
Funciones y clases

Las funciones y clases son lo que usamos para envolver ciertas piezas de funcionalidad en bloques reutilizables. Al usar una función o clase, podemos repetir tareas específicas muchas veces sin tener que reescribir el código. Cuando JavaScript se lanzó originalmente, solo tenía funciones y usaba prototipos para la herencia. Cubrimos este tipo de herencia cuando revisamos a los objetos.

Las clases llegaron más tarde a JavaScript, pero en gran medida son solo ‘azúcar sintáctica’ para la herencia basada en prototipos. Como tal, muchos desarrolladores en JavaScript eligen no usar clases y, en su lugar, dependen de la herencia prototípica.

Al igual que con todas las cosas en el desarrollo de software, realmente no importa si decides usar clases o dependes de la herencia prototípica con las funciones. Lo importante es que eres consistente a lo largo de tus proyectos.

Introducción a las funciones

Una función típica consta de tres partes:

1. Una entrada, conocida como argumentos (aunque a veces, no se dan entradas).
2. Algún código para manipular esas entradas o realizar algún tipo de acción.
3. Una salida, generalmente relacionada con la entrada. No tiene que ser así, pero es mejor si lo es.

Las entradas en las funciones se denominan argumentos, y devolvemos salidas utilizando la palabra clave *return*. Luego, cuando ejecutamos una función, devuelve ese valor. Si su función no devuelve nada, eso también es aceptable, pero es mejor evitar si es posible.

Cuando no devuelve algo de una función en JavaScript, la función siempre devolverá “indefinida (*undefined*)” de forma predeterminada, por lo que la palabra clave de *return* es técnicamente opcional en una función.

Entonces, ¿cómo se ve una función? En el siguiente ejemplo, creamos una función simple que devuelve "Hello World" al ejecutar:

```
function myFunction() {  
  return "Hello World"
```

```
}  
console.log(myFunction())
```

Ejecutamos funciones escribiendo sus nombres, seguido de paréntesis ().

En el ejemplo anterior, myFunction no tiene entradas ni argumentos, y todo lo que hace es devolver el texto "Hello World". Ejecutamos esta función dentro del registro de la consola, lo que significa que la consola produciría el texto "Hello World" como se muestra en la figura 1.



```
> function myFunction() {  
    return "Hello World"  
}  
console.log(myFunction())  
Hello World
```

Figura 1: Se puede ejecutar una función en cualquier contexto, incluso dentro de otra función. Dado que el valor de retorno de la función anterior es "Hello World", la ejecución de myFunction devolverá esa cadena, que luego se puede visualizar con console log.

Los argumentos se agregan dentro de la declaración de la función, dentro de los paréntesis redondos (). Luego, los argumentos actúan como variables dentro del alcance de la función. Por ejemplo, la siguiente función crea una oración compuesta de dos partes. Aunque no es particularmente útil en la práctica, te da una idea de cómo funcionan los argumentos:

```
function words(word1, word2) {  
    return word1 + " " + word2  
}  
console.log(words("Hello", "John")) // Hello John  
console.log(words("Hello", "Jake")) // Hello Jake  
console.log(words("Good bye", "Alice")) // Good bye Alice
```

Como discutimos temprano en las notas, JavaScript es dinámicamente tipado.

Cuando definimos nuestra función anterior, llamamos a nuestros argumentos word1 y word2. Aunque se llaman a esto, podemos proporcionar múltiples palabras, y a JavaScript no le importará. Esto tiene sus ventajas, ya que es más simple, pero también tiene sus desventajas.

Por ejemplo, ¿qué pasaría si solo esperaras palabras en tu función, pero se dieron números u objetos? Esto hace que las declaraciones de nombres y control como if...else dentro de las funciones sean más importantes en JavaScript que en otros lenguajes.

Cuando llamamos a una nueva función, se agrega a la parte superior de la pila para nuestro programa, al igual que una variable, como se demuestra en la figura 2.

Nota: Las funciones son de tipo objeto, por lo que cuando las declaramos, sus datos se almacenan en el heap. Sin embargo, llamarlas las agregará a la pila de llamadas como variables.

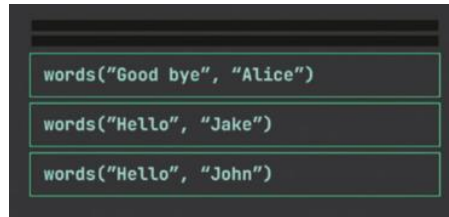


Figura 2: Las funciones se agregan a la parte superior de la pila a medida que se declaran, al igual que las variables.

Ejecutar argumentos con los “tres puntos”

Los argumentos también se pueden llamar en una función a través de un arreglo, utilizando la sintaxis de tres puntos, que cubrimos anteriormente cuando observamos cómo fusionar arreglos y objetos. Veamos cómo funciona eso usando nuestro ejemplo anterior:

```
function words(word1, word2) {
  return word1 + " " + word2
}
let validWords = [ "Hello", "John" ]
console.log(words(...validWords)) // Hello John
```

Esto es realmente útil en el mundo real, donde a menudo tenemos datos almacenados en arreglos y objetos, que luego queremos pasar a las funciones.

Formas alternativas de llamar a las funciones

JavaScript tiene una tendencia a tener muchas formas de hacer lo mismo, y las funciones no son la excepción. Hay tres formas de declarar y llamar a las funciones.

Expresiones de funciones sin nombre

Las expresiones de funciones no identificadas se nombran confusamente ya que tienen un nombre, pero el nombre se expresa a través de la variable. Puedes definirlos a través de variables `let` o `const` - y luego el nombre de la variable se convierte en el nombre de la función. Aquí está nuestro ejemplo anterior, usando variables en su lugar:

```
let words = function(word1, word2) {
  return word1 + " " + word2
}
```

```
}  
console.log(words("Hello", "World"))
```

También podemos poner funciones como está dentro de los objetos, lo que nos permite agrupar las funciones o agregar métodos a un prototipo:

```
let wordFunctions = {  
  words: function(word1, word2) {  
    return word1 + " " + word2  
  }  
}
```

En el ejemplo anterior, llamar a `wordFunctions.words()` nos permitiría ejecutar esta función.

Funciones anónimas

Las funciones anónimas (también a veces denominadas expresión de la función invocada inmediatamente o IIFES, Immediately Invoked Function Expression) son funciones que en realidad no tienen nombre y se llaman de inmediato. Para que se ejecuten de inmediato, lo envolvemos en los paréntesis redondos y lo llamamos con doble `()` como antes:

```
(function(word1, word2) {  
  return word1 + " " + word2  
})("Hello", "World")
```

Los argumentos se colocan en el segundo conjunto de paréntesis para que puedan pasar a la función. El uso de funciones anónimas está decayendo, pero a veces todavía se usan para crear un alcance separado para trabajar dentro.

Funciones con notación de flecha

La notación de la flecha es otra forma de definir funciones. Esta notación viene con una funcionalidad diferente en comparación con las otras definiciones de funciones que hemos cubierto. La notación de la flecha se llama como tal porque usa `=>` para indicar dónde comienza el cuerpo de la función.

Así es como se ve nuestra función anterior con la notación de flecha. Cuando intentas llamarla, funciona igual que nuestras otras expresiones de funciones:

```
let words = (word1, word2) => {  
  return word1 + " " + word2  
}  
console.log(words("Hello", "John")) // Hello John
```

Las funciones de notación de flecha son diferentes en que no almacenan un contexto único. Para comprender lo que eso significa, primero tenemos que entender lo que significa esta palabra clave en JavaScript y, por lo tanto, un poco sobre el modo estricto.

Funciones y la palabra clave “this”

La palabra clave *this* puede ser un poco difícil de entender en JavaScript, ya que se comporta de manera diferente a la de otros lenguajes. El objetivo principal de la palabra clave es contener información sobre su contexto actual.

En el nivel superior de tu código, fuera de cualquier función, el contexto es “global”. Cuando usamos la palabra clave “this” en el contexto global en los navegadores, se refiere a un objeto llamado *ventana*. El objeto de la ventana contiene mucha información útil sobre tu contexto actual. Por ejemplo, `window.innerWidth` te indica el ancho de la ventana del navegador del usuario. También contiene información sobre la posición del mouse, y es donde existen todas las APIs web.

Como tal, los dos registros de consola en el siguiente ejemplo muestran lo mismo:

```
:console.log(this) // Console logs window object  
console.log(window) // Console logs window object
```

Dado lo que hemos dicho hasta ahora, cuando llamamos a esta palabra clave dentro de una función, puedes esperar que esta palabra clave se refiera al contexto de la función, pero encontrarás que todavía muestra el global this del objeto:

```
console.log(this) // Window { }  
let words = function (word1, word2) {  
  console.log(this) // Window { }  
  return word1 + " " + word2  
}
```

Esto no tiene mucho sentido cuando lo piensas conceptualmente.

¿No debería una función tener su propio contexto y, por lo tanto, tener su propio this?

La razón por la cual este es el caso no es muy complicado: JavaScript fue creado originalmente para que los novatos hicieran scripting rápido. Entonces, para facilitar las cosas, se pone automáticamente una función de este valor con el global this o window.

Eso significa que window está disponible de forma predeterminada en todas tus funciones a través de esta palabra clave. Eso es un poco útil, pero se vuelve desordenado si deseas que una función tenga un contexto coherente y separado.

Modo descuidado (sloppy)

Cuando escribimos el código JavaScript, de forma predeterminada, todo el código generalmente se escribe en algo llamado “modo descuidado”, que se adapta a los novatos al ignorar algunos errores y hacer que las funciones hereden el contexto global.

Para salir del modo descuidado, tenemos que cambiar a algo llamado modo “estricto (strict)”.

El modo estricto aporta muchas ventajas a tu código, el principal está separando los contextos de funciones del contexto global. Tanto los archivos como las funciones se pueden hacer estrictas agregando el texto “use strict” en la parte superior. Al poner nuestro código en modo estricto, podemos dar a cada función su propio contexto y, por lo tanto, la palabra clave `this` devolverá indefinido dentro de una función. En el siguiente ejemplo, el modo estricto está habilitado. Para habilitar el modo estricto, solo debes agregar “usar estricto” en la parte superior de tu archivo:

```
"use strict"
console.log(this) // Window { }
let words = function (word1, word2) {
  console.log(this) // undefined
  return word1 + " " + word2
}
```

Funcionalidad de notación de flecha con `this`

Ahora que hemos analizado cómo funcionan diferentes contextos con funciones, volvamos a las funciones de notación de flecha. Las funciones de flecha son un poco diferentes de otras funciones en que no tienen su propio contexto.

Eso significa que incluso en modo estricto, lo heredan de sus padres. Esta funcionalidad solo tiene sentido en modo estricto:

```
"use strict"
console.log(this) // Window { }
let words = () => {
  console.log(this)
}
words() // console logs Window { }
```

Si tu función de flecha está dentro de otra función, que no está utilizando la notación de flecha, hereda el contexto de esa función principal. Esto se puede ver en el siguiente ejemplo:

```
"use strict"
```

```
let contextualFunction = function() {
  let words = () => {
    console.log(this) // console logs undefined
  }
  words()
}
contextualFunction() // console logs undefined
```

En general, el modo estricto es una forma más confiable de escribir código. Además de eso, es una práctica bastante mala exponer variables globales, sin saberlo, a los scripts posteriores, que tal vez no deberían tener acceso a ellas.

Hasta ahora, mientras trabajas en modo estricto, `this` ha sido indefinido dentro de las funciones. Para que esta palabra clave sea más valiosa, queremos darle algún tipo de valor. Para hacer eso, necesitamos comprender las diferentes formas en que podemos llamar funciones con contexto.

Llamar a las funciones con contexto

Hay tres métodos heredados por todas las funciones a través de la herencia prototípica, que nos permiten llamar a funciones con un contexto personalizado.

Estos se enumeran a continuación:

1. `call()`, que llama a una función y le da algún contexto. Usar `call()` puede definir un contexto para una función y también pasar las variables (separadas por comas).
2. `apply()`, que es lo mismo que `call()` pero usa un arreglo para definir los argumentos de una función.
3. `bind()`, que une permanentemente algún contexto a una función, por lo que nunca tienes que redefinir su contexto nuevamente.

`call()`. Supongamos que queremos definir un valor constante que esté disponible dentro de una función específica. Podemos lograr esta funcionalidad agregando nuestra constante al contexto de la función

```
"use strict"
let words = function (word, punctuation) {
  return this.keyword + " " + word + punctuation
}
let wordContext = { keyword: "Hello" }
let helloWorld = words.call(wordContext, "World", "!")
console.log(helloWorld) // "Hello World!"
```

En este ejemplo, pasamos el objeto `wordContext` a `call()` para que se convierta en el contexto de la función y, por lo tanto, es este valor.

Los argumentos se definen después del contexto y se separan por comas, por lo que "World" y "!" Ambos se convierten en palabra y puntuación, respectivamente. Puedes ver la salida de esto en la figura 3.

```
> "use strict"
let words = function (word, punctuation) {
  return this.keyword + " " + word + punctuation
}
let wordContext = { keyword: "Hello" }
let helloWorld = words.call(wordContext, "World", "!")
console.log(helloWorld) // "Hello World!"
Hello World!
```

Figura 3: Las funciones se agregan a la parte superior de la pila a medida que se declaran, al igual que las variables.

Similar al ejemplo anterior, `apply()` funciona esencialmente de la misma manera, con la única diferencia de que los argumentos se definen en un arreglo. Con `apply`, el código se vería así:

```
let helloWorld = words.apply(wordContext, [ "World", "!" ])
```

Si llamamos a `words()` todo el tiempo, encontraremos que tendremos que seguir mencionando nuestro contexto una y otra vez, lo cual no es ideal.

Esto se debe a que necesitamos hacer referencia a él cada vez que usamos `call` o `apply`. En el siguiente ejemplo, queremos llamar a `words()` dos veces y usar el mismo contexto para cada llamada. Esto significa que tenemos que escribir el mismo código dos veces:

```
"use strict"
let words = function (word, punctuation) {
  return this.keyword + " " + word + punctuation
}
let wordContext = {
  keyword: "Hello"
}
let helloWorld = words.call(wordContext, "World", "!")
let goodbye = words.call(wordContext, "Goodbye", "!")
console.log(helloWorld) // "Hello World!"
console.log(goodbye) // "Hello Goodbye!"
```

Aunque no es un problema importante, comienza a convertirse en un problema en las grandes bases de código. En cambio, es más eficiente usar `bind()`. Usando `bind()`, solo mencionamos

nuestro contexto una vez, y luego se enreda permanentemente con nuestra función. Esto se puede ver en el siguiente ejemplo:

```
"use strict"
let words = function (word, punctuation) {
  return this.keyword + " " + word + punctuation
}
let wordContext = {
  keyword: "Hello"
}
let boundWord = words.bind(wordContext)
let helloWorld = boundWord("World", "!")
let goodbye = boundWord("Goodbye", "!")
console.log(helloWorld) // "Hello World!"
console.log(goodbye) // "Hello Goodbye!"
```

Llamar a las funciones con el contexto se presta bien a la herencia funcional. Las funciones pueden heredar ciertas variables o métodos globales a través del contexto.

Funciones de constructor en JavaScript

De la misma manera que pudimos hacer nuevas instancias de arreglos y objetos, podemos usar la palabra clave *new* para generar nuevas instancias de funciones en JavaScript.

Cuando usa la palabra clave *new*, también se tiene el beneficio adicional de crear un nuevo contexto, por lo que tu función no heredará esto desde el contexto global. Debido a esta razón, no puedes usar la palabra clave *new* con funciones de notación de flecha, ya que carecen de contexto.

Veamos un ejemplo para familiarizarnos con este concepto. Funciones llamadas con un *new* actúan muy similares a funciones que no tienen *new*:

```
let myFunction = function(name, age, country) {
  console.log(this) // myFunction { }
}
let newFunction = new myFunction("John", 24, "Britain")
```

Nota: Con la palabra clave *new*, puedes omitir los corchetes dobles al llamar a tu función si no tiene argumentos. Entonces, `new myFunction()` es lo mismo que `new myFunction`.

Dado que generan su propio contexto, el registro de la consola de `this` devolverá `myFunction{ }`, que contiene detalles sobre la función (en el constructor) y el prototipo global

para los tipos de objetos, ya que las funciones son de objeto tipo. Puedes ver cómo se ve esto en la figura 4.

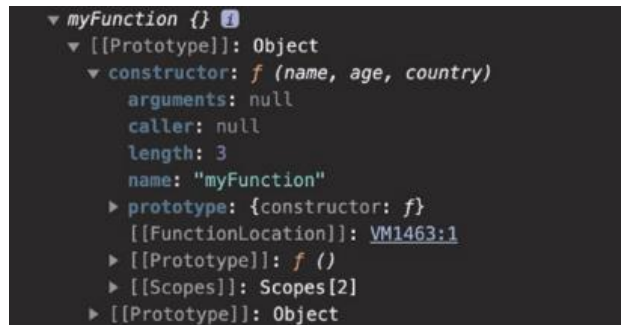


Figura 4: Al usar la palabra clave new, this se refiere al prototipo de la función, que contiene detalles sobre la función y su prototipo de objeto heredado. Esto no es muy importante para escribir código día a día, pero es útil saberlo.

La creación de funciones encapsuladas que tienen su propio contexto resulta ser bastante útil. Por ejemplo, podemos asignar detalles pasados en el argumento de nuestra función a nuestro contexto así:

```
let myFunction = function(name, age, country) {
  this.assignedName = name
  this.assignedAge = age
  this.assignedCountry = country
  console.log(this) // Contains assignedAge, assignedCountry, and assignedName
}
let newFunction = new myFunction("John", 24, "Britain")
console.log(newFunction) // Contains assignedAge,
assignedCountry, and assignedName
```

Cada vez que llamamos a una nueva instancia de nuestra función, creamos un nuevo contexto, y la función es autónoma. Si bien esto parece similar a la creación de un objeto que contiene nuevos valores, nos da el valor agregado de poder manipular o cambiar los valores asignados.

Por ejemplo, supongamos que estamos creando una aplicación que permita el registro del usuario. Podríamos crear una función del constructor que, cuando se llame, es diferente para cada usuario, como se muestra en el siguiente ejemplo:

```
let User = function(firstName, lastName, age) {
  this.fullName = firstName + " " + lastName
  this.age = age
}
```

```
let userOne = new User("John", "Big", 24)
console.log(userOne) // { fullName: "John Big", age: 24 }
let userTwo = new User("John", "Small", 24)
console.log(userTwo) // { fullName: "John Small", age: 24 }
```

También es posible verificar si las funciones se construyeron con `new` si es necesario. Aunque no es una mejor práctica, puedes darle contexto en algunos escenarios de bordes (edge-case). Puedes hacer esto revisando `new.target`, que cuando está indefinido significa que la función se llamaba sin `new`:

```
let User = function(firstName, lastName, age) {
  if(new.target) {
    this.fullName = firstName + " " + lastName
    this.age = age
  }
  else {
    return "Hello World"
  }
}
let userOne = new User("John", "Big", 24)
console.log(userOne) // { fullName: "John Big", age: 24 }
let userTwo = User("John", "Small", 24)
console.log(userTwo) // "Hello World"
```

Métodos de función adicional

Esta forma de escribir código está orientado a objetos. El usuario es el objeto, en este caso, y luego podemos definir que un “usuario” puede hacer ciertas cosas. Podemos adjuntar los métodos que un usuario puede hacer a la función del usuario en sí. La función del usuario, conocida como constructor, luego nos permite construir un usuario, que puede hacer varias cosas. Para ilustrar esto, intentemos agregar un método a nuestra función. Por ejemplo, permitamos que un usuario pueda dar su nombre:

```
let User = function(firstName, lastName, age) {
  this.fullName = firstName + " " + lastName
  this.age = age
}
User.prototype.giveName = function() {
  return `My name is ${this.fullName}!`
}
```

Dado que `fullName` ya está definido por la función constructor, estará disponible en `User.prototype.name`. Luego, cuando queremos que el usuario dé su nombre, solo tenemos que llamar a la función `userOne.giveName()`:

```
let userOne = new User("John", "Big", 24)
// Console logs "My name is John Big!"
console.log(userOne.giveName())
```

Getters and setters

Además de todas las expresiones de funciones regulares que hemos visto hasta ahora, las funciones especiales se pueden definir dentro de los objetos conocidos como getters y setters. Como su nombre indica, nos permitieron obtener o establecer valores en un objeto.

Los getters y los setters son solo ‘azúcar sintáctico’: hacen que tu código sea un poco más fácil de entender desde el exterior, pero su funcionalidad también se puede lograr con funciones independientes.

Considera este ejemplo, donde almacenamos nombres de animales en nuestro objeto. Aquí, tenemos una función `get` - para obtener la propiedad `"value"` de nuestro objeto y un método establecido para agregar nuevos animales a esa propiedad de `value`. Dentro de un objeto, esto se refiere al objeto en sí. Esto también es cierto para las funciones normales establecidas dentro de los objetos:

```
let Animals = {
  value: [ 'dog', 'cat' ],
  get listAnimals() {
    return this.value
  },
  set newAnimal(name) {
    this.value.push(name)
    console.log("New animal added: " + name)
  }
}
Animals.newAnimal = "sheep"
console.log(Animals.listAnimals)
```

Lo interesante de getters y setters es que no los llamas usando los dobles corchetes como `Animals.newAnimal()`. En cambio, se llaman cada vez que verifica el nombre de la propiedad. Entonces `Animals.newAnimal = "sheep"` se usa en lugar de `Animals.newAnimal("sheep")`. Del mismo modo, los `Animals.listAnimals` enumerarán todos los animales, sin necesidad de ejecutar la función.

Esto les da la apariencia de ser propiedades normales, con el beneficio adicional de poder ejecutar código funcional.

Funciones del generador

El tipo de función final que veremos se conoce como la función del generador.

Anteriormente, cuando revisamos la iteración del arreglo, mencionamos cómo podríamos crear un iterador con acceso al método `next()` accediendo al protocolo `iterator`:

```
let myArray = [ "lightning", "apple", "squid", "speaker" ]
let getIterator = myArray[Symbol.iterator]()
console.log(getIterator.next()) // { value: 'lightning', done: false }
```

Las funciones del generador funcionan de manera similar y se denotan por función*. Una función de generador simple se ve así:

```
function* someGenerator(x) {
  yield x;
}
const runG = generator(1)
console.log(runG.next()) // { value: 1, done: false }
console.log(runG.next()) // { value: undefined, done: true }
```

Cada vez que usas el rendimiento en una función de generador, representa un punto de detención para `next()`. Puedes pensar en el rendimiento como la versión de la función del generador de `return`. Dado que solo usamos el rendimiento una vez en esta función, `next()` solo funcionará una vez. Después de eso, se marcará como se hace como se muestra en el ejemplo anterior. Como tal, las funciones del generador recuerdan dónde la dejaste y continúa desde ese punto.

En el siguiente ejemplo, ejecutamos un ciclo infinito, que también aumenta el valor de una variable llamada índice (`index`) cada vez. La función recuerda el valor del índice, cada vez que ejecutamos a continuación, lo que nos permite acceder al siguiente cálculo en la secuencia cada vez:

```
function* someGenerator(x) {
  let index = 0
  while(true) {
    yield x * 10 * index
    ++index
  }
}
```

```
const runG = someGenerator(5)
console.log(runG.next()) // { value: 0, done: false }
console.log(runG.next()) // { value: 50, done: false }
console.log(runG.next()) // { value: 100, done: false }
console.log(runG.next()) // { value: 150, done: false }
```

Si intentamos usar `return` en una función de generador, el estado "done" se establecerá en `true` y el generador dejará de funcionar:

```
function* someGenerator(x) {
  let index = 0
  while(true) {
    yield x * 10 * index
    return 5
  }
}
const runG = someGenerator(5)
console.log(runG.next()) // { value: 0, done: false }
console.log(runG.next()) // { value: 5, done: true }
console.log(runG.next()) // { value: undefined, done: true }
```

Si necesitas usar otra función de generador dentro de una función de generador, puedes diferir el rendimiento a esa nueva función de generador. En esos escenarios, usamos `yield*`. Por ejemplo, el `yield* myFunc()` tomaría el valor de rendimiento de `myFunc()` y lo usaría en la función actual.

Clases

JavaScript es un lenguaje prototípico, y como ya hemos visto, la herencia ocurre a través de prototipos. Muchos otros lenguajes tienen clases, lo que puede hacer que JavaScript parezca una gran salida en la sintaxis.

Para aliviar este problema de falta de familiaridad, JavaScript implementó clases. Sin embargo, en su mayor parte, las clases en JavaScript son básicamente solo 'azúcar sintáctica' para escribir funciones del constructor, con algunas capacidades adicionales específicas para las clases.

Las clases no tienen que usarse para escribir JavaScript, ya que proporcionan poco en la forma de una nueva funcionalidad, pero se están volviendo más comunes a medida que JavaScript hereda más desarrolladores que están acostumbrados a escribir software basado en clases. Las clases siempre deben llamarse con la palabra clave `new`, y siempre se ejecutan

en modo estricto, por lo que crean su propio contexto de forma predeterminada sin la necesidad de modo estricto.

Clases y funciones del constructor

Las clases pueden tener una función del constructor, que es la función que se ejecuta en cualquier momento que se les llame, pero no necesitan tener una. En las clases, las funciones del constructor se definen utilizando la palabra clave *constructor*, y solo puede tener una función de constructor por clase. Se comportan como funciones normales, y cualquier argumento pasado a la clase irá a la función del constructor, como se muestra en el siguiente ejemplo:

```
let myClass = class {  
  constructor(name) {  
    console.log(name)  
  }  
}  
new myClass("hello") // Console logs "hello"
```

Para el contexto, este ejemplo es el equivalente a escribir el siguiente código funcional:

```
let myFunction = function(name) {  
  console.log(name)  
}  
new myFunction("hello") // Console logs "hello"
```

Las clases se pueden escribir de dos maneras, ya sea donde usamos una variable `let` o `const` para definir la clase:

```
let myClass = class { // ...
```

O usando la palabra clave `class` seguida del nombre de la clase:

```
class myClass { // ...
```

Métodos de clase

Al igual que cuando declaramos funciones utilizando la palabra clave `new`, también podemos definir métodos en las clases. Estos se definen directamente en el cuerpo de la clase y funcionan de la misma manera. Como ejemplo de cómo funciona esto en la práctica, creemos una clase llamada `HotSauce`, con un par de propiedades y un método para recuperar cuán caliente es la salsa.

Hay algunas cosas que vale la pena señalar aquí:

1. En el nivel superior del cuerpo de la clase, las variables se pueden definir dentro de una clase sin palabra clave variable.
Eso es porque actúan como las propiedades de un objeto. Las variables en el nivel superior de la clase unen el contexto de la clase. En este ejemplo, asignamos unidades y maxHotness al contexto de la clase, por lo que se puede acceder a través de esto en los métodos.
2. Los métodos se escriben como method() en lugar de function method().
3. Las clases no tienen argumentos, pero puedes llamar a una clase con argumentos. Los argumentos sobre el constructor son donde se transmiten los argumentos utilizados en la clase.
4. Las clases finalmente crean objetos, al igual que las funciones.
Si intentas console.log new HotSauce ('Chilli Wave', 4600), obtendrás un objeto, como se muestra en la figura 5.
5. Finalmente, los argumentos pasados a una clase no están disponibles de forma predeterminada para toda la clase. Como tal, es bastante común ver el código que toma argumentos de la función del constructor y los asigna a this.
Eso es lo que hacemos a continuación para el nombre y el grado 'picoso' de la salsa picante definida:

```
let HotSauce = class {
  // Fields here are added to this, so they are available
  // via this.units and this.maxHotness in methods
  units = 'scoville'
  maxHotness = 20000000
  constructor(name, hotness) {
    // We can assign arguments from new instances of
    // our class to this as well
    this.hotness = hotness
    this.name = name
  }
  getName() {
    if(this.hotness < this.maxHotness) {
      return `${this.name} is ${this.hotness}
        ${this.units}`
    }
    else {
      return `${this.name} is too hot!`
    }
  }
}
```



```
let newSauce = new HotSauce('Chilli Wave', 4600)
// Console logs 'Chilli Wave is 4600 scoville
scovilles'
console.log(newSauce.getName())
```

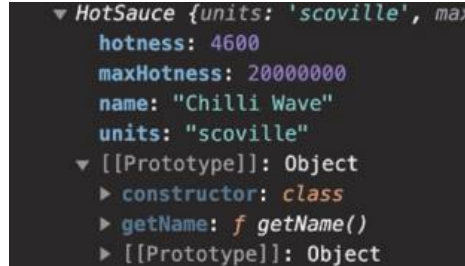


Figura 5: Las clases crean objetos. Si registramos la instancia que creamos de nuestra clase HotSauce, newSauce, obtendremos un objeto que contiene todas las propiedades que definimos, junto con un prototipo que contiene nuestros métodos.

Tipos de métodos de clase

Por defecto, todos los métodos de clase son públicos. Eso significa que pueden cambiarse fuera de la clase. También aparecerán en el registro de la consola e incluso se pueden eliminar. Por ejemplo, podríamos reescribir totalmente nuestro método getName si quisiéramos, fuera de la clase:

```
let newSauce = new HotSauce('Chilli Wave', 4600)
newSauce.getName = function() {
  return "No Hot Sauce for you!"
}
// No Hot Sauce for you!
console.log(newSauce.getName())
```

A veces, no quieres una clase editable como esta. Como tal, JavaScript proporciona otros dos tipos de campos que podemos usar con clases:


- **Campos estáticos**, a los que no se puede acceder en una nueva instancia de una clase en sí, pero solo en la clase original (que veremos con más detalle pronto).
- **Campos privados**, a los que solo se puede acceder desde la clase misma.

Campos de método estático

Los métodos y campos estáticos generalmente se usan para funciones de utilidad. Considera esta clase, donde un método estático simple devolverá algunos detalles sobre la clase:

```
let Utility = class {  
  static className = "Utility Functions"  
  static author = "Some Author"  
  
  static classDetails() {  
    return `${this.className} by ${this.author}`  
  }  
}
```

Si intentamos inicializar esta clase y llamar a `classDetails()`, recibiremos un error de tipo ya que no se puede llamar a `classDetails` en una nueva instancia de utilidad.



```
let callUtility = new Utility  
console.log(callUtility.classDetails())
```

En cambio, necesitamos llamar al método estático directamente en la clase de utilidad como se muestra en el siguiente ejemplo:

```
// Utility Functions by Some Author  
console.log(Utility.classDetails())
```

Dado que no iniciamos una nueva instancia de nuestra clase, los métodos estáticos tampoco tendrán acceso a propiedades no estáticas en la clase. Por ejemplo, si no hubiéramos llamado a `className` y al autor estático en el ejemplo anterior, entonces habrían sido indefinidos cuando intentamos hacer referencia a `classDetails()`.

Al igual que sus homólogos públicos, los métodos estáticos se pueden eliminar o cambiar en la clase misma. Por ejemplo, podríamos redefinir `classDetails()` a algo completamente diferente:

```
Utility.classDetails = function() {  
  return "Some return text"  
}
```

Los métodos estáticos también pueden ser getters y setters. Si cambiamos nuestra función de `classDetails()` de `static classDetails()` a `static get classDetails()`, podríamos llamarlo como `Utility.classDetails`. Lo mismo se aplica a los setters.

Finalmente, podemos crear inicializadores estáticos para ejecutarse tras la inicialización de la clase.

Esto nos permite ejecutar alguna funcionalidad al llamar a las propiedades o métodos estáticos, aunque los métodos estáticos no pueden aceptar argumentos. En el siguiente ejemplo, actualizamos la variable estática del autor a "Hello World" tras la inicialización de la clase:

```
let Utility = class {
  static className = "Utility Functions"
  static author = "Some Author"
  static {
    this.author = "Hello World"
  }
  static classDetails() {
    return `${this.className} by ${this.author}`
  }
}
Utility.classDetails() // Utility Functions by Hello World
```

Campos de método privado

Los campos de métodos privados no están disponibles fuera de la clase misma. Si bien los constructores en las clases no pueden ser privados, todo lo demás puede ser. El siguiente ejemplo muestra un campo privado definido dentro de una clase. Los campos privados se definen con un hash al comienzo:

```
let myClass = class {
  #privateField = 1
}
let newClass = new myClass()
```

Si intentamos con la consola `log newClass`, `privateField` no estará disponible. Además, tratar de referirse a `newClass.#PrivateField` lanzará un error de sintaxis.

Nota: Si bien los campos privados arrojarán errores en código si se hace referencia fuera de una clase, pueden mostrarse en la consola en algunos navegadores. Esto se debe a que puede ser útil en los registros de consola y la depuración para poder acceder a estos campos.

Si bien todas las demás características de las clases son solo características que ya tenían objetos, los campos privados son en realidad una nueva funcionalidad que solo están disponibles a través de clases. Los campos privados no están disponibles en objetos de la misma manera que en las clases.

La herencia de clase a través de extends

Dado que las clases son en su mayoría de ‘azúcar sintáctica’ en la parte superior de los objetos, también tienen una herencia prototípica típica. Si creas una nueva clase que extiende otra clase, la clase que se extiende simplemente se convertirá en el prototipo de la clase hijo. Considera el siguiente ejemplo, usando nuestra clase HotSauce:

```
class HotSauce {
  // Fields here are added to this, so they are available
  // via this.units and this.maxHotness in methods
  units = 'scoville'
  maxHotness = 20000000
  constructor(name, hotness) {
    // We can assign arguments from new instances of
    // our class to this as well
    this.hotness = hotness
    this.name = name
  }
  getName() {
    if(this.hotness < this.maxHotness) {
      return `${this.name} is ${this.hotness}
        ${this.units}`
    }
    else {
      return `${this.name} is too hot!`
    }
  }
}
```

De la figura 5, verás que el prototipo de esta clase contiene los métodos getName() y el constructor. Para explicar aún más este concepto, creemos una nueva clase que extiende esto, llamada VeganHotSauce.

Esto lógicamente tiene sentido ya que solo algunas salsas calientes son veganas:

```
class VeganHotSauce extends HotSauce {
  constructor(meat, name, hotness) {
    super(name, hotness)
    this.meat = meat
  }
  checkMeatContent() {
    if(this.meat) {
      return "this is not vegan"
    }
    else {
      return "no meat detected"
    }
  }
}
```

```

    }
  }
}
let newVeganOption = new VeganHotSauce(false, "VeganLite", 2400)
console.log(newVeganOption)

```

Aquí vale la pena señalar algunas cosas:

- Estamos presentando la función `super()` por primera vez. Esta es una función especial que llama a la clase principal, junto con su constructor. Aquí, esto es lo mismo que ejecutar `new HotSauce (name, hotness)`. Esencialmente, esto significa que estamos ejecutando el constructor `HotSauce` en cualquier momento que se ejecute `VeganHotSauce`, al tiempo que se establece `this.meat` a la variable `meat` de `VeganHotSauce`.
- Usando esta palabra clave, nuestro nuevo constructor para `VeganHotSauce` puede transmitir argumentos a la clase principal. Aquí, podemos pasar el `name` y el `hotness`.
- -El resultado de ejecutar esta clase, se puede ver en la figura 6.



Figura 6-6. `VeganHotSauce` contiene todas las propiedades de nivel superior de `HotSauce` en el nivel superior también. Los métodos para `VeganHotSauce` están contenidos dentro de su prototipo, y los métodos de `HotSauce` están en el prototipo de su prototipo.

Como es de esperar, los métodos estáticos y privados no se heredan cuando se usan extensiones. Eso es porque ambos, conceptualmente, pertenecen a la clase principal.

Herencia de clase con la palabra clave `super`

Utilizamos la palabra clave `super` en el ejemplo `VeganHotSauce` para llamar a su clase padre. La palabra clave `super` es especial en JavaScript, ya que se comporta de manera diferente

dependiendo de dónde la pongas. Si colocas `super` en un constructor para una clase extendida, se puede usar como una función, por lo que `super(... argumentos)` es completamente válido.

Mientras tanto, fuera de los constructores, solo puedes usar `super` en la forma `super[field]` o `super.field` para recurrir a las propiedades de los padres. De hecho, si intentas llamarlo como una función fuera del constructor, se lanzará un error. Eso significa que, en los métodos de una clase, podemos usar `super` solo para referir las propiedades de la clase principal. Entonces, por ejemplo, si quisiéramos usar el método de nuestros padres, `getName()`, en nuestra clase hijo `VeganHotSauce`, entonces llamaríamos a `super.getName()`:

```
class VegenHotSauce extends HotSauce {
  constructor(meat, name, hotness) {
    super(name, hotness)
    this.meat = meat
  }
  checkMeatContent() {
    console.log()
    if(this.meat) {
      return "this is not vegan.. but " + super.getName()
    }
    else {
      return "no meat detected.. and " + super.getName()
    }
  }
}

let newVeganOption = new VegenHotSauce(false, "VeganLite", 2400)
// console logs
// no meat detected.. and Vegan Lite is 2400 scovilles
console.log(newVeganOption.checkMeatContent())
```

Vale la pena señalar que si bien podemos llamar a `super.getName()`, no podemos llamar a `super.units`, y eso se debe a que las unidades no están en el prototipo de `VeganHotSauce`. Echa un vistazo a la figura 6 nuevamente, y verás por qué.

Aunque las unidades son una propiedad principal, cuando se hereda, aparece en la clase `VeganHotSauce`. La palabra clave `super` se utiliza para acceder a los métodos y propiedades prototipo de la clase principal. Si queremos acceder a unidades, tenemos que usar `this.units` en lugar de `super.units`.

Resumen

En este documento, hemos cubierto bastante terreno sobre funciones y clases. Te hemos mostrado cómo crear funciones y cómo heredan en función de la herencia prototípica. Hemos hablado sobre cómo dar a las funciones contexto con la palabra clave `this`. También hemos analizado las diferentes formas en que puedes definir funciones en tu código y cómo difieren.

Finalmente, hemos visto clases y explicamos cómo son en su mayoría ‘azúcar sintáctica’ para objetos. Hemos entrado en detalles utilizando ejemplos sobre cómo funciona la herencia de clase en JavaScript. Si bien las clases pueden ser familiares para algunos desarrolladores, si te resulta difícil entender o no intuitivo, todavía está bien trabajar con funciones y herencia prototípica cuando sea necesario.

Si decides usar clases o funciones es principalmente estilístico.

En JavaScript, muchos desarrolladores usan funciones ya que esa es la forma en que el lenguaje se ideó originalmente. Como se mencionó anteriormente, lo más importante es que mantengas la consistencia en tu código y que tu equipo comprenda lo que significa el código que estás escribiendo.