Objetos

Un objeto es una colección de datos relacionados y/o funcionalidad (que generalmente consta de algunas variables y funciones, que se denominan propiedades y métodos cuando están dentro de objetos). Vamos a trabajar a través de un ejemplo para mostrarte cómo son.

Al igual que con muchas cosas en JavaScript, la creación de un objeto a menudo comienza con la definición e iniciación de una variable.

```
var persona = {};
```

Si ingresas persona en tu entrada de texto y presionas el botón, debes obtener el siguiente resultado:

```
[objeto Objeto]
```

¡Felicidades!, acabas de crear tu primer objeto. ¡Trabajo realizado! Pero este es un objeto vacío, por lo que realmente no podemos hacer mucho con él. Actualicemos nuestro objeto para que se vea así:

```
var persona = {
nombre: ['Bob', 'Smith'],
edad: 32,
genero: 'masculino',
intereses: ['música', 'esquí'],
bio: function () {
        alert(this.nombre[0] + '' + this.nombre[1] + ' tiene ' + this.edad + ' años. Le gusta ' +  this.intereses[0] + ' y ' + this.intereses[1] + '.');
},
saludo: function() {
        alert('Hola, Soy '+ this.nombre[0] + '. ');
```

```
}
};
```

Después de guardar y actualizar, intenta ingresar algunos de los siguientes en tu entrada de texto:

persona.nombre

persona.nombre[0]

persona.edad

persona.intereses[1]

persona.bio()

persona.saludo()

¡Ahora tienes algunos datos y funcionalidades dentro de tu objeto, y ahora puedes acceder a ellos con una sintaxis simple y agradable!

Un objeto se compone de varios miembros, cada uno de los cuales tiene un nombre (por ejemplo, nombre y edad) y un valor (por ejemplo, ['Bob', 'Smith'] y 32). Cada par nombre/valor debe estar separado por una coma, y el nombre y el valor en cada caso están separados por dos puntos. La sintaxis siempre sigue este patrón:

```
var nombreObjeto = {
    miembro1Nombre: miembro1Valor,
    miembro2Nombre: miembro2Valor,
    miembro3Nombre: miembro3Valor
}
```

El valor de un miembro de un objeto puede ser prácticamente cualquier cosa: en nuestro objeto persona tenemos una cadena de texto, un número, dos arreglos y dos funciones. Los primeros cuatro elementos son elementos de datos y se denominan propiedades del objeto. Los dos últimos elementos son funciones que le permiten al objeto hacer algo con esos datos, y se les denomina métodos del objeto.

Un objeto como este se conoce como un objeto literal — literalmente hemos escrito el contenido del objeto tal como lo fuimos creando. Esto está en contraste con los objetos instanciados de las clases, que veremos más adelante.

Es muy común crear un objeto utilizando un objeto literal cuando deseas transferir una serie de elementos de datos relacionados y estructurados de alguna manera, por ejemplo, enviando una solicitud al servidor para ponerla en una base de datos. Enviar un solo objeto es mucho más eficiente que enviar varios elementos individualmente, y es más fácil de procesar que con un arreglo, cuando deseas identificar elementos individuales por nombre.

Notación de punto

Arriba, accediste a las propiedades y métodos del objeto usando notación de punto (dot notation). El nombre del objeto (persona) actúa como el espacio de nombre (namespace); al cual se debe ingresar primero para acceder a cualquier elemento encapsulado dentro del objeto. A continuación, escribe un punto y luego el elemento al que deseas acceder: puede ser el nombre de una simple propiedad, un elemento de una propiedad de arreglo o una llamada a uno de los métodos del objeto, por ejemplo:

```
persona.edad
```

```
persona.intereses[1]
```

```
persona.bio()
```

Espacios de nombres secundarios

Incluso es posible hacer que el valor de un miembro del objeto sea otro objeto. Por ejemplo, intenta cambiar el miembro nombre de

```
nombre: ['Bob', 'Smith'],
```

Al siguiente:

```
nombre : {
```

```
    pila: 'Bob',
    apellido: 'Smith'
},
```

Aquí estamos creando efectivamente un espacio de nombre secundario (sub-namespace). Esto suena complejo, pero en realidad no es así: para acceder a estos elementos sólo necesitas un paso adicional que es encadenar con otro punto al final. Prueba estos:

```
persona.nombre.pila
persona.nombre.apellido
```

Importante: en este punto, también deberás revisar tu código y cambiar cualquier instancia de:

```
nombre[0]
nombre[1]
```

Al siguiente:

```
nombre.pila
nombre.apellido
```

De lo contrario, sus métodos ya no funcionarán.

Notación de corchetes

Hay otra manera de acceder a las propiedades del objeto, usando la notación de corchetes. En lugar de usar estos:

```
persona.edad
persona.nombre.pila
```

Puedes usar:

```
persona['edad']
```

```
persona['nombre']['pila']
```

Esto se ve muy similar a cómo se accede a los elementos en un arreglo, y básicamente es lo mismo: en lugar de usar un número de índice para seleccionar un elemento, se está utilizando el nombre asociado con el valor de cada miembro. No es de extrañar que los objetos a veces se denominen arreglos asociativos: asocian cadenas de texto a valores de la misma manera que los arreglos asocian números a valores.

Establecer miembros de objetos

Hasta ahora solo hemos buscado recuperar (u obtener) miembros del objeto: también puede establecer (actualizar) el valor de los miembros del objeto simplemente declarando el miembro que deseas establecer (usando la notación de puntos o corchetes), de esta manera:

```
persona.edad = 45;
```

```
persona['nombre']['apellido'] = 'Cratchit';
```

Intenta ingresar estas líneas y luego vuelve a ver a los miembros para ver cómo han cambiado:

```
persona.edad
```

```
persona['nombre']['apellido']
```

Establecer miembros no solo es actualizar los valores de las propiedades y métodos existentes; también puedes crear miembros completamente nuevos. Prueba estos:

```
persona['ojos'] = 'avellana';
```

```
persona.despedida = function() { alert("¡Adiós a todos!"); }
```

Ahora puedes probar a los nuevos miembros:

```
persona['ojos']
```

```
person.despedida()
```

Un aspecto útil de la notación de corchetes es que se puede usar para establecer dinámicamente no solo los valores de los miembros, sino también los nombres de los

miembros. Digamos que queremos que los usuarios puedan almacenar tipos de valores personalizados en sus datos personales, escribiendo el nombre y el valor del miembro en dos entradas de texto. Podríamos obtener esos valores de esta manera:

```
var nombrePerzonalizado = entradaNombre.value;
var valorPerzonalizado = entradaValor.value;
```

Entonces podríamos agregar este nuevo miembro nombre y valor al objeto persona de esta manera:

```
persona[nombrePersonalizado] = valorPersonalizado;
```

Para probar esto, intenta agregar las siguientes líneas en tu código, justo debajo de la llave de cierre del objeto persona:

```
var nombrePersonalizado = 'altura';
var valorPersonalizado = '1.75m';
persona[nombrePersonalizado] = valorPersonalizado;
```

Ahora intenta guardar y actualizar, e ingresa lo siguiente en tu entrada de texto:

```
persona.altura
```

Agregar una propiedad a un objeto no es posible con la notación de puntos, que solo puede aceptar un nombre de miembro literal, no un valor variable que apunte a un nombre.

¡Has estado usando objetos todo el tiempo!

A medida que has estado repasando estos ejemplos, probablemente hayas pensado que la notación de puntos que has usado es muy familiar. ¡Eso es porque la has estado usando a lo largo del curso! Cada vez que hemos estado trabajando en un ejemplo que utiliza una API de navegador incorporada o un objeto JavaScript, hemos estado usando objetos, porque tales características se crean usando exactamente el mismo tipo de

estructuras de objetos que hemos estado viendo aquí, aunque más complejos que nuestros propios ejemplos personalizados.

Entonces cuando usaste métodos de cadenas de texto como:

```
myCadena.split(‘,’);
```

Estabas usando un método disponible en una instancia de la clase String. Cada vez que creas una cadena en tu código, esa cadena se crea automáticamente como una instancia de String, y por lo tanto tiene varios métodos/propiedades comunes disponibles en ella.

Cuando accediste al modelo de objetos del documento (document object model, o DOM) usando líneas como esta:

```
var miDiv = document.createElement(‘div’);
var miVideo = document.querySelector(‘video’);
```

Estaba usando métodos disponibles en una instancia de la clase Document. Para cada página web cargada, se crea una instancia de Document, llamada document, que representa la estructura, el contenido y otras características de la página entera, como su URL. De nuevo, esto significa que tiene varios métodos/propiedades comunes disponibles en él.

Lo mismo puede decirse de prácticamente cualquier otro Objeto/API incorporado que hayas estado utilizando: Array, Math, etc.

Ten en cuenta que los Objetos/API incorporados no siempre crean instancias de objetos automáticamente. Como ejemplo, la API de Notificaciones, que permite que los navegadores modernos activen las notificaciones del sistema, requiere que crees una instancia de un nuevo objeto para cada notificación que desees disparar. Intenta ingresar lo siguiente en tu consola de JavaScript:

```
var miNotificacion = new Notification(‘¡Hola!’);
```

Nota: Es útil pensar en la forma en que los objetos se comunican como paso de mensajes — cuando un objeto necesita otro objeto para realizar algún tipo de acción a menudo enviará un mensaje a otro objeto a través de uno de sus métodos, y esperará una respuesta, que conocemos como un valor de retorno.

Función constructura y new

JavaScript **no tiene una notación formal de clase** y recurre a las **funciones constructoras para este fin**. Mencionar también que JavaScript utiliza los **prototipos de los objetos para propagar la herencia**, algo que sin duda cuesta entender al principio y al que dedicaremos un artículo independiente más adelante.

Función constructora

Una función constructora es una función normal y corriente de JavaScript que se utiliza para definir una especie de plantilla para nuestros objetos personalizados. Veamos un ejemplo:

```
function Cliente(nombre, fecha, direccion) {
        this._nombre = nombre;
        this._fechaNacimiento = fecha;
        this._direccion = direccion;
}
```

Como podemos observar, se trata de una típica función de JavaScript que admite una serie de parámetros de entrada, aunque **estos no son obligatorios en absoluto**. La única particularidad de esta función es que se utiliza la palabra reservada **this** de JavaScript para definir una serie de propiedades (también podrán ser métodos) que formarán parte de nuestros objetos personalizados.

En la ilustración lateral vemos cómo podemos utilizar esta función constructora para crear instancias de nuestros objetos personalizados.

El operador new utilizado junto a una función de JavaScript es lo que nos permite obtener un objeto constructor o función constructora. Lo que sucede por debajo es que new primeramente crea un objeto sin propiedades y posteriormente llama a la función pasándole el nuevo objeto como valor de la palabra reservada this**.** Finalmente, la función nos devuelve un nuevo objeto con las propiedades y métodos definidos dentro de la constructora.

Como se aprecia en el intellisense de la imagen observamos que el nuevo objeto miCliente tiene todas las propiedades definidas anteriormente dentro del constructor.

Como hemos comentado, no es necesario que el constructor tome parámetros, podemos crear una plantilla en blanco e ir rellenando los objetos con datos cuando lo necesitemos:

```
// Constructor vacío
function Cliente() {
        this._nombre;
        this._fechaNacimiento;
        this._direccion;
}
// Creamos el objeto y le asignamos valores
var cliente = new Cliente();
cliente._nombre = "Cristina Rodriguez";
cliente._fechaNacimiento = new Date(1987, 3, 25);
cliente._direccion = "Plaza Bilbao 25";
```

Comprobar la función constructora de un objeto

Todos los objetos de JavaScript ya sean nativos o de usuario, tienen una **propiedad constructor** que heredan del objeto genérico Object, la cual hace referencia a la **función constructora que inicializa el objeto** lo que en principio (ahora veremos por qué digo esto) nos permite determinar la función constructora de un objeto, y casi por extensión, la clase de éste:

```
function Cliente() {
        // Definición de miembros de Cliente…
}
var unCliente = new Cliente();
if (unCliente.constructor == Cliente) {
        // Hacer algo con el objeto unCliente
}
```

Por otro lado, también podríamos **utilizar el operador instanceof** para determinar la constructora de un objeto, pero con algunas diferencias. El operador instanceof, a diferencia del anterior, comprueba la jerarquía del objeto, por lo tanto podríamos preguntar directamente sobre el objeto padre con idénticos resultados:

```
unCliente instanceof Cliente // es true
unCliente instanceof Object // es true
```

Lamentablemente en JavaScript nada es tan sencillo como parece. Las cosas se complican cuando hablamos de modificar el prototipo de un objeto y la **propiedad constructor** parece perder la referencia a la función constructora.

Uso del This para el público y privado

¿Qué es "this" (este)?

Es posible que hayas notado algo un poco extraño en nuestros métodos. Mira esto, por ejemplo:

```
saludo: function() {
        alert('¡Hola!, Soy '+ this.nombre.pila + '.');
}
```

Probablemente te estés preguntando qué es "this". La palabra clave this se refiere al objeto actual en el que se está escribiendo el código, por lo que en este caso this es equivalente a la persona.

Entonces, ¿por qué no escribir persona en su lugar? Como verás en el artículo JavaScript orientado a objetos para principiantes cuando comenzaremos a crear constructores, etc., this es muy útil: siempre asegurará que se usen los valores correctos cuando cambie el contexto de un miembro (por ejemplo, dos diferentes instancias de objetos persona) pueden tener diferentes nombres, pero querráx usar su propio nombre al decir su saludo).

Vamos a ilustrar lo que queremos decir con un par de objetos persona simplificados:

```
var persona1 = {
        nombre: 'Chris',
        saludo: function() {
         alert('¡Hola!, Soy '+ this.nombre + '.');
 }
}
var persona2 = {
        nombre: 'Brian',
```

```
    saludo: function() {
    alert('¡Hola!, Soy '+ this.nombre + '.');
}
}
```

En este caso, persona1.saludo() mostrará "¡Hola!, Soy Chris"; persona2.saludo() por otro lado mostrará "¡Hola!, Soy Brian", aunque el código del método es exactamente el mismo en cada caso. Como dijimos antes, this es igual al objeto en el que está el código; esto no es muy útil cuando se escriben objetos literales a mano, pero realmente se vuelve útil cuando se generan objetos dinámicamente (por ejemplo, usando constructores) Todo se aclarará más adelante.

this es un keyword de JavaScript que tiene un comportamiento muy diferente a otros lenguajes de programación, así para algunos es considerado uno de los grandes errores de diseño del lenguaje.

La clave para entender el comportamiento de this, es tener claro donde se invoca, para saber qué objeto le asigna.

¿Dónde se está invocando this?

Asignación implícita

Caso 1

En el primer caso this está siendo invocado dentro de un método.

```
let yo = {
    nombre: 'yeison',
    edad: 22,
    hablar: function () {
     console.log(this.nombre);
```

```
}
};
yo.hablar(); // yeison
```

this hace referencia al objeto, que contiene el método donde se invoca.

Caso 2

En este caso, existe una función que recibe un objeto como parámetro, y le agrega el método hablar, luego, se ejecuta la función sobre dos objetos.

```
let decirNombre = function(obj) {
        obj.hablar = function() {
        console.log(this.nombre);
};
};
const Mateo = {
        nombre: 'Mateo',
        edad: 22
};
const juan = {
        nombre: 'Juan',
        edad: 25
};
decirNombre(juan);
decirNombre(Mateo);
juan.hablar(); // Juan
Mateo.hablar(); // Mateo
```

This en este caso hace referencia al objeto que se añade este método.

Caso 3

En este caso tenemos una función que retorna un objeto, que contiene un método hablar, que invoca this.

```
let Persona = function (nombre, edad, madre) {
 return {
        nombre: nombre,
        edad: edad,
        hablar: function() {
                console.log(this.nombre);
        },
        madre: {
        nombre: madre,
        hablar: function() {
                console.log(this.nombre);
        }
}
 };
};
const ana = Persona('Ana', 30, 'Clara');
ana.hablar(); // Ana
ana.madre.hablar(); // Clara
```

This en este caso hace referencia al objeto que contiene el método donde se invoca.

Como vimos en los 3 casos, cuando this es invocado dentro de un método,

implícitamente este hace referencia al objeto que contiene el método, sin importar si el método es añadido luego de haber sido creado el objeto, o si es una función que retorna un objeto.

Asignación explícita

En el caso de asignación implícita, this hace referencia al objeto, que contenía el método donde se invoca this, pero si tenemos una función y deseamos explícitamente asignarle a que va a hacer referencia this, desde ES5 contamos con los métodos call(), apply() y bind(). Vamos a tener una función, que reciba unos parámetros y muestre en consola, con la propiedad nombre a la que haga referencia this, y los parámetros que recibe.

```
const hablar = function(l1, l2, l3) {
        console.log(`Hola mi nombre es ${this.nombre}
        y sé programar en ${l1}, ${l2}, ${l3}.`);
};
const yeison = {
        nombre: 'Yeison',
        edad: 22
};
const lenguajes = ['JavaScript', 'Python', 'C']
```

Ahora, nosotros debemos referenciar la variable yeison para que esta sea this.

call()

```
hablar.call(yeison, lenguajes[0],lenguajes[1],lenguajes[2]);
```

El método call nos permite definir a que va a hacer referencia this, en su primer parámetro, los parámetros siguientes son los parámetros que recibe la función.

apply()

```
hablar.apply(yeison, lenguajes);
```

El método apply, funciona igual que call, permitiendo referencia this en el primer parámetro, pero este nos permite pasar un array, como los parámetros de la funcion.

bind()

Este método funciona diferente a los anteriores, este nos devuelve una función en donde this hace referencia al objeto que pasamos en su parámetro.

```
const hablaYeison = hablar.bind(yeison, lenguajes[0],lenguajes[1],lenguajes[2]);
hablaYeison();
```

Estos tres métodos nos permiten hacer una referencia explícita, y tener claro el valor de this, en la ejecución.

Asignación con new

Otro caso, es cuando invocamos this en un constructor, como el siguiente ejemplo:

```
let Animal = function(color, nombre, tipo) {
        this.color = color;
        this.nombre = nombre;
        this.tipo = tipo;
}
const bipa = new Animal('gris', 'Bipa', 'Felino');
```

En este caso, this hace referencia al objeto que se instanciando.