7
Clases

En este capítulo, vamos a discutir las clases de JavaScript. Ya hemos visto objetos de JavaScript, y las clases son un modelo o plantilla para la creación de objetos. Por lo tanto, muchas de las cosas discutidas aquí no deberían sonar demasiado desconocidas o revolucionarias.

Las clases permiten la programación orientada a objetos, que fue uno de los avances de diseño más importantes en el desarrollo de software. Este desarrollo redujo la complejidad de las aplicaciones y aumentó la capacidad de mantenimiento por un margen enorme.

Por lo tanto, la programación y las clases orientadas a objetos son de gran importancia para la informática en general. Sin embargo, este no es necesariamente el caso cuando lo aplicamos a JavaScript. Las clases de JavaScript son algo especial en comparación con otros lenguajes de programación. Debajo de la superficie, las clases están envueltas en algún tipo de función especial. Esto significa que en realidad son una sintaxis alternativa para definir objetos usando una función constructora. En este capítulo, aprenderemos qué son las clases y cómo podemos crearlas y usarlas. En el camino, cubriremos los siguientes temas:

- Programación orientada a objetos
- Clases y objetos
- Clases
- Herencia
- Prototipos

Nota: las respuestas de los ejercicios, proyectos y cuestionarios de autoevaluación se pueden encontrar en el *Apéndice*.

Programación orientada a objetos

Antes de comenzar a sumergirnos en la diversión de las clases, digamos brevemente algo sobre **la programación orientada a objetos (POO).** OOP es un paradigma de programación muy importante en el que el código está estructurado en objetos, lo que lleva a un código más fácil de mantener y reutilizable. Trabajar con programación orientada a objetos te enseña a tratar realmente de pensar en todo tipo de temas en los objetos, agrupando propiedades de tal manera que puedan envolverse en un modelo llamado clase. Esto, a su vez, podría heredar propiedades de una clase principal.

Por ejemplo, si estamos pensando en un animal, se nos pueden ocurrir ciertas propiedades: nombre, peso, altura, velocidad máxima, colores y muchas más. Y luego, si pensamos en una especie específica de pez, podemos reutilizar todas las propiedades de "animal" y agregar algunas propiedades específicas de peces allí también. Lo mismo para los perros; si luego pensamos en un perro, podemos reutilizar todas las propiedades de "animal" y agregarle algunas específicas para perros. De esta forma tenemos código reutilizable de nuestra clase animal. Y cuando nos damos cuenta de que olvidamos una propiedad muy importante para muchos animales en nuestra aplicación, solo necesitamos agregarla a la clase animal.

Esto es muy importante para Java, .NET y otras formas clásicas de escribir código orientadas a objetos. JavaScript no gira necesariamente en torno a los objetos. Los necesitaremos y los usaremos, pero no son la estrella de nuestro código, por así decirlo.

clases y objetos

Como repaso rápido, los objetos son una colección de propiedades y métodos. Los vimos en el *Capítulo* 3, *Valores múltiples de JavaScript.* Las propiedades de un objeto deben tener nombres sensibles. Entonces, por ejemplo, si tenemos un objeto persona , este objeto podría tener propiedades llamadas edad y apellido que contienen valores. Aquí hay un ejemplo de un objeto:

```
let dog = { dogName: "JavaScript", peso:
                    2.4, color: "marrón", raza:
                    "chihuahua"

                };
```

Las clases en JavaScript encapsulan datos y funciones que forman parte de esa clase. Si crea una clase, luego puede crear objetos usando esa clase usando la siguiente sintaxis:

```
clase nombre de clase {
   constructor(prop1, prop2) {
      esto.prop1 = prop1;
```

```
        this.prop2 = prop2;
    }
}

let obj = new ClassName("arg1", "arg2");
```

Este código define una clase con ClassName como nombre, declara una variable obj e inicializa esto con una nueva instancia del objeto. Se proporcionan dos argumentos. Estos argumentos serán utilizados por el constructor para inicializar las propiedades. Como puede ver, los parámetros del constructor y las propiedades de la clase (prop1 y prop2) tienen el mismo nombre. Las propiedades de la clase se pueden reconocer por la palabra clave this delante de ellas. La palabra clave this se refiere al objeto al que pertenece, por lo que es la primera propiedad de la instancia de ClassName.

Recuerde que dijimos que las clases son solo una función especial debajo de la superficie. Podríamos crear el objeto con una función especial como esta:

```
function Perro(nombrePerro, peso, color, raza) {
    this.dogName = dogName;
    este.peso = peso; este.color
    = color; esta.raza = raza;

}

let dog = new Dog("Jacky", 30, "brown", "labrador");
```

El ejemplo del perro también podría haberse hecho usando una sintaxis de clase. Se habría visto así:

```
perro de clase {
    constructor(nombreperro, peso, color, raza) {
        this.dogName = dogName;
        este.peso = peso;
        este.color = color;
        esta.raza = raza;
    }
}

let dog = new Dog("JavaScript", 2.4, "brown", "chihuahua");
```

Esto da como resultado un objeto con las mismas propiedades. Si hacemos un registro de la siguiente manera, podremos verlo:

```
console.log(perro.nombrePerro, "es un", perro.raza, "y pesa", perro.peso, "kg.");
```

Esto generará:

```
JavaScript es un chihuahua y pesa 2,4 kg.
```

En la siguiente sección, profundizaremos en todas las partes de las clases.

Clases

Puede preguntarse, si las clases hacen exactamente lo mismo que simplemente definir un objeto, ¿por qué necesitamos clases? La respuesta es que las clases son esencialmente planos para la creación de objetos. Esto significa que necesitamos escribir mucho menos si necesitamos crear 20 perros cuando tenemos una clase de perros . Si tenemos que crear los objetos, tendremos que especificar todos los nombres de las propiedades cada vez. Y sería fácil cometer un error tipográfico y escribir mal el nombre de una propiedad. Las clases son útiles en este tipo de situaciones.

Como se muestra en la sección anterior, usamos la palabra clave class para decirle a JavaScript que queremos crear una clase. A continuación, le damos un nombre a la clase. Es la convención comenzar los nombres de las clases con una letra mayúscula.

Echemos un vistazo a todos los diferentes elementos de una clase.

Constructores

El método constructor es un método especial que usamos para inicializar objetos con nuestro modelo de clase. Solo puede haber un constructor en una clase. Este constructor contiene propiedades que se establecerán al iniciar la clase.

Aquí puedes ver un ejemplo de un constructor en una clase Person :

```
Persona de clase {
    constructor(nombre, apellido) {
        este.nombre = nombre;
        este.apellido = apellido;
    }
}
```

Debajo de la superficie, JavaScript crea una función especial basada en este constructor.
Esta función obtiene el nombre de la clase y creará un objeto con las propiedades dadas. Con esta función especial, puede crear instancias (objetos) de la clase.

Así es como puede crear un nuevo objeto a partir de la clase Persona :

```
let p = nueva Persona("Maaike", "van Putten");
```

La nueva palabra es lo que le dice a JavaScript que busque la función constructora especial en la clase Person y cree un nuevo objeto. Se llama al constructor y devuelve una instancia del objeto persona con las propiedades especificadas. Este objeto se almacena en la variable p .

Si usamos nuestra nueva variable p en una declaración de registro, puede ver que las propiedades están realmente establecidas:

```
console.log("Hola", p.nombre);
```

Esto da como resultado:

```
hola maaike
```

¿Qué crees que pasará cuando creamos una clase sin todas las propiedades?
Vamos a averiguar:

```
let p = nueva Persona("Maaike");
```

Muchos idiomas fallarían, pero no JavaScript. Simplemente establece las propiedades restantes como indefinidas. Puede ver lo que sucede al iniciar sesión:

```
console.log("Hola", p.nombre, p.apellido);
```

Esto resulta en:

```
Hola Maaike indefinido
```

Puede especificar valores predeterminados en el constructor. Lo harías así:

```
constructor(nombre, apellido = "Doe") {
        este.nombre = nombre;
        este.apellido = apellido;
    }
```

De esta forma, no habría impreso Hi Maaike indefinido, sino Hi Maaike Doe.

Ejercicio de práctica 7.1

Realice los siguientes pasos para crear una clase de persona e imprimir instancias de amigos.
nombres:

1. Cree una clase para Person que incluya el constructor para firstname y apellido.
2. Cree una variable y asigne un valor al nuevo objeto Persona usando su nombre y apellido de un amigo.
3. Ahora agregue una segunda variable con el nombre de otro amigo usando su primer nombre Y apellido.
4. Envíe a ambos amigos a la consola con un saludo de hola.

Métodos

En una clase, podemos especificar funciones. Esto significa que nuestro objeto puede comenzar a hacer cosas utilizando las propiedades del objeto, por ejemplo, imprimir un nombre. Las funciones en una clase se llaman métodos. Al definir estos métodos, no usamos la palabra clave function . Empezamos directamente con el nombre:

```
Persona de clase {
    constructor(nombre, apellido) {
        este.nombre = nombre;
        este.apellido = apellido;
    }

    saludar() {
        console.log("¡Hola ! Soy", this.firstname);
    }
}
```

Podemos llamar al método de saludo en un objeto Person como este:

```
let p = nueva Persona("Maaike", "van Putten");
p.saludo();
```

Saldrá:

```
¡Hola! soy maaike
```

Puede especificar tantos métodos en una clase como desee. En este ejemplo, estamos usando la propiedad firstname . Lo hacemos diciendo this.property. Si tuviéramos una persona con un valor diferente para la propiedad del nombre , por ejemplo, Rob, habría impreso:

```
¡Hola! soy roberto
```

Al igual que las funciones, los métodos también pueden tomar parámetros y devolver resultados:

```
Persona de clase {
    constructor(nombre, apellido) {
        este.nombre = nombre;
        este.apellido = apellido;
    }

    saludar() {
        console.log("¡Hola !");
    }

    cumplido(nombre, objeto) {
        regresar "Eso es maravilloso"          + objeto + ", "          + nombre;
    }
}
```

La función de cumplido no genera nada en sí misma, por lo que la estamos registrando.

```
let cumplido = p. cumplido("Harry", "sombrero");
consola.log(feliz);
```

La salida será:

```
Ese es un sombrero maravilloso, Harry.
```

En este caso, estamos enviando parámetros a nuestro método, porque normalmente no complementas tus propias propiedades (¡esa es una buena oración, Maaike!). Sin embargo, siempre que el método no requiera una entrada externa sino solo las propiedades del objeto, ningún parámetro funcionará y el método puede usar las propiedades de su objeto.
Hagamos un ejercicio y luego pasemos a usar las propiedades de las clases fuera de la clase.

Ejercicio de práctica 7.2

Obtenga el nombre completo de su amigo:

1. Utilizando la clase Person del *ejercicio de práctica 7.1,* agregue un método llamado nombre completo, que devuelve el valor concatenado de nombre y apellido cuando es invocado.
2. Cree valores para person1 y person2 usando los nombres y apellidos de dos amigos.
3. Usando el método de nombre completo dentro de la clase, devuelva el nombre completo de uno o ambas personas.

Propiedades

Las propiedades, a veces también llamadas campos, contienen los datos de la clase. Ya hemos visto un tipo de propiedad, cuando las creamos en nuestros constructores:

```
Persona de clase {
    constructor(nombre, apellido) {
        este.nombre = nombre;
        este.apellido = apellido;
    }
}
```

Aquí, la clase Persona obtiene dos propiedades del constructor: nombre y apellido . Las propiedades se pueden agregar o eliminar tal como lo hicimos con los objetos. Se puede acceder a estas propiedades desde fuera de la clase, como vimos cuando las registramos fuera de la clase accediendo a ellas en la instancia:

```
let p = nueva Persona("Maaike", "van Putten");
console.log("Hola", p.nombre);
```

A menudo, no es deseable proporcionar acceso directo a nuestras propiedades. Queremos que nuestra clase tenga el control de los valores de las propiedades por varias razones, tal vez queremos validar una propiedad para asegurarnos de que tiene un valor determinado. Por ejemplo, imagina querer validar una edad que no sea menor de 18 años. Esto lo podemos lograr imposibilitando el acceso directo a la propiedad desde fuera de la clase.

Así es como se agregan propiedades que no son accesibles desde el exterior. Los prefijamos con un símbolo # :

```
Persona de clase {
    #primer nombre;
    #apellido;
```

```
    constructor(nombre, apellido) {
        this.#firstname = firstname;
        this.#apellido = apellido;
    }
}
```

En este momento, no se puede acceder a las propiedades firstname y lastname desde fuera de la clase. Esto se hace agregando # delante de la propiedad. Si lo intentamos:

```
let p = nueva Persona("María", "Saga");
consola.log(p.nombre);
```

Nosotros recibiremos:

```
indefinido
```

Si quisiéramos asegurarnos de que solo pudiéramos crear objetos con nombres que comenzaran con un "M", podríamos modificar un poco nuestro constructor:

```
constructor(nombre, apellido)
        { if(nombre.empiezaCon("M")){
            this.#firstname = firstname; } más {

            this.#firstname = "M" + firstname;
        }
        this.#apellido = apellido;
    }
```

Ahora, cuando intente crear una persona que tenga un valor de nombre que no comience con una "M", agregará una M al frente. Entonces, por ejemplo, el valor del siguiente nombre es Mkay:

```
let p = nueva Persona("kay", "Luna");
```

Este es un ejemplo muy tonto de validación. En este punto, no podemos acceder a él desde fuera de la clase después del constructor. Solo podemos acceder desde dentro de la clase. Aquí es donde entran en juego los getters y setters.

Getters y setters

Getters y setters son propiedades especiales que podemos usar para obtener datos de una clase y establecer campos de datos en la clase. Getters y setters son propiedades calculadas. Entonces, son más como propiedades que como funciones. Los llamamos accesorios. Se parecen un poco a las funciones, porque tienen () detrás de ellas, ¡pero no lo son!

Estos elementos de acceso comienzan con las palabras clave get y set . Se considera una buena práctica hacer que los campos sean privados tanto como sea posible y proporcionar acceso a ellos mediante getters y setters. De esta forma, las propiedades no se pueden establecer desde el exterior sin que el propio objeto tenga el control. Este principio se llama **encapsulación.** La clase encapsula los datos y el objeto tiene el control de sus propios campos.

Aquí está cómo hacerlo:

```
Persona de clase {
    #primer nombre;
    #apellido;
    constructor(nombre, apellido) {
        this.#firstname = firstname;
        this.#apellido = apellido;
    }

    obtener nombre() {
        devuelve esto.#nombre;
    }

    establecer nombre(nombre) {
        this.#firstname = firstname;
    }

    obtener apellido() {
        devuelve esto.#apellido;
    }

    establecer apellido(apellido) {
        this.#apellido = apellido;
    }
}
```

El captador se utiliza para obtener la propiedad. Por lo tanto, no toma ningún parámetro, sino que simplemente devuelve la propiedad. El setter es al revés: toma un parámetro, asigna este nuevo valor a la propiedad y no devuelve nada. El setter puede contener más lógica, por ejemplo, alguna validación, como veremos a continuación. El getter se puede usar fuera del objeto como si fuera una propiedad. Ya no se puede acceder directamente a las propiedades desde fuera de la clase, pero se puede acceder a ellas mediante el getter para obtener el valor y mediante el setter para actualizar el valor. Aquí se explica cómo usarlo fuera de la instancia de la clase:

```
let p = nueva Persona("María", "Saga");
consola.log(p.nombre);
```

Esto generará:

```
María
```

Hemos creado un nuevo objeto Persona con el nombre de María y el apellido de Saga. La salida muestra el primer nombre, que solo es posible porque tenemos un descriptor de acceso getter. También podemos establecer el valor en otra cosa, porque hay un setter. A continuación se explica cómo actualizar el nombre, de modo que el nombre ya no sea María, sino Adnane.

```
p.nombre = "Adnane";
```

En este punto, nada especial sucede en el colocador. Podríamos hacer una validación similar a la del constructor anterior, así:

```
establecer nombre(nombre)
        { if(nombre.comienzaCon("M")){
            this.#firstname = firstname; } más {

            this.#firstname = "M" + firstname;
        }
    }
```

Esto verificará si firstname comienza con una M, y si lo hace, actualizará el valor a cualquiera que sea el parámetro firstname . Si no es así, concatenará una M delante del parámetro.

Tenga en cuenta que no accedemos a firstname como si fuera una función. Si coloca dos paréntesis () después, en realidad obtendrá un error que le indicará que no es una función.

Herencia

La herencia es uno de los conceptos clave de la programación orientada a objetos. Es el concepto de que las clases pueden tener clases secundarias que heredan las propiedades y métodos de la clase principal. Por ejemplo, si necesita todo tipo de objetos de vehículos en su aplicación, puede especificar una clase llamada Vehículo en la que especifique algunas propiedades y métodos compartidos de los vehículos. Luego continuaría y crearía las clases secundarias específicas basadas en esta clase de Vehículo , por ejemplo, bote, automóvil, bicicleta y motocicleta.

Esta podría ser una versión muy simple de la clase Vehicle :

```
clase de vehículo {
    constructor (color, velocidad actual, velocidad máxima) {
        este.color = color;
```

```
        this.currentSpeed = currentSpeed;
        this.maxSpeed = maxSpeed;
    }

    mover() {
        console.log("moviéndose a", this.currentSpeed);
    }

    acelerar (cantidad) {
        this.currentSpeed += cantidad;
    }
}
```

Aquí tenemos dos métodos en nuestra clase Vehicle : mover y acelerar. Y esto podría ser una clase Motorcyle heredada de esta clase usando la palabra clave extends :

```
clase Motocicleta extiende Vehículo {
    constructor (color, velocidad actual, velocidad máxima, combustible) {
        super (color, velocidad actual, velocidad máxima);
        this.combustible = combustible;
    }
    doWheelie() {
        console.log("¡Conduciendo sobre una rueda!");
    }
}
```

Con la palabra clave extends especificamos que cierta clase es hija de otra clase.
En este caso, Motocicleta es una clase secundaria de Vehículo. Esto significa que tendremos acceso a las propiedades y métodos de Vehicle en nuestra clase Motorcycle . Hemos agregado un método especial doWheelie() . Esto no es algo que tenga sentido agregar a la clase Vehículo , porque esta es una acción que es específica de ciertos vehículos.

La superpalabra en el constructor llama al constructor desde el padre, el constructor del vehículo en este caso. Esto asegura que los campos del padre también estén configurados y que los métodos estén disponibles sin tener que hacer nada más: se heredan automáticamente. Llamar a super() no es opcional, debe hacerlo cuando está en una clase que hereda de otra clase, de lo contrario obtendrá un ReferenceError.

Debido a que tenemos acceso a los campos de Vehículo en Motocicleta, esto funcionará:

```
let motor = motocicleta nueva ("Negra", 0, 250, "gasolina");
consola.log(motor.color);
```

```
motor.acelerar(50);
motor.mover();
```

Y esto es lo que saldrá:

```
Negro
mudarse a los 50
```

No podemos acceder a ninguna propiedad o método específico de motocicleta en nuestra clase de vehículo . Esto se debe a que no todos los vehículos son motocicletas, por lo que no podemos estar seguros de que tendríamos las propiedades o los métodos de un niño.

En este momento, no usamos getters ni setters aquí, pero claramente podríamos hacerlo. Si hay getters y setters en la clase principal, la clase secundaria también los hereda.
De esta manera, podríamos influir en qué propiedades se pueden buscar y cambiar (y cómo) fuera de nuestra clase. Esto es generalmente una buena práctica.

prototipos

Un prototipo es el mecanismo en JavaScript que hace posible tener objetos.
Cuando no se especifica nada al crear una clase, los objetos heredan del prototipo Object.prototype . Esta es una clase de JavaScript incorporada bastante compleja que podemos usar. No necesitamos ver cómo se implementa esto en JavaScript, ya que podemos considerarlo el objeto base que siempre está en la parte superior del árbol de herencia y, por lo tanto, siempre presente en nuestros objetos.

Hay una propiedad de prototipo disponible en todas las clases, y siempre se denomina "prototipo". Podemos acceder a él así:

```
ClassName.prototipo
```

Demos un ejemplo de cómo agregar una función a una clase usando el prototipo propiedad. Para hacerlo, usaremos esta clase Person :

```
Persona de clase {
    constructor(nombre, apellido) {
        este.nombre = nombre;
        este.apellido = apellido;
    }

    saludar() {
        console.log("¡Hola !");
    }
}
```