escribir una función y llamarla posteriormente con la palabra clave new. Por tanto, la función jugará el papel de clase sin serlo realmente.

Veamos con un ejemplo el desarrollo que es necesario seguir:

```
/* Definición de una función constructor, de nombre Coche */
var Coche = function()
{
    /* Atributo(s) del objeto */
    this.tieneMotor = true;
    /* Método(s) del objeto */
    this.avanzar = function()
    {
        document.write("avanza");
    }
}

/* Instanciación de un objeto simca1100 a través del constructor Coche */
var simca1100 = new Coche();

/* Visualización del atributo tieneMotor del objeto simca1100 */
if (simca1100.tieneMotor)
{
    document.write("El coche simca1100 tiene un motor<br />");
}
else
{
    document.write("El coche simca1100 no tiene motor !<br />");
}

/* Llamada al método avanzar del objeto simca1100 */
document.write("El coche simca1100 ");
simca1100.avanzar();
```

En nuestro ejemplo, se define en primer lugar una función `Coche`. Integra un atributo booleano que indica que los coches tienen un motor y un método (función) de nombre `avanzar`, que mostrará "avanza" cuando se pida, a partir de un objeto de tipo `Coche` (se entenderá rápidamente).

Posteriormente, se construye un objeto de nombre `simca1100` a partir del constructor `Coche` (siento que el término "constructor" pueda ser confuso en un ejemplo basado en coches):

```
/* Instanciación de un objeto simca1100 a través del constructor Coche */
var simca1100 = new Coche();
```

Ahora, la propiedad (atributo) `tieneMotor` se puede consultar para el objeto `simca1100` instanciado y también se puede ejecutar el método `avanzar`. Cuando se ejecute, tendremos:

```
El coche simca1100 tiene un motor
El coche simca1100 avanza
```

2.3 Secuencia 3: Variables privadas en una instancia de objeto

En el ejemplo de la secuencia anterior, habrá observado que las propiedades (atributos) llevan como prefijo la palabra clave `this`. Esto es lo que las hace usables desde el exterior (caso de la propiedad `tieneMotor`). Por el contrario, por necesidades locales del constructor (cálculo interno), puede declarar variables no expuestas, usando como prefijo la palabra clave `var`.

Veamos un ejemplo concreto:

```
/* Definición de una función constructor de nombre Coche */
var Coche = function()
{
    /* Variable(s) local(s) no accesible(s) desde el exterior del objeto */
    var numeroRuedas = 4;
    /* Método(s) del objeto */
    this.avanzar = function()
    {
    document.write("avanza");
    }
}

/* Instanciación de un objeto simca1100 a través del constructor Coche */
var simca1100 = new Coche();

/* Llamada al método avanzar del objeto simca1100 */
document.write("El coche simca1100 ");
simca1100.avanzar();

/* Intento de visualización de la variable local del constructor Coche */
document.write("<br />");
document.write("El coche simca1100 tiene " + simca1100.numeroRuedas +
" ruedas");
```

Se usa una variable local `numeroRuedas` en el constructor con el prefijo `var`. Esto solo es accesible, como estaba previsto, desde dentro del constructor, como se muestra en la ejecución de este script:

```
El coche simca1100 avanza
El coche simca1100 tiene undefined ruedas
```

© Editions ENI - All rights reserved

2.4 Secuencia 4: Paso de argumento(s) a un constructor

En el ejemplo siguiente, vamos ver que es posible pasar uno o varios argumentos (lo habíamos visto ya para las funciones clásicas) a un constructor:

```
/* Definición de una función constructor de nombre Coche */
var Coche = function(modelo)
{
    /* Atributo(s) del objeto informado durante la instanciación */
    this.modelo = modelo;
}

/* Instanciación de un objeto simca1100 a través del constructor Coche
con paso de argumento */
var miCoche = new Coche("simca1100");

/* Visualización del atributo modelo del objeto miCoche */
document.write("Tengo un " + miCoche.modelo);
```

El argumento `modelo` está en los paréntesis que siguen al nombre del constructor:

```
var Coche = function(modelo)
```

y está disponible en el cuerpo del constructor por:

```
    this.modelo = modelo;
```

A continuación, es suficiente a nivel de la instanciación del objeto `miCoche` con pasar como argumento un valor ("`simca1100`" en nuestro caso):

```
var miCoche = new Coche("simca1100");
```

La propiedad (atributo) `modelo` del objeto se mostrará por:

```
document.write("Tengo un " + miCoche.modelo);
```

2.5 Secuencia 5: No compartición de los métodos por las instancias de objetos

Dado que los métodos se declaran durante la instanciación de los objetos, sus definiciones están duplicadas en memoria.

En un ejemplo pequeño, el impacto es bajo.

Por el contrario, si su aplicación manipula muchos objetos con métodos múltiples y complejos en los constructores, esto se convierte en inmanejable.

El siguiente ejemplo destaca el problema que acabamos de comentar:

```
/* Definición de una función constructor de nombre Coche */
var Coche = function()
{
    /* Atributo(s) del objeto */
    this.tieneMotor = true;
    /* Método(s) del objeto */
    this.avanzar = function()
    {
        document.write("avanza");
    }
    this.retroceder = function()
    {
        document.write("retrocede");
    }
}

/* Instanciación de un objeto simca1100 a través del constructor Coche */
var simca1100 = new Coche();

/* Instanciación de un objeto renault12 a través del constructor Coche */
var renault12 = new Coche();

/* Comprobación de la igualdad de métodos avanzar de los objetos simca1100
y renault12 */
if (simca1100.avanzar == renault.avanzar)
{
   document.write("Método avanzar compartido por los objetos simca1100 y
Renault12<br />");
}
else
{
    document.write("Método avanzar no compartido por los objetos simca1100
y Renault12<br />");
}
```

La ejecución confirma que el método `avanzar` no está factorizado:

```
Método avanzar no compartido por los objetos simca1100 y Renault12
```

La noción de prototipo que vamos a descubrir a continuación va a resolver este problema.

© Editions ENI - All rights reserved

2.6 Secuencia 6: Noción de prototipo

Un prototipo es un conjunto de elementos (atributos/propiedades y métodos) que se va a asociar a un constructor (sin "almacenamiento" en el constructor en sí mismo). Durante la ejecución, cuando una propiedad de objeto solicitada en el código no se encuentre en el constructor del objeto en cuestión, se realizará una búsqueda en esta lista "adicional".

Veamos un ejemplo completo:

```
/* Definición de una función constructor de nombre Coche */
/* NB: El constructor aquí está vacío */
var Coche = function() {};

/* Comprobación de la existencia por defecto de un prototipo para
cualquier constructor */
document.write("Prototipo del constructor: " + Coche.prototype);

/* Añadir un método zigzaguear al prototipo del constructor Coche */
Coche.prototype.zigzaguear = function()
{
    document.write("zigzaguea peligrosamente<br />");
};

/* Instanciación de un objeto simca1100 a través del constructor Coche */
var simca1100 = new Coche();

/* Llamada al método zigzaguear del objeto simca1100, accesible a través
del prototipo del constructor Coche */
document.write("<br />¿Qué hace el simca1100? ");
simca1100.zigzaguear();

/* Instanciación de un objeto renault12 a través del constructor Coche */
var renault12 = new Coche();

/* Comprobación del método zigzaguear compartido o no, entre los objetos
simca1100 y renault12 */
if (simca1100.zigzaguear == renault12.zigzaguear)
{
    document.write("Método zigzaguear compartido por los objetos
simca1100 y renault12<br />");
}
else
{
    document.write("Método zigzaguear no compartido por los objetos
simca1100 y renault12<br />");
}
```

La opción que se ha utilizado en el ejemplo es codificar un constructor vacío. Después, se realiza una comprobación para demostrar que cualquier constructor tiene un prototipo:

```
document.write("Prototipo del constructor: " + Coche.prototype);
```

Posteriormente, se ha asociado un método `zigzaguear` al prototipo relacionado con el constructor `Coche`:

```
Coche.prototype.zigzaguear = function()
{
    document.write("zigzaguea peligrosamente<br />");
};
```

Después de la instanciación habitual de un objeto `simca1100` a partir del constructor `Coche`, se hace una llamada al método `zigzaguear` para el objeto `simca1100`.

También se instancia un segundo objeto, `renault12`, a partir del constructor `Coche` para demostrar que esta vez el método `zigzaguear` está compartido (es común a ambos objetos).

La ejecución da:

```
Prototipo del constructor: [object Object]
¿Qué hace el simca1100? Zigzaguea peligrosamente
Método zigzaguear compartido por los objetos simca1100 y renault12
```

2.7 Secuencia 7: Sobrecarga de un método

Para una instancia dada de objeto, es posible sobrecargar (modificar) un método (o una propiedad/atributo).

Veamos un ejemplo concreto de cómo aplicarlo:

```
/* Definición de una función constructor, de nombre Coche */
/* NB: El constructor aquí está vacío */
var Coche = function() {};

/* Añadir un método cargar al prototipo del constructor Coche */
Coche.prototype.cargar = function()
{
    document.write("carga<br />");
};

/* Instanciación de un objeto simca1100 a través del constructor Coche */
var simca1100 = new Coche();

/* Llamada al método cargar del objeto simca1100, accesible a través
del prototipo del constructor Coche */
document.write("El simca1100 ");
```

© Editions ENI - All rights reserved

```
simca1100.cargar();

/* Instanciación de un objeto renault12 a través del constructor Coche */
var renault12 = new Coche();

/* Modificación (sobrecarga) del método cargar para el objeto
simca1100 */
simca1100.cargar = function()
{
    document.write("carga rápidamente <br />");
};

/* Llamada al método cargar (sobrecargado) del objeto simca1100 */
document.write("El simca1100 ");
simca1100.cargar();

/* Llamada al método cargar (no sobrecargado) del objeto renault12 */
document.write("El renault12 ");
renault12.cargar();
```

La ejecución da como resultado:

```
El simca1100 carga
El simca1100 carga rápidamente
El renault12 carga
```

Puede comprobar que la modificación (sobrecarga) solo impacta al objeto `simca1100`.

2.8 Secuencia 8: Extensión de un prototipo

La extensión es la suma posterior de un método adicional a nivel de un prototipo de constructor. Veamos un ejemplo de extensión:

```
/* Definición de una función constructor de nombre Coche */
/* NB: El constructor aquí está vacío */
var Coche = function() {};

/* Añadir un método acelerar al prototipo del constructor Coche */
Coche.prototype.acelerar = function()
{
    document.write("acelera<br />");
};

/* Instanciación de un objeto simca1100 a través del constructor Coche */
var simca1100 = new Coche();

/* Llamada al método acelerar del objeto simca1100, accesible a través
del prototipo del constructor Coche */
```

Capítulo 3
Programación orientada a objetos

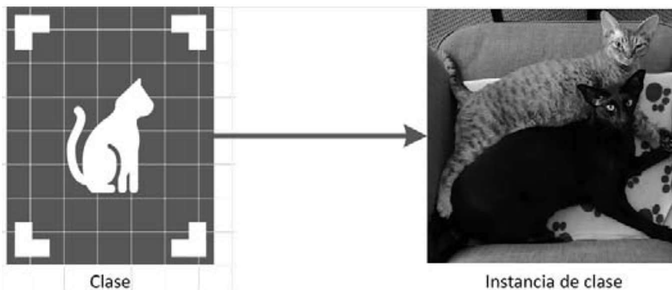
1. Introducción

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.

© Editions ENI - All rights reserved

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.

© Editions ENI - All rights reserved

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);
```

© Editions ENI - All rights reserved

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):

```
const employee = new Employee();
employee.firstName = "Evelyn";
employee.
        firstName   (property) Employee.firstName: string
        lastName
```

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);
```

© Editions ENI - All rights reserved

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).