Herencia de clase

La herencia de clase es el modo para que una clase extienda a otra.

De esta manera podemos añadir nueva funcionalidad a la ya existente.

La palabra clave “extends”

Digamos que tenemos la clase Animal:

```
class Animal {
    constructor(name) {
      this.speed = 0;
      this.name = name;
    }
    run(speed) {
      this.speed = speed;
      alert(`${this.name} corre a una velocidad de ${this.speed}.`);
    }
    stop() {
      this.speed = 0;
      alert(`${this.name} se queda quieto.`);
    }
  }

  let animal = new Animal("Mi animal");
```

Así es como podemos representar gráficamente el objeto animal y la clase Animal:

…Y nos gustaría crear otra clase Rabbit.

Como los conejos son animales, la clase ‘Rabbit’ debería basarse en ‘Animal’ y así tener acceso a métodos animales, para que los conejos puedan hacer lo que los animales “genéricos” pueden hacer.

La sintaxis para extender otra clase es: class Hijo extends Padre.

Construyamos la clase Rabbit que herede de Animal:

```
class Rabbit extends Animal {
    hide() {
      alert(`¡${this.name} se esconde!`);
    }
  }

  let rabbit = new Rabbit("Conejo Blanco");
```

```
 8
 9  rabbit.run(5); // Conejo Blanco corre a una velocidad de 5.
10  rabbit.hide(); // ¡Conejo Blanco se esconde!
```

Los objetos de la clase Rabbit tienen acceso a los métodos de Rabbit, como rabbit.hide(), y también a los métodos Animal, como rabbit.run().

Internamente, la palabra clave **extends** funciona con la buena mecánica de prototipo: establece **Rabbit.prototype.[[Prototype]] a Animal.prototype**. Entonces, si no se encuentra un método en **Rabbit.prototype**, JavaScript lo toma de **Animal.prototype**.

Por ejemplo, para encontrar el método rabbit.run, el motor revisa (en la imagen, de abajo hacia arriba):

1. El objeto rabbit: no tiene el método run.
2. Su prototipo, que es Rabbit.prototype: tiene el método hide, pero no el método run.
3. Su prototipo, que es Animal.prototype (debido a extends): Este finalmente tiene el método run.

Cualquier expresión está permitida después de extends

La sisntaxis de clase permite especificar no solo una clase, si no cualquier expresión después de **extends**. Por ejemplo, una llamada a la función que genera la clase padre

```
1  function f(phrase) {
2    return class {
3      sayHi() { alert(phrase); }
4    };
5  }
6
7  class User extends f("Hola") {}
8
9  new User().sayHi(); // Hola
10
```

Sobrescribir un método

Ahora avancemos y sobrescribamos un método. Por defecto, todos los métodos que no están especificados en la clase Rabbit se toman directamente "tal cual" de la clase Animal.

Pero Si especificamos nuestro propio método stop() en Rabbit, es el que se utilizará en su lugar:

```
class Rabbit extends Animal {
  stop() {
    // ...esto se usará para rabbit.stop()
    // en lugar de stop() de la clase Animal
  }
}
```

Sin embargo, no siempre queremos reemplazar totalmente un método padre sino construir sobre él, modificarlo o ampliar su funcionalidad. Hacemos algo con nuestro método, pero queremos llamar al método padre antes, después o durante el proceso.

Las clases proporcionan la palabra clave "super" para eso.

- **super.metodo(...)** llama un método padre.
- **super(...)** llama un constructor padre (solo dentro de nuestro constructor).

Por ejemplo, hagamos que nuestro conejo se oculte automáticamente cuando se detenga:

```
class Animal {

  constructor(name) {
    this.speed = 0;
    this.name = name;
  }

  run(speed) {
    this.speed = speed;
    alert(`${this.name} corre a una velocidad de ${this.speed}.`);
  }

  stop() {
    this.speed = 0;
    alert(`${this.name} se queda quieto.`);
  }

}

class Rabbit extends Animal {
  hide() {
    alert(`¡${this.name} se esconde!`);
  }

  stop() {
    super.stop(); // llama el stop padre
    this.hide(); // y luego hide
```

```
28    }
29  }
30
31  let rabbit = new Rabbit("Conejo Blanco");
32
33  rabbit.run(5); // Conejo Blanco corre a una velocidad de 5.
34  rabbit.stop(); // Conejo Blanco se queda quieto. ¡Conejo Blanco se
esconde!
```

Ahora `Rabbit` tiene el método `stop` que llama al padre `super.stop()` en el proceso.

Sobrescribir un constructor

Con los constructores se pone un poco complicado.

Hasta ahora, Rabbit no tenía su propio constructor.

De acuerdo con la especificación, si una clase extiende otra clase y no tiene constructor, se genera el siguiente constructor "vacío":

```
1  class Rabbit extends Animal {
2    // es generado por extender la clase sin constructor propio
3    constructor(...args) {
4      super(...args);
5    }
6  }
```

Como podemos ver, básicamente llama al constructor padre pasándole todos los argumentos. Esto sucede si no escribimos un constructor propio.

Ahora agreguemos un constructor personalizado a Rabbit. Especificará earLength además de name:

```
 1  class Animal {
 2    constructor(name) {
 3      this.speed = 0;
 4      this.name = name;
 5    }
 6    // ...
 7  }
 8
 9  class Rabbit extends Animal {
10
11    constructor(name, earLength) {
12      this.speed = 0;
```

```
13      this.name = name;
14      this.earLength = earLength;
15    }
16
17    // ...
18  }
19
20  // No funciona!
21  let rabbit = new Rabbit("Conejo Blanco", 10); // Error: this no está
definido.
```

¡Vaya! Tenemos un error. Ahora no podemos crear conejos. ¿Qué salió mal?

La respuesta corta es:

- Los constructores en las clases heredadas deben llamar a super(...), y hacerlo antes de usar this.

¿Pero por qué? ¿Qué está pasando aquí? De hecho, el requisito parece extraño.

Por supuesto, hay una explicación. Vamos a entrar en detalles, para que realmente entiendas lo que está pasando.

En JavaScript, hay una distinción entre una función constructora de una clase heredera (llamada "constructor derivado") y otras funciones. Un constructor derivado tiene una propiedad interna especial [[ConstructorKind]]:"derived". Esa es una etiqueta interna especial.

Esa etiqueta afecta su comportamiento con new.

- Cuando una función regular se ejecuta con new, crea un objeto vacío y lo asigna a this.

- Pero cuando se ejecuta un constructor derivado, no hace esto. Espera que el constructor padre haga este trabajo.

Entonces un constructor derivado debe llamar a super para ejecutar su constructor padre (base), de lo contrario no se creará el objeto para this. Y obtendremos un error.

Para que el constructor Rabbit funcione, necesita llamar a super() antes de usar this, como aquí:

```
class Animal {

  constructor(name) {
    this.speed = 0;
    this.name = name;
  }

  // ...
}

class Rabbit extends Animal {

  constructor(name, earLength) {
    super(name);
    this.earLength = earLength;
  }

  // ...
}

// todo bien ahora
let rabbit = new Rabbit("Conejo Blanco", 10);
alert(rabbit.name); // Conejo Blanco
alert(rabbit.earLength); // 10
```

Sobrescribiendo campos de clase.

Podemos sobrescribir no solo métodos, sino también los campos de la clase.

Pero hay un comportamiento peculiar cuando accedemos a los campos sobrescritos en el constructor padre, muy diferente a de la mayoría de los demás lenguajes de programación.

Considera este ejemplo:

```
class Animal {
   name = 'animal';

 constructor() {
     alert(this.name); // (*)
   }
}

class Rabbit extends Animal {
   name = 'rabbit';
```

```
11  }
12
13  new Animal(); // animal
14  new Rabbit(); // animal
15 Aquí, la clase Rabbit extiende Animal y sobrescribe el campo name con un valor propio.
```

Rabbit no tiene su propio constructor, entonces es llamado el de Animal.

Lo interesante es que en ambos casos: new Animal() y new Rabbit(), el alert en la línea (*) muestra animal.

En otras palabras, el constructor padre siempre usa el valor de su propio campo de clase, no el sobrescrito.

¿Qué es lo extraño de esto?

Si esto aún no está claro, comparalo con lo que ocurre con los métodos.

Aquí está el mismo código, pero en lugar del campo this.name llamamos el método this.showName():

```
1 class Animal {
2    showName() {  // en vez de this.name = 'animal'
3      alert('animal');
4    }
5
6    constructor() {
7      this.showName(); // en vez de alert(this.name);
8    }
9  }
10
11  class Rabbit extends Animal {
12    showName() {
13      alert('rabbit');
14    }
15  }
16
17  new Animal(); // animal
18  new Rabbit(); // rabbit
```

Observa que ahora la salida es diferente.

Y es lo que esperamos naturalmente. Cuando el constructor padre es llamado en la clase derivada, usa el método sobrescrito.

…Pero con los campos esto no es así. Como dijimos antes, el constructor padre siempre utiliza el campo padre.

¿Por que existe la diferencia?

Bien, la razón está en el orden de inicialización, El campo de clase es inicializado:

- Antes del constructor para la clase de base (que no extiende nada),
- Inmediatamente después de super() para la clase derivada.

En nuestro caso, Rabbit es la clase derivada. No hay constructor() en ella. Como establecimos previamente, es lo mismo que si hubiera un constructor vacío con solamente super(...args).

Entonces, new Rabbit() llama a super() y se ejecuta el constructor padre, y (por la regla de la clase derivada) solamente después de que sus campos de clase sean inicializados. En el momento de la ejecución del constructor padre, todavía no existen los campos de clase de Rabbit, por ello los campos de Animal son los usados.

Esta sutil diferencia entre campos y métodos es particular de JavaScript

Afortunadamente este comportamiento solo se revela si los campos sobrescritos son usados en el constructor padre. En tal caso puede ser difícil entender qué es lo que está pasando, por ello lo explicamos aquí.

Si esto se vuelve un problema, uno puede corregirlo usando métodos o getters/setters en lugar de campos.