
Introducción a los objetos, arreglos

Ahora que ahora hemos analizado los conceptos fundamentales de JavaScript, pasemos a uno de los tipos de datos más importantes en JavaScript, que son los objetos. Los objetos se usan ampliamente en JavaScript como un almacén de datos. Difieren de otros tipos de datos en que son el único tipo de datos que son mutables.

En este documento, cubriremos los conceptos básicos de cómo funcionan los objetos. También cubriremos arreglos, que son un tipo especial de objeto con características específicas. A medida que nos sentimos más cómodos con los objetos y los arreglos, veremos algunos conceptos más complicados, como la programación basada en prototipos y la herencia de objetos.

Arreglos

Para comenzar este documento, veamos los arreglos. Los arreglos son un tipo especial de objeto que se utilizan para almacenar datos unidimensionales. No tienen llaves explícitamente definidas. Los arreglos generalmente se definen dentro de [] paréntesis cuadrados que son diferentes de los objetos, que cubriremos con más detalle más adelante, ya que usan {}. En el siguiente ejemplo, creamos un arreglo que contiene una mezcla de diferentes tipos de datos, incluido un objeto:

```
let myArr = [ "one", 2, "three", { "value": "four" } ]
```

La notación de [] anterior que utilizamos es un tipo de “azúcar sintáctico”. El azúcar sintáctico es otro nombre para una forma más limpia y agradable de escribir algo que tenemos que escribir todo el tiempo. Debajo de los soportes cuadrados hay un “constructor” llamado array. El ejemplo anterior se puede reescribir de esta manera utilizando el constructor de Array:

```
let myArr = new Array("one", 2, "three", { "value": "four" })
```

La palabra clave *new* le dice a JavaScript que cree una nueva instancia de un objeto. Eso es también lo que la mano corta de los paréntesis cuadrados también hace. Como creamos arreglos todo el tiempo en JavaScript, es mucho más común usar la notación cuadrada []. La notación de paréntesis cuadrado se llama formalmente el “**array literal**”.

Un arreglo puede contener cualquier tipo de datos, incluso cosas como funciones.

Si bien los arreglos no tienen llaves explícitamente definidas, tienen índices numerados. Cada arreglo es cero indexado, por lo que, para acceder al primer elemento, usamos la llave que es 0, y la segunda es 1 y otras. Para acceder a elementos de arreglos específicos, usamos paréntesis cuadrados nuevamente. El siguiente ejemplo te muestra cómo funciona:

```
let myArr = [ "banana", "apple", "squid", "cake", "pear" ]
console.log(myArr[0]) // shows "banana"
console.log(myArr[2]) // shows "apple"
console.log(myArr[3]) // shows "squid"
```

Obtener la longitud de un arreglo

Cuando le pedimos a JavaScript que cree una nueva instancia de un arreglo, también hizo que el arreglo heredara muchos métodos y propiedades estándar. Todos estos métodos y propiedades tienen utilidades específicas.

Un ejemplo de estos es la propiedad de longitud, que se usa para obtener el tamaño de un arreglo. Se puede acceder a los métodos y propiedades en arreglos utilizando un punto, seguido del nombre de esa propiedad o método. Por ejemplo, para obtener el tamaño de un arreglo determinado, usamos el método `.length` directamente en ese arreglo.

Intentemos poner la longitud de `MyArray` en una nueva variable llamada `arrayLength`. Cuando usamos la `console log`, mostrará 4:

```
let myArr = [ "one", 2, "three", { "value": "four" } ]
let arrayLength = myArr.length
console.log(arrayLength) // shows 4
```

Obtener el último elemento de un arreglo

Conocer la longitud de un arreglo se usa con frecuencia en el proceso de obtener el último elemento de un arreglo. Dado que los arreglos están indexados por cero, una forma común de obtener el último elemento de un arreglo es usar el método `.length` con 1 restado de él. La razón por la que restamos 1 es porque si comienza a contar desde cero, el índice del último elemento siempre será 1 menos que el tamaño del arreglo:

```
let myArr = [ "one", 2, "three", "four" ]
let arrayLength = myArr.length
console.log(myArr[myArr.length - 1]) // shows "four"
```

La longitud del arreglo aquí es 4, por lo que restar 1 de ese número da como resultado 3, que es el índice del último elemento en el arreglo.

Si bien verás esta forma de obtener el último elemento de un arreglo en todas partes, un método más moderno y eficiente es usar el método de arreglo, `.at()`. Nuevamente, como la longitud, `.at()` está predefinido en todos los arreglos.

Nota sobre los métodos del constructor: Todos los arreglos heredan un conjunto estándar de métodos, que pueden ser útiles para realizar funciones comunes. Del mismo modo, todos los demás tipos principales también tienen métodos de constructor, como cadenas, números, booleanos y objetos.

El método `.at()` funciona mucho como recuperar un elemento del arreglo con paréntesis cuadrados. La principal diferencia es que, si usa números negativos, comienza a contar desde el extremo opuesto, entonces `.at(-1)` también obtiene el último elemento de un arreglo:

```
let myArr = [ "one", 2, "three", "four" ]
let arrayLength = myArr.length
console.log(myArr.at(-1)) // shows "four"
```

Nota: Los paréntesis cuadrados son una notación estándar en los arreglos de JavaScript, pero `at()` es una función. Todavía no hemos cubierto funciones en detalle, pero las funciones pueden tomar argumentos. En este caso, `-1` es el argumento que hemos utilizado.

Métodos de manipulación de arreglos

Ahora hemos analizado cómo podemos construir y crear arreglos, pero en el desarrollo de JavaScript, es común querer mutar o cambiar los arreglos. Para aprender cómo funciona esto, veamos algunos métodos que son particularmente útiles al usar arreglos. Dado que los objetos (y, por lo tanto, arreglos) son mutables, muchos de estos métodos cambian el valor del arreglo en sí.

Push y unshift

Cuando queremos agregar un elemento a un arreglo, `push` y `unshift`, hagamos eso al final y al inicio del arreglo, respectivamente. Agregar elementos al final de un arreglo siempre es más rápido. La razón por la que es más rápido es porque `unshift` debe reasignar la memoria para agregar elementos al comienzo del arreglo.

En el siguiente ejemplo, `push` se usa para agregar un elemento al final de cualquier arreglo:

```
let myArr = [ "banana", "apple", "squid", "cake", "pear" ]  
myArr.push("pear")  
console.log(myArr) // [ "banana", "apple", "squid", "cake", "pear" ]
```

En el siguiente ejemplo, unshift se usa para agregar algo al principio:

```
let myArr = [ "banana", "apple", "squid", "cake", "pear" ]  
myArr.unshift("pear")  
console.log(myArr) // [ "pear", "banana", "apple", "squid", "cake", "pear" ]
```

Pop y shift

Mientras que push y unshift te permiten agregar datos al final y el comienzo de tu arreglo, respectivamente, pop y shift te permiten hacer lo mismo pero para la eliminación. Para ilustrar esto, veamos un ejemplo. Si queremos eliminar el valor "pear" que agregamos al final de nuestro arreglo, simplemente necesitamos ejecutar pop() en él:

```
let myArr = [ "banana", "apple", "squid", "cake", "pear" ]  
myArr.pop()  
console.log(myArr) // [ "banana", "apple", "squid", "cake" ]
```

Del mismo modo, si queremos eliminar el "banana" desde el comienzo del arreglo, podemos usar shift():

```
let myArr = [ "banana", "apple", "squid", "cake" ]  
myArr.shift()  
console.log(myArr) // [ "apple", "squid", "cake" ]
```

Splice

Agregar y eliminar elementos desde el principio y el final de un arreglo es útil, pero la mayoría de las veces, queremos cambiar algo en medio de un arreglo. Para hacer esto, podemos usar otro método llamado splice(). A diferencia de los métodos que hemos visto hasta ahora, splice() toma un par de argumentos, aunque solo se requiere uno. Dado que solo se requiere un argumento, la sintaxis para el splice ('empalme') puede parecerse a cualquiera de las siguientes variaciones:

```
someArray.splice(start)  
someArray.splice(start, end)  
someArray.splice(start, end, item1)  
someArray.splice(start, end, item1, item2, item3 ... itemN)
```

Los argumentos que damos a splice proporcionan información a la función sobre lo que queremos hacer. Estos argumentos se definen en lo siguiente:

- **start (requerido):** Esta es la posición en el arreglo en la que deseas comenzar el 'empalme' (slice). Si es un número negativo, se contará desde el final del arreglo.
- **end (opcional):** Esto es cuántos elementos deseas eliminar. Si solo deseas insertar algo, configura esto en 0. Si no colocas un número aquí, entonces todos los elementos después de la posición de inicio se eliminarán.
- **item1 ... itemN (opcional):** Estos son elementos del arreglo que se insertarán después de la posición de inicio. Puedes agregar tantos como quieras aquí, todas las comas separadas.

Si solo usamos el primer argumento en empalme, cada elemento después de cierto punto en el arreglo se eliminará, como se muestra en el siguiente ejemplo:

```
let myArr = [ "banana", "apple", "squid", "cake" ]
myArr.splice(1)
console.log(myArr) // [ "banana", "cake" ]
```

Si definimos un valor final, podemos eliminar elementos en medio de un arreglo. Por ejemplo, eliminemos "apple" y "octopus":

```
let myArr = [ "banana", "apple", "squid", "cake" ]
myArr.splice(0, 2)
console.log(myArr) // [ "banana", "cake" ]
```

Finalmente, si usamos los argumentos después del inicio y el final, podemos agregar nuevos elementos en partes específicas de nuestro arreglo. Agreguemos "strawberry" y "box" entre "banana" y "cake", utilizando el tercer y cuarto argumentos.

Cualquier argumento dado después del argumento final se agregará al arreglo en la posición de inicio:

```
let myArr = [ "banana", "apple", "squid", "cake" ]
myArr.splice(0, 2)
console.log(myArr) // [ "banana", "cake" ]
myArr.splice(1, 0, "strawberry", "box")
console.log(myArr) // [ "banana", "strawberry", "box", "cake" ]
```

Objetos

Ahora que hemos cubierto arreglos, pasemos a los objetos. Los objetos son similares a los arreglos, pero difieren en que han definido las llaves. Los objetos son posiblemente lo más importante que puedes entender en JavaScript, y conducen a conceptos más avanzados sobre cómo escribimos y estructuramos nuestro código.

En esta sección, analizaremos cómo funcionan los objetos y cómo se usan más ampliamente en la herencia prototípica, que es el estilo de programación principal en JavaScript. Los objetos de JavaScript se parecen mucho a lo que se llaman “diccionarios” en otros lenguajes y consisten en pares de valor-llave definidos. Se pueden definir dentro de los {}, como se muestra en el siguiente ejemplo:

```
let myObject = {  
  "key": "value",  
  "someKey": 5,  
  "anotherKey" : true  
}
```

Este objeto tiene tres llaves y tres valores asignados a cada una de esas llaves. Las llaves tienen que ser únicas, pero los valores no lo hacen, y los objetos pueden ser tan grandes como quieras.

Los {}, se conocen como la notación “literal de objetos”, por lo que como era de esperar, es el equivalente a escribir un “nuevo objeto” (al igual que para los arreglos).

Sin embargo, dado que los objetos tienen llaves, si queremos usar el constructor para definir un objeto, necesitamos escribir todas las llaves y valores por separado y adjuntarlas al objeto, lo que hace que la notación literal del objeto sea infinitamente más fácil de usar.

En el siguiente ejemplo, creamos el mismo objeto que en nuestro ejemplo anterior, pero usamos el constructor de objeto en lugar del objeto literal:

```
// Los objetos definidos sin la notación literal de los objetos  
// son más difíciles de definir, ya que necesitamos definir cada  
// llave por separado.  
let myObject = new Object()  
myObject.key = "value"  
myObject.someKey = 5  
myObject.anotherKey = true
```

Puedes pensar en los objetos como bases de datos en miniatura de información, donde cada llave única hace referencia a algunos datos. Los objetos tienen estas características comunes:

1. Las llaves deben ser una cadena, número o símbolos (un identificador único especial en JavaScript).
2. Los valores pueden ser de cualquier tipo y contener cualquier dato (objetos, arreglos, números, cadenas, funciones, etc.)
3. Los objetos son reajustables (resizable) y pueden ser mutados (a diferencia de otros datos en JavaScript).

Recuerda: ¡Los arreglos también son objetos! Entonces, todas estas propiedades también se aplican a ellos.

Acceso a datos de los objetos

Tal como vimos con arreglos, podemos usar paréntesis cuadrados para acceder a los datos en un objeto. Puedes ver una ilustración de cómo funciona en el siguiente ejemplo. El resultado de este código también se puede ver en la figura 1.

```
let myObject = {  
  "key": "value",  
  "someKey": 5,  
  "anotherKey" : true  
}  
console.log(myObject["key"]) // shows 'value'
```



```
> console.log(myObject['key']) // shows 'value'  
value  
< undefined  
>
```

Figura 1: Usando la notación de paréntesis cuadrado, podemos acceder al valor de cualquier llave en un objeto en JavaScript.

La notación de punto versus paréntesis cuadrados con objetos

Otra forma de acceder a los valores de los objetos es usar el punto . en lugar de []. Esto se muestra en el siguiente ejemplo:

```
let myObject = {  
  "key": "value",  
  "someKey": 5,  
  "anotherKey" : true  
}  
console.log(myObject.key) // shows 'value'
```

Cuando usamos paréntesis cuadrados, debemos usar comillas si la llave es una cadena ("llave", no llave), pero lo mismo no es cierto para la notación de puntos.

Como ejemplo de esto, considera el siguiente código. Con los paréntesis cuadrados, la variable `keyName` se usa si omitimos las comillas.

Sin embargo, cuando se usa el dot (punto), JavaScript buscará una llave en `MyObject` llamada `KeyName`, que, por supuesto, devuelve indefinido. Por lo tanto, tanto los paréntesis cuadrados como la notación de puntos tienen diferentes utilidades al acceder a objetos:

```
let myObject = {  
  "key": "value",  
  "someKey": 5,  
  "anotherKey" : true  
}  
let keyName = "key"  
console.log(myObject[keyName]) // shows "value"  
console.log(myObject.keyName) // shows undefined
```

Puedes ver este código ejecutándose en la figura 2.



```
> let myObject = {  
  "key": "value",  
  "someKey": 5,  
  "anotherKey" : true  
}  
let keyName = "key"  
console.log(myObject[keyName]) // shows "value"  
console.log(myObject.keyName) // shows undefined  
value  
undefined
```

Figura 2: Usando la notación de paréntesis cuadrado, podemos acceder al valor de cualquier llave en un objeto en JavaScript.

Acceso a objetos: Mientras exploramos el acceso de objetos aquí, ya hemos usado objetos en los documentos 1 y 2. Cuando usamos `console.log`, la consola es el objeto y `log` es una propiedad en el objeto de la consola. Si intentas usar la consola en `console.log`, ¡podrás ver todo el objeto en tu consola!

Destrucción de objetos

Ahora hemos cubierto las muchas formas en que puedes acceder a los datos en un objeto. Otra forma útil de hacerlo es destruyendo el objeto. La destrucción de objetos funciona al permitirnos dividir el objeto en un conjunto de variables, cada una de las cuales se puede usar de forma independiente. Para ilustrar esto, veamos un ejemplo. Primero, creemos un objeto simple:

```
const myObj = {  
  z: 5,  
  y: 4,  
  x: 3  
}
```

Ahora podemos acceder a partes de este objeto destruyéndolas en variables, como se muestra en el siguiente ejemplo:

```
const myObj = {  
  z: 5,  
  y: 4,  
  x: 3  
}  
const { x, y } = myObj  
console.log(y) // 4
```

Esta es una forma útil de tomar un objeto y solo acceder a los bits que necesitas a través de variables. Es particularmente común al desenvolver (unwrapping) los paquetes externos en Node.js, pero también se usa ampliamente en el cliente front-end JavaScript. La destrucción puede incluso dar valores predeterminados a los indefinidos. Si se encuentra que un valor está indefinido cuando se destruye, el valor predeterminado se usará en su lugar:

```
const myObj = {  
  z: undefined,  
  y: 4,  
  x: 3  
}  
const { z = 5 } = myObj  
console.log(z) // 5
```

Los nombres de variables que usas al destruir deben coincidir con los nombres de la propiedad, a menos que estés destruyendo un arreglo. En ese caso, puedes llamar a sus variables cualquier cosa que desees:

```
const [a, b] = [1, 2]
console.log(a) // 1
```

Puedes desenvolver un conjunto de objetos juntos usando el operador de tres puntos, como se muestra en el siguiente ejemplo:

```
const myObj = {
  z: undefined,
  y: 4,
  x: 3
}
const { x, ...rest } = myObj
// Only shows z and y: { z: undefined, y: 4 }
console.log(rest)
```

Mutabilidad del objeto

Como hemos discutido anteriormente, los objetos son mutables incluso si están dentro de una variable `const`. Podemos actualizar cualquier llave en un objeto a otra cosa usando un signo igual para establecerlo en algo completamente diferente:

```
let myObject = {
  "key": "value",
  "someKey": 5,
  "anotherKey": true
}
// Let's update one of the keys on myObject
myObject["key"] = "NEW VALUE"
console.log(myObject["key"]) // shows 'NEW VALUE'
```

Agregar nuevas llaves también se puede lograr definiéndolas directamente en el código, como se muestra en el siguiente ejemplo:

```
let myObject = {
  "key": "value",
  "someKey": 5,
  "anotherKey": true
}
// Let's update one of the keys on myObject
myObject["aNewKey"] = "Some Value"
console.log(myObject["aNewKey"]) // shows 'Some Value'
```

Los objetos también pueden contener otros objetos dentro de ellos, y se puede acceder a estos mediante múltiples paréntesis cuadrados o utilizando la notación de puntos si lo prefieres:

```
let myObject = {  
  "key": {  
    "key" : 5,  
    "newKey" : "value"  
  },  
  "someKey": 5,  
  "anotherKey" : true  
}  
console.log(myObject['key']['newKey']) // shows 'value'  
console.log(myObject.key.newKey) // shows 'value'
```

Objetos no mutables

La única vez que cambia la mutabilidad de un objeto es si lo cambias tú mismo. Todos los objetos tienen tres propiedades ocultas que configuran su mutabilidad junto con su valor:

- *Writable* - True si el valor de la propiedad se puede cambiar, false si es de solo lectura.
- *Enumerable* - True si la propiedad se mostrará en ciclos, false si no lo será.
- *Configurable* - True si el valor de la propiedad se puede eliminar o modificar, false si no puede.

Por defecto, todas estas propiedades son verdaderas. Existen algunos métodos de utilidad útiles para cambiar estas propiedades. Por ejemplo, puedes cambiar un objeto a solo lectura usando `Object.Free`, después de lo cual ningún cambio de propiedad afectará al objeto.

En la práctica, todo esto se hace al establecer el objeto a `{Writable: False, configurable: false}`. Otra propiedad similar, `Object.Seal`, establecerá un objeto a `{configurable: false}`. Esto significa que las propiedades existentes se pueden cambiar, pero no eliminar, y las nuevas no se pueden agregar.

En la figura 3, puedes ver cómo el `Objet.freeze` evita que se modifique un objeto. No se lanzan errores, pero el cambio se ignora.

Mientras que en los ejemplos anteriores hemos aplicado `Objet.freeze` y `Objet.seal` a objetos completos, las propiedades individuales tienen sus propias propiedades enumerables, configurables y de escritura. Puedes configurar estos ajustes si defines una nueva propiedad

utilizando el método `DefineProperty` en su lugar. Un ejemplo de esto se muestra en la figura 4.

```
> let myObject = { "name" : "John" }  
Object.freeze(myObject)  
myObject["age"] = 5  
myObject["name"] = "Johnny"  
console.log(myObject)  
  
▶ {name: 'John'}
```

Figura 3: En esta imagen, `Objeto.freeze` significa que `MyObject` ya no se puede cambiar. Si hubiéramos usado `Object.seal`, entonces cambiando a través con `myObject["name"]` habría funcionado, mientras que intentar agregar la propiedad de la edad (`age`) aún habría fallado.

```
> const myObject = {}  
  
Object.defineProperty(myObject, "name", {  
  value: "John",  
  writable: false,  
})  
  
◀ ▶ {name: 'John'}
```

Figura 4: En el ejemplo anterior, creamos una nueva propiedad llamada `name`, que no es de escritura. Esto significa que la propiedad no se puede actualizar. Si la propiedad fuera un objeto en sí, aún podrías extenderla, ya que configurable de forma predeterminada a verdadero.

Extendiendo la sintaxis o los “tres puntos”

Hemos cubierto una variedad de métodos que pueden usarse para manipular arreglos y objetos. Otra forma útil de manipular objetos y arreglos de una manera específica es a través de un operador conocido como sintaxis de propagación o extensión (*spread syntax*) o tres puntos, que se usa ampliamente para cambiar arreglos y objetos. La sintaxis de propagación puede hacer cuatro cosas:

1. Fusionar arreglos
2. Fusionar objetos
3. Coaccionar arreglos en objetos
4. Pasar matrices como argumentos a las funciones

Usando la sintaxis de propagación, puedes fusionar fácilmente dos arreglos, como se muestra en el siguiente ejemplo:

```
let animals1 = [ "cats", "dogs" ]  
let animals2 = [ "pigeons" ]  
let allAnimals = [ ...animals1, ...animals2 ]
```

```
console.log(allAnimals) // [ "cats", "dogs", "pigeons" ]
```

Los objetos también se pueden fusionar de la misma manera, pero si se encuentran llaves duplicadas, el segundo objeto sobrescribirá el primero:

```
let user1 = { "name" : "John", age: 24 }  
let user2 = { "name" : "Joe" }  
let combineUsers = { ...user1, ...user2 }  
console.log(combineUsers) // { "name" : "Joe", age: 24 }
```

El uso de la sintaxis extendida en un arreglo dentro de un objeto literal también lo convertirá en un objeto, donde las llaves son los índices del arreglo. Esto proporciona una forma simple de convertir un arreglo a un objeto:

```
let animals = [ "cats", "dogs"]  
let objectAnimals = { ...animals }  
console.log(objectAnimals) // {0: 'cats', 1: 'dogs'}
```

Programación basada en prototipos

Cuando comenzamos este tercer documento, mencionamos que todos los arreglos tienen el método `.at()`. La razón de esto es que los arreglos se crean iniciando una nueva instancia de un arreglo a través de un `new Array()`. Cuando se realiza una nueva instancia de un arreglo, hereda todas las propiedades y métodos del objeto `Array`, que incluye `.at()`.

La herencia en JavaScript ocurre a través de algo que llamamos prototipos, una parte especial de todos los objetos que permite la herencia. Mientras que otros lenguajes usan clases y programación orientada a objetos, el paradigma más común en JavaScript es el uso de prototipos. Este tipo de programación se llama **programación basada en prototipos**.

Todos los objetos en JavaScript tienen un prototipo, incluidos los arreglos. Puedes ver que si intentas usar en la console log cualquier objeto. En la figura 5, un console log de un objeto que acabamos de crear muestra la propiedad `[[Prototype]]`.



```
> let myObject = { "name" : "John" }  
console.log(myObject)  
▼ {name: 'John'} ⓘ  
  name: "John"  
  ► [[Prototype]]: Object
```

Figura 5: Todos los objetos en JavaScript tienen una propiedad prototipo.

Ya mencionamos que todos los arreglos “heredan” métodos estándar, como `.at ()`:

```
let myArray = [ "one", 2, "three", "four" ]
let arrayLength = myArray.length
console.log(myArray.at(-1)) // shows 'four'
```

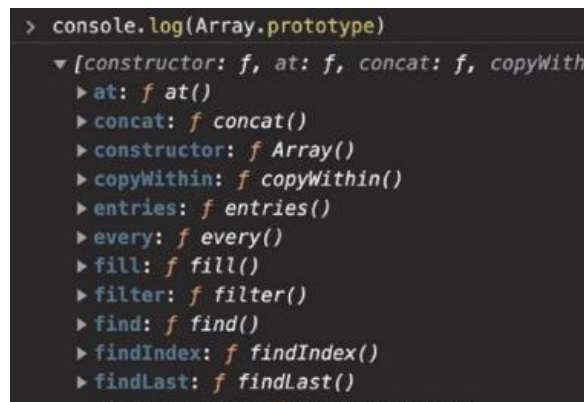
Pero, ¿cómo “heredan” el método `.at ()`? Bueno, primero, es importante recordar que dijimos que los paréntesis cuadrados actúan como “azúcar sintáctica” para el constructor del arreglo:

```
let myArray = new Array("one", 2, "three", "four")
```

Aquí, le estamos diciendo a JavaScript que queremos hacer una nueva instancia del objeto `Array`. Dado que el arreglo es solo un objeto predefinido en JavaScript, ya tiene su propio prototipo. Cuando le decimos a JavaScript que haga una nueva instancia de arreglo, se necesita cualquier JavaScript que define un objeto “`Array`” y hace una copia nueva, incluidos cualquier método, prototipos y propiedades.

En este prototipo es donde encontramos propiedades como `.length`, y métodos como `.at ()`. Esto no es muy obvio cuando se usa la notación literal de arreglo de paréntesis cuadrado, pero es mucho más claro cuando usas un nuevo arreglo.

Antes de profundizar, probemos esto. Puedes usar `console.log ()` en `Array.prototype` para encontrar todos los métodos que existen en el objeto del arreglo, que son heredados por todos los arreglos. En la parte superior, verás `.at ()`, pero hay mucho más. Se puede acceder a todos estos métodos en todos los arreglos, ya que todos los arreglos se crean utilizando nuevos `Array` o paréntesis cuadrados (que es lo mismo que el nuevo arreglo). Un ejemplo de la respuesta al registro de la consola se puede ver en la figura 6.



```
> console.log(Array.prototype)
▼ [constructor: f, at: f, concat: f, copyWith
  ▶ at: f at()
  ▶ concat: f concat()
  ▶ constructor: f Array()
  ▶ copyWithin: f copyWithin()
  ▶ entries: f entries()
  ▶ every: f every()
  ▶ fill: f fill()
  ▶ filter: f filter()
  ▶ find: f find()
  ▶ findIndex: f findIndex()
  ▶ findLast: f findLast()
```

Figura 6: El registro de consola de cualquier objeto en JavaScript puede ser bastante revelador, y hacerlo en el arreglo nos muestra todos los métodos posibles que podemos usar en cada arreglo. ¡Recuerda este consejo si te confundes al escribir JavaScript!

Aunque hasta ahora hemos estado hablando de arreglos, lo mismo es cierto para todos los objetos. Dado que todos los objetos son nuevas instancias de un tipo de “objeto” estándar, puedes encontrar todos los métodos de objetos estándar utilizando `console.log (Objeto.prototype)`.

Herencia prototípica


Ahora nos estamos familiarizando relativamente con los objetos y los arreglos y, con suerte, obtenemos una comprensión de cómo los arreglos y los objetos obtienen sus métodos y propiedades. También sabemos cómo acceder a las propiedades del objeto. Por ejemplo, sabemos que, dado el siguiente código, podríamos acceder a la llave de propiedad escribiendo `myObject.Key`:

```
let myObject = {  
  "key": "value",  
  "someKey": {  
    "someOtherKey" : 5  
  },  
  "anotherKey" : true  
}
```

También vimos anteriormente que podríamos usar el método `Array.at ()` para obtener un elemento del arreglo en un índice específico:

```
let myArray = [ "one", 2, "three", "four" ]  
console.log(myArray.at(-1)) // shows 'four'
```

Esto puede parecer extraño, ya que este método existe en `Array.prototype`, no en `Array`. Entonces, dado lo que sabemos sobre cómo accedemos a los objetos, ¿no debería el código anterior ser `myArray.prototype.at (-1)`?

 `let myArray = ["one", 2, "three", "four"]
console.log(myArray.prototype.at(-1))`

Bueno, la respuesta es, por supuesto, no, y el código anterior realmente producirá un error. Para entender por qué, tenemos que aprender cómo JavaScript verifica las propiedades en un objeto, así también cómo JavaScript copia los prototipos en nuevas instancias. Cuando escribimos `myArray.at (-1)`, JavaScript hace lo siguiente:

1. Verifica el objeto, `myArray`, para la propiedad `at`.
2. Si no existe, verifica el prototipo del objeto `myArray` para la propiedad.

3. Si todavía no existe, verifica el prototipo del prototype del objeto myArray para la propiedad.
4. Sigue haciendo esto hasta que no existan más prototipos, momento en el que regresará indefinido.

Es por eso que myArray.at (-1) funciona, pero no explica exactamente por qué funciona myArray.prototype.at (-1). La razón de esto es que myArray.prototype no está definido. Si intentas registrar un nuevo arreglo, encontrarás una propiedad [[Prototype]], pero no hay propiedad prototype. Dado que los arreglos heredan estos métodos en una sección de prototipo especial y no en el objeto en sí, no podemos acceder a los métodos a través de myArray.prototype. Esto se puede ver en la figura 7.

```
> let myArray = [ 1, 2, 3, 4 ]
console.log(myArray)

▼ (4) [1, 2, 3, 4] ⓘ
  0: 1
  1: 2
  2: 3
  3: 4
  length: 4
  ► [[Prototype]]: Array(0)
```

Figura 7: Si bien un nuevo arreglo heredará todas las propiedades y métodos prototipo de su padre, no existen dentro de una propiedad prototipo. En cambio, se sientan en un área especial llamada [[Prototype]]

[[Prototype]] versus prototipo (y __proto__)

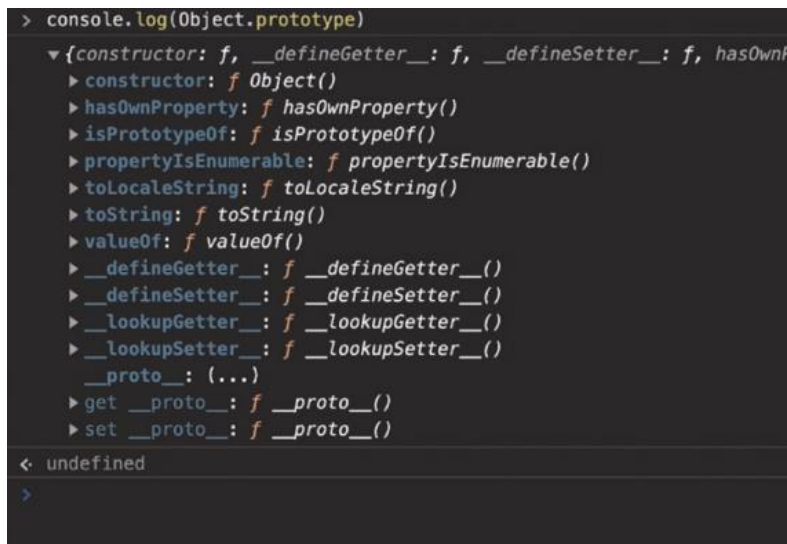
La razón por la cual myArray.prototype no está definido es porque los métodos y propiedades prototipo que se heredan existen en una propiedad especial y oculta llamada [[Prototype]].

Cuando creamos una nueva instancia del arreglo, apunta a [[Prototype]] de este nuevo arreglo al Array.prototype del constructor del Array.

Si bien eso está claro, también existe otra propiedad llamada __proto__ para confundir aún más los asuntos. Encontrarás esto al intentar usar console log con Objet.prototype. Es posible que hayas pensado: “Eso es extrañamente similar al prototipo”, y no estaría equivocado de que los dos estén relacionados.

La propiedad __proto__ es en realidad solo una forma de acceder a la [[Prototype]] subyacente de un objeto. Es una característica **desactivada** (deprecated), lo que significa que no debes usarla. Si la usas, es muy lenta, y dado que no es una parte estándar de la especificación de JavaScript, puedes encontrarte con algunos problemas inesperados. Un

registro de consola para `Object.prototype` donde puedes ver la propiedad `__proto__` se puede encontrar en la figura 8.



```
> console.log(Object.prototype)
▼ {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnP...
  ▶ constructor: f Object()
  ▶ hasOwnProperty: f hasOwnProperty()
  ▶ isPrototypeOf: f isPrototypeOf()
  ▶ propertyIsEnumerable: f propertyIsEnumerable()
  ▶ toLocaleString: f toLocaleString()
  ▶ toString: f toString()
  ▶ valueOf: f valueOf()
  ▶ __defineGetter__: f __defineGetter__()
  ▶ __defineSetter__: f __defineSetter__()
  ▶ __lookupGetter__: f __lookupGetter__()
  ▶ __lookupSetter__: f __lookupSetter__()
  ▶ __proto__: (...)
```

Figura 8: El prototipo de un objeto tiene una propiedad llamada `__proto__`, como se mostró anteriormente.

En resumen:

- La propiedad `prototype` que se encuentra en el arreglo y el objeto es un conjunto de todas las propiedades heredadas en arreglos y objetos `[[Prototype]]` s cuando hacemos nuevas instancias de ellas (con notación literal de arreglo u objeto, o `new Array/Object()`).
- El `[[Prototype]]` es una propiedad oculta, que existe en todos los objetos y se puede ver usando `console.log ()`. Cuando realizas una nueva instancia de un arreglo u objeto, el `[[Prototype]]` de esa nueva instancia apunta hacia la propiedad prototipo de `Array/Object`.
- La propiedad `__proto__` es una forma desactivada de acceder a los objetos `[[Prototype]]`. Está fuertemente desaconsejado y no se implementa lo mismo en cada navegador o motor JavaScript.

Objeto shallow y copias profundas

Cuando los datos se crean dentro de una variable, la variable apunta hacia esos datos, en lugar de contenerlos. Cuando hemos utilizado métodos de arreglo anteriormente en este documento, parecíamos modificar el arreglo original. Por ejemplo, el uso de `splice` en un arreglo cambia su valor cuando intentamos usarlo en `console log`:

```
let myArr = [ "banana", "apple", "squid", "cake" ]  
myArr.splice(0, 2)  
console.log(myArr) // [ "banana", "cake" ]
```

Esto parece tener mucho sentido. La variable myVar ha cambiado, por lo que, por supuesto, los datos originales deben ser diferentes. Sin embargo, las cosas comienzan a ponerse complicadas cuando pones el empalme en otra variable. Cuando hacemos eso, encontrarás que las dos variables tienen valores diferentes en el registro de la consola:

```
let myArr = [ "lightning", "search", "key", "bolt" ]  
let newArr = myArr.slice(2, 3);  
console.log(myArr); // [ "lightning", "search", "key", "bolt" ]  
console.log(newArr); // [ "key" ]
```

¿No debería una actualización de myArr usando splice hacer que el valor original cambie también para que tanto myArr como newArr sean los mismos? En la práctica, no funciona de esa manera, y la razón por la cual en realidad es otra peculiaridad de JavaScript.

JavaScript hace lo que se conoce como una “copia superficial” (shallow copy) de datos cuando usa cualquier método de arreglo. Eso significa que tienes dos “referencias”, que apuntan a los mismos datos subyacentes. Para visualizar esto, ver la figura 9.



Figura 9: En la imagen anterior, los “datos” (DATA) subyacentes son el “valor” (value). Entonces myArr y newArr son ambas referencias a esos datos.

Si bien los valores almacenados en myArr y newArr ahora son diferentes, todavía hacen referencia a los mismos datos subyacentes. Los datos en la referencia son diferentes ya que el método slice lo ha cambiado. Como almacenamos la salida slice en una variable, los datos subyacentes nunca cambiaron. Esto puede hacer que parezca que ambas variables funcionan de forma independiente, ¡pero no lo son!

Si intenta experimentar con esto, puedes confundirte aún más porque JavaScript solo actualiza el “valor” subyacente (es decir, los datos en la figura 4) en ciertas circunstancias. Por ejemplo, actualizar el valor de solo newArr, parece solo afectar a newArr:

```
let myArr = [ "lightning", "search", "key", "bolt" ]
let newArr = myArr.slice(2, 3)
newArr[2] = "lightning"
console.log(myArr) // [ "lightning", "search", "key", "bolt" ]
console.log(newArr) // [ "key", empty, "lightning" ]
```

Entonces, si ambos están trabajando desde los mismos datos debajo del capó (under the hood), ¿por qué no actualiza esto el valor subyacente de nuestros datos? Eso se debe a algo que también parece extraño, que es que, si usas paréntesis cuadrados específicamente, ¡solo asignas datos a la copia superficial! Si en su lugar usas la notación dot (.), como en situaciones en las que un objeto está en tu arreglo, tanto superficial como original cambiarán ya que cambiar una propiedad cambia el valor subyacente:

```
let myArr = [ { items: [ "search" ] }, "search", "key", "bolt" ]
let newArr = myArr.slice(0, 3)
// Update arrayOneSlice
newArr[0].items = [ "lightning" ]
console.log(myArr) // [ { items: [ "lightning" ] }, "search", "key", "bolt" ]
console.log(newArr); // [ { items: [ "lightning" ] }, "search", "key" ]
```

Puedes ver en el ejemplo anterior que cambiar newArr[0].items afecta tanto la copia superficial como la original. Es importante que tengas cuidado al usar métodos como slice, ya que podrías terminar cambiando tus datos de una manera que nunca esperabas.

La solución para evitar este problema es crear una “copia profunda” (deep copy) del original. Eso significa que se hace una referencia completamente nueva en la memoria para que ambas variables puedan funcionar de forma independiente. Esto es bastante costoso computacionalmente cuando se trabaja con objetos grandes, así que solo úsalo cuando lo necesites.

La forma recomendada de hacerlo es usar la función structuredClone:

```
let myArray = [ 1, 2, 3, 4 ];
let deepCopy = structuredClone(myArray);
```

Las copias profundas te dan certeza. En el ejemplo anterior, si aplicas algún método a deepCopy, solo afectará a deepCopy. La función, structuredClone, se puede usar en cualquier objeto o arreglo para asegurarte de que sea una copia profunda en lugar de una superficial.

Resumen

En este tercer documento, hemos cubierto los fundamentos de los objetos y los arreglos.

Te hemos mostrado cómo construir tus propios objetos y arreglos, y luego cómo mutarlos utilizando una variedad de métodos y operadores. Además de esto, hemos analizado cómo funciona la herencia prototípica y cómo los objetos forman la columna vertebral del paradigma de programación principal de JavaScript, la programación basada en prototipos.

También hemos analizado la variedad de formas en que puedes manipular o acceder a objetos y cómo puedes limitar el acceso a los objetos a través de propiedades ocultas como configurable. Finalmente, observamos algunas de las peculiaridades relacionadas con el trabajo con objetos, como copias profundas y superficiales.

En los documentos futuros, construiremos sobre los temas que hemos aprendido aquí presentando nuevos conceptos.