Trabajando con objetos

JavaScript está diseñado en un paradigma simple basado en objetos. Un objeto es una colección de propiedades, y una propiedad es una asociación entre un nombre (o *clave*) y un valor. El valor de una propiedad puede ser una función, en cuyo caso la propiedad es conocida como un método. Además de los objetos que están predefinidos en el navegador, puedes definir tus propios objetos. Este capítulo describe cómo usar objetos, propiedades, funciones y métodos; y cómo crear tus propios objectos.

Visión general sobre Objetos

Los objetos en JavaScript, como en tantos otros lenguajes de programación, se pueden comparar con objetos de la vida real. El concepto de Objetos en JavaScript se puede entender con objetos tangibles de la vida real.

En JavaScript, un objeto es un entidad independiente con propiedades y tipos. Compáralo con una taza, por ejemplo. Una taza es un objeto con propiedades. Una taza tiene un color, un diseño, un peso, un material del que está hecha, etc. Del mismo modo, los objetos de JavaScript pueden tener propiedades que definan sus características.

Objetos y propiedades

Un objeto de JavaScript tiene propiedades asociadas a él. Una propiedad de un objeto se puede explicar como una variable adjunta al objeto. Las propiedades de un objeto básicamente son lo mismo que las variables comunes de JavaScript, excepto por el nexo con el objeto. Las propiedades de un objeto definen las características del objeto. Accedes a las propiedades de un objeto con una simple notación de puntos:

```
objectName.propertyName
```

Como todas las `variables` de JavaScript, tanto el nombre del objeto (que puede ser una variable normal) como el nombre de la propiedad son sensibles a mayúsculas y minúsculas. Puedes definir propiedades asignándoles un valor. Por ejemplo, vamos a crear un objeto llamado `myCar` y le vamos a asignar propiedades denominadas `make`, `model`, y `year` de la siguiente manera:

```
var myCar = new Object();
myCar.make = 'Ford';
myCar.model = 'Mustang';
myCar.year = 1969;
```

El ejemplo anterior también se podría escribir usando un **iniciador de objeto**, que es una lista delimitada por comas de cero o más pares de nombres de propiedad y valores asociados de un objeto, encerrados entre llaves (`{}`):

```
var myCar = {
    make: 'Ford',
    model: 'Mustang',
    year: 1969
};
```

Las propiedades no asignadas de un objeto son `undefined` (yno `null`).

```
myCar.color; // undefined
```

También puedes acceder o establecer las propiedades de los objetos en JavaScript mediante la notación de corchetes ↑[]↓ (Para más detalle ve Accesores de propiedades). Los objetos, a veces son llamados *arreglos asociativos*, debido a que cada propiedad está asociada con un valor de cadena que se puede utilizar para acceder a ella. Por lo tanto, por ejemplo, puedes acceder a las propiedades del objeto `myCar` de la siguiente manera:

```
myCar['make']  = 'Ford';
myCar['model'] = 'Mustang';
myCar['year']  = 1969;
```

El nombre de la propiedad de un objeto puede ser cualquier cadena válida de JavaScript, o cualquier cosa que se pueda convertir en una cadena, incluyendo una cadena vacía. Sin embargo, cualquier nombre de propiedad que no sea un identificador válido de JavaScript (por ejemplo, el nombre de alguna propiedad que tenga un espacio o un guión, o comience con un número) solo se puede acceder utilizando la notación de corchetes. Esta notación es muy útil también cuando los nombres de propiedades son determinados dinámicamente (cuando el nombre de la propiedad no se determina hasta el tiempo de ejecución). Ejemplos de esto se muestran a continuación:

```
// Se crean y asignan cuatro variables de una sola vez,
// separadas por comas
var myObj = new Object(),
    str = 'myString',
    rand = Math.random(),
    obj = new Object();

myObj.type                  = 'Sintaxis de puntos';
myObj['fecha de creación'] = 'Cadena con espacios';
```

```
myObj[str]                  = 'Valor de cadena';
myObj[rand]                 = 'Número aleatorio';
myObj[obj]                  = 'Object';
myObj['']                   = 'Incluso una cadena vacía';

console.log(myObj);
```

Por favor, ten en cuenta que todas las claves con notación en corchetes se convierten a cadenas a menos que estas sean símbolos, ya que los nombres de las propiedades (claves) en Javascript pueden solo pueden ser cadenas o símbolos (en algún momento, los nombres privados también serán agregados a medida que progrese la propuesta de los campos de clase, pero no las usarás con el formato `[]`). Por ejemplo, en el código anterior, cuando la clave `obj` se añadió a `myObj`, Javascript llamará al método `obj.toString()`, y usará la cadena resultante de esta llamada como la nueva clave.

También puedes acceder a las propiedades mediante el uso de un valor de cadena que se almacena en una variable:

```
var propertyName = 'make';
myCar[propertyName] = 'Ford';

propertyName = 'model';
myCar[propertyName] = 'Mustang';
```

Puedes usar la notación de corchetes con `for...in` para iterar sobre todas las propiedades enumerables de un objeto. Para ilustrar cómo funciona esto, la siguiente función muestra las propiedades del objeto cuando pasas el objeto y el nombre del objeto como argumentos a la función:

```
function showProps(obj, objName) {
  var result = ``;
  for (var i in obj) {
    // obj.hasOwnProperty() se usa para filtrar propiedades de la cadena de prot
    if (obj.hasOwnProperty(i)) {
      result += `${objName}.${i} = ${obj[i]}\n`;
    }
  }
  return result;
}
```

Por lo tanto, la llamada a la función `showProps(myCar, "myCar")` devolverá lo siguiente:

```
myCar.make = Ford
myCar.model = Mustang
myCar.year = 1969
```

Enumerar las propiedades de un objeto

A partir de ECMAScript 5, hay tres formas nativas para enumerar/recorrer las propiedades de objetos:

- `bucles for...in`
  Este método recorre todas las propiedades enumerables de un objeto y su cadena de prototipos
- `Object.keys(o)`
  Este método devuelve un arreglo con todos los nombres de propiedades enumerables ("`claves`") propias (no en la cadena de prototipos) de un objeto `o`.
- `Object.getOwnPropertyNames(o)`
  Este método devuelve un arreglo que contiene todos los nombres (enumerables o no) de las propiedades de un objeto `o`.

Antes de ECMAScript 5, no existía una manera nativa para enumerar todas las propiedades de un objeto. Sin embargo, esto se puede lograr con la siguiente función:

```
function listAllProperties(o) {
       var objectToInspect;
       var result = [];

       for(objectToInspect = o; objectToInspect !== null;
           objectToInspect = Object.getPrototypeOf(objectToInspect)) {
        result = result.concat(
            Object.getOwnPropertyNames(objectToInspect)
        );
    }

        return result;
}
```

Esto puede ser útil para revelar propiedades "ocultas" (propiedades de la cadena de prototipos a las que no se puede acceder a través del objeto, porque otra propiedad tiene el mismo nombre en la cadena de prototipos). Enumerar las propiedades accesibles solo es posible eliminando los duplicados en el arreglo.

Creación de nuevos objetos

JavaScript tiene una colección de objetos predefinidos. Además, puedes crear tus propios objetos. En JavaScript 1.2 y versiones posteriores, puedes crear un objeto usando un iniciador de objeto (en-US). Como alternativa, puedes crear primero una función constructora y luego crear una instancia de un objeto invocando esa función con el operador `new`.

Uso de iniciadores de objeto

Además de la creación de objetos utilizando una función constructora, puedes crear objetos utilizando un iniciador de objeto (en-US). El uso de iniciadores de objetos a veces se denomina crear objetos con notación literal. "Iniciador de objeto" es consistente con la terminología utilizada por C++.

La sintaxis para un objeto usando un iniciador de objeto es:

```
var obj = { property_1:   value_1,   // property_# puede ser un identificador...
            2:            value_2,   // o un número...
            // ...,
            'property n': value_n }; // o una cadena
```

donde `obj` es el nombre del nuevo objeto, cada `property_i` es un identificador (ya sea un nombre, un número o una cadena literal), y cada `value_i` es una expresión cuyo valor se asigna a la `property_i`. El `obj` y la asignación es opcional; si no necesitas hacer referencia a este objeto desde otro lugar, no necesitas asignarlo a una variable. (Ten en cuenta que tal vez necesites envolver el objeto literal entre paréntesis si el objeto aparece donde se espera una declaración, a fin de no confundir el literal con una declaración de bloque).

Los iniciadores de objetos son expresiones, y cada iniciador de objeto da como resultado un nuevo objeto donde la instrucción de creación sea ejecutada. Los iniciadores de objetos idénticos crean objetos distintos que no se compararán entre sí como iguales. Los objetos se crean como si se hiciera una llamada a `new Object()`; es decir, los objetos hechos a partir de expresiones literales de objeto son instancias de `Object`.

La siguiente declaración crea un objeto y lo asigna a la variable `x` si y solo si la expresión `cond` es `true`.

```
if (cond) var x = {greeting: '¡Hola!'};
```

El siguiente ejemplo crea `myHonda` con tres propiedades. Observa que la propiedad `engine` también es un objeto con sus propias propiedades.

```
var myHonda = {color: 'red', wheels: 4, engine: {cylinders: 4, size: 2.2}};
```

También puedes utilizar iniciadores de objetos para crear arreglos. Consulta arreglos literales.

Usar una función constructora

Como alternativa, puedes crear un objeto con estos dos pasos:

1. Definir el tipo de objeto escribiendo una función constructora. Existe una fuerte convención, con buena razón, para utilizar en mayúscula la letra inicial.
2. Crear una instancia del objeto con el operador `new`.

Para definir un tipo de objeto, crea una función para el objeto que especifique su nombre, propiedades y métodos. Por ejemplo, supongamos que deseas crear un tipo de objeto para coches. Quieres llamar `Car` a este tipo de objeto, y deseas que tenga las siguientes propiedades: `make`, `model` y `year`. Para ello, podrías escribir la siguiente función:

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
```

Observa el uso de `this` para asignar valores a las propiedades del objeto en función de los valores pasados a la función.

Ahora puedes crear un objeto llamado `myCar` de la siguiente manera:

```
var mycar = new Car('Eagle', 'Talon TSi', 1993);
```

Esta declaración crea `myCar` y le asigna los valores especificados a sus propiedades. Entonces el valor de `myCar.make` es la cadena "Eagle", para `myCar.year` es el número entero 1993, y así sucesivamente.

Puedes crear cualquier número de objetos `Car` con las llamadas a `new`. Por ejemplo,

```
var kenscar = new Car('Nissan', '300ZX', 1992);
var vpgscar = new Car('Mazda', 'Miata', 1990);
```

<s0>Un objeto puede tener una propiedad que en sí misma es otro objeto. Por ejemplo, supongamos que defines un objeto llamado `person` de la siguiente manera:

```
function Person(name, age, sex) {
  this.name = name;
  this.age = age;
  this.sex = sex;
}
```

y luego instancias dos nuevos objetos `person` de la siguiente manera:

```
var rand = new Person('Rand McKinnon', 33, 'M');
var ken = new Person('Ken Jones', 39, 'M');
```

Entonces, puedes volver a escribir la definición de `Car` para incluir una propiedad `owner` que tomará el objeto `person`, de la siguiente manera:

```
function Car(make, model, year, owner) {
  this.make = make;
  this.model = model;
  this.year = year;
  this.owner = owner;
}
```

Para crear instancias de los nuevos objetos, utiliza lo siguiente:

```
var car1 = new Car('Eagle', 'Talon TSi', 1993, rand);
var car2 = new Car('Nissan', '300ZX', 1992, ken);
```

Nota que en lugar de pasar un valor de cadena o entero cuando se crean los nuevos objetos, las declaraciones anteriores pasan al objetos `rand` y `ken` como argumentos para los `owner`s. Si luego quieres averigüar el nombre del propietario del `car2`, puedes acceder a la propiedad de la siguiente manera:

```
car2.owner.name
```

Ten en cuenta que siempre se puede añadir una propiedad a un objeto previamente definido. Por ejemplo, la declaración

```
car1.color = 'black';
```

agrega la propiedad `color` a `car1`, y le asigna el valor "`black`". Sin embargo, esto no afecta a ningún otro objeto. Para agregar la nueva propiedad a todos los objetos del mismo tipo, tienes que añadir la propiedad a la definición del tipo de objeto `Car`.

Usar el método `Object.create`

Los objetos también se pueden crear por medio del método `Object.create()`. Este método puede ser muy útil, ya que te permite elegir el prototipo del objeto que deseas crear, sin tener que definir una función constructora.

```
// Propiedades y método de encapsulación para Animal
var Animal = {
  type: 'Invertebrates', // Valor predeterminado de las propiedades
  displayType: function() {  // Método que mostrará el tipo de Animal
    console.log(this.type);
  }
};

// Crea un nuevo tipo de animal llamado animal1
var animal1 = Object.create(Animal);
animal1.displayType(); // Muestra: Invertebrates

// Crea un nuevo tipo de animal llamado Fishes
var fish = Object.create(Animal);
fish.type = 'Fishes';
fish.displayType();    // Muestra: Fishes
```

Herencia

Todos los objetos en JavaScript heredan de al menos otro objeto. El objeto del que se hereda se conoce como el prototipo, y las propiedades heredadas se pueden encontrar en el objeto `prototype` del constructor. Para más información consulta Herencia y cadena prototipos (en-US).

Propiedades del objeto indexado

En <s0>JavaScript 1.0</s0>, puedes hacer referencia a una propiedad de un objeto, ya sea por el nombre de la propiedad o por su índice ordinal. Si inicialmente defines una propiedad por su nombre, siempre debes referirte a ella por su nombre, y si inicialmente defines una propiedad por un índice, siempre debes referirte a ella por su índice.

Esta restricción se aplica cuando creas un objeto y sus propiedades con una función constructora (como hicimos anteriormente con el tipo de objeto `Car`) y cuando defines propiedades individuales

explícitamente (por ejemplo, `myCar.color = "red"`). Si inicialmente defines una propiedad de objeto con un índice, como `myCar[5] = "25 mpg"`, posteriormente te refiere a la propiedad solo como `myCar[5]`.

La excepción a esta regla son los objetos HTML, como por ejemplo los objetos contenidos en `formularios`. Siempre puedes hacer referencia a los objetos en estos objetos en forma de arreglo por su número ordinal (según el lugar en el que aparecen en el documento) o por su nombre (si está definido). Por ejemplo, si la segunda etiqueta `<FORM>` en un documento tiene un atributo `NAME` con valor "`myForm`", puedes hacer referencia al formulario como `document.forms[1]` o `document.forms["myForm"]` o `document.forms.myForm`.

Definición de las propiedades de un tipo de objeto

Puedes agregar una propiedad a un tipo de objeto definido previamente mediante el uso de la propiedad `prototype`. Esto define una propiedad que es compartida por todos los objetos del tipo especificado, en lugar de por una sola instancia del objeto. El siguiente código agrega una propiedad `color` a todos los objetos del tipo `Car`, y luego asigna un valor a la propiedad `color` del objeto `car1`.

```
Car.prototype.color = null;
car1.color = 'black';
```

Para más información, consulta la propiedad prototype (en-US) del objeto `Function` en la Referencia de JavaScript.

Definición de métodos

Un *método* es una función asociada a un objeto, o, simplemente, un método es una propiedad de un objeto que es una función. Los métodos se definen normalmente como una función, con excepción de que tienen que ser asignados como la propiedad de un objeto. Consulte también definiciones de métodos para obtener más detalles. Un ejemplo puede ser:

```
objectName.methodname = functionName;

var myObj = {
  myMethod: function(params) {
    // ...hacer algo
  }

  // O ESTO TAMBIÉN FUNCIONA
```

```
  myOtherMethod(params) {
    // ...hacer algo más
  }
};
```

<s0>donde `objectName` es un objeto existente, `methodname` es el nombre que se le va a asignar al método, y `functionName` es el nombre de la función.

Entonces puedes llamar al método en el contexto del objeto de la siguiente manera:

```
object.methodname(params);
```

Puedes definir métodos para un tipo de objeto incluyendo una definición del método en la función constructora del objeto. Podrías definir una función que formateé y muestre las propiedades de los objetos del tipo `Car` previamente definidas; por ejemplo:

```
function displayCar() {
  var result = `Un hermoso ${this.year} ${this.make} ${this.model}`;
  pretty_print(result);
}
```

donde `pretty_print` es una función para mostrar una línea horizontal y una cadena. Observa el uso de `this` para referirse al objeto al que pertenece el método.

Puedes hacer de esta función un método de `Car` agregando la declaración

```
this.displayCar = displayCar;
```

a la definición del objeto. Por lo tanto, la definición completa de `Car` ahora se verá así:

```
function Car(make, model, year, owner) {
  this.make = make;
  this.model = model;
  this.year = year;
  this.owner = owner;
  this.displayCar = displayCar;
}
```

Entonces puedes llamar al método `displayCar` para cada uno de los objetos de la siguiente manera:

```
car1.displayCar();
car2.displayCar();
```

Usar `this` para referencias a objetos

JavaScript tiene una palabra clave especial, `this`, que puedes usar dentro de un método para referirte al objeto actual. Por ejemplo, supongamos que tienes 2 objetos, `Manager` e `Intern`. Cada objeto tiene su propio `name`, `age` y `job`. En la función `sayHi()`, observa que hay `this.name`. Cuando se agregan a los 2 objetos, se pueden llamar y devuelve el `'Hola, mi nombre es'` y luego agrega el valor `name` de ese objeto específico. Como se muestra abajo.

```
const Manager = {
  name: "John",
  age: 27,
  job: "Software Engineer"
}

const Intern= {
  name: "Ben",
  age: 21,
  job: "Software Engineer Intern"
}

function sayHi() {
    console.log('Hola, mi nombre es ', this.name)
}

// agrega la función sayHi a ambos objetos
Manager.sayHi = sayHi;
Intern.sayHi = sayHi;

Manager.sayHi() // Hola, mi nombre es John'
Intern.sayHi() // Hola, mi nombre es Ben'
```

`this` se refiere al objeto en el que se encuentra. Puedes crear una nueva función llamada `howOldAmI()` que registra una oración que dice cuántos años tiene la persona.

```
function howOldAmI() {
  console.log('Tengo ' + this.age + ' años.')
}
Manager.howOldAmI = howOldAmI;
Manager.howOldAmI() // Tengo 27 años.
```

Definición de captadores (`getters`) y establecedores (`setters`)

Un captador (getter) es un método que obtiene el valor de una propiedad específica. Un establecedor (setter) es un método que establece el valor de una propiedad específica. Puedes definir captadores y establecedores en cualquier objeto principal predefinido o en un objeto definido por el usuario que admita la adición de nuevas propiedades.

En principio, los captadores y establecedores pueden ser

- definido usando iniciadores de objeto, o
- agregado posteriormente a cualquier objeto en cualquier momento usando un método de adición para el captador o el establecedor.

Al definir captadores y establecedores usando iniciadores de objeto, todo lo que necesitas hacer es prefijar un método captador con `get` y un método establecedor con `set`. Por supuesto, el método captador no debe esperar un parámetro, mientras que el método establecedor espera exactamente un parámetro (el nuevo valor a establecer). Por ejemplo:

```
var o = {
  a: 7,
  get b() {
    return this.a + 1;
  },
  set c(x) {
    this.a = x / 2;
  }
};

console.log (o.a); // 7
console.log (o.b); // 8 <-- En este punto se inicia el método get b().
o.c = 50;          // <-- En este punto se inicia el método set c(x)
console.log(o.a); // 25
```

var o = {

- a: 7,
- `o.b` — un captador que devuelve `o.a` más 1
- `o.c` — un establecedor que establece el valor de `o.a` en la mitad del valor que se establece en `o.c`

Ten en cuenta que los nombres de función de los captadores y establecedores definidos en un objeto literal usando "[gs]et *propiedad*()" (en contraposición a `__define [GS]etter__`) no son los nombres de los captadores en sí, aunque la sintaxis `[gs]et *propertyName*() {}` te puede inducir a pensar lo contrario.

Los captadores y establecedores también se pueden agregar a un objeto en cualquier momento después de la creación usando el método `Object.defineProperties`. El primer parámetro de este método es el objeto sobre el que se quiere definir el captador o establecedor. El segundo parámetro es un objeto cuyo nombre de propiedad son los nombres `getter` o `setter`, y cuyos valores de propiedad son objetos para la definición de las funciones `getter` o `setter`. Aquí hay un ejemplo que define el mismo `getter` y `setter` utilizado en el ejemplo anterior:

```
var o = { a: 0 };

Object.defineProperties(o, {
    'b': { get: function() { return this.a + 1; } },
    'c': { set: function(x) { this.a = x / 2; } }
});

o.c = 10; // Ejecuta el establecedor, que asigna 10/2 (5) a la propiedad 'a'
console.log(o.b); // Ejecuta el captador, que produce un + 1 o 6
```

¿Cuál de las dos formas elegir? depende de tu estilo de programación y de la tarea que te ocupa. Si ya utilizas el iniciador de objeto al definir un prototipo probablemente escojas la primer forma la mayoría de las veces. Esta forma es más compacta y natural. Sin embargo, si más tarde necesitas agregar captadores y establecedores — porque no lo escribiste en el objeto prototipo o particular — entonces la segunda forma es la única forma posible. La segunda forma, probablemente representa mejor la naturaleza dinámica de JavaScript — pero puede hacer que el código sea difícil de leer y entender.</s6>

Eliminar propiedades

Puedes eliminar una propiedad no heredada mediante el operador `delete`. El siguiente código muestra cómo eliminar una propiedad.

```
//Crea un nuevo objeto, myobj, con dos propiedades, a y b.
var myobj = new Object;
myobj.a = 5;
myobj.b = 12;

// Elimina la propiedad a, dejando a myobj solo con la propiedad b.
```

```
delete myobj.a;
console.log ('a' in myobj); // muestra: "false"
```

También puedes utilizar `delete` para eliminar una variable global siempre y cuando no se haya utilizado la palabra clave `var` para declarar la variable:

```
g = 17;
delete g;
```

Comparar objetos

Como sabemos los objetos son de tipo referencia en JavaScript. Dos distintos objetos nunca son iguales, incluso aunque tengan las mismas propiedades. Solo comparar la misma referencia de objeto consigo misma arroja verdadero.

```
// Dos variables, dos distintos objetos con las mismas propiedades
var fruit = { name: 'apple' };
var fruitbear = { name: 'apple' };

fruit == fruitbear; // devuelve false
fruit === fruitbear; // devuelve false
```

```
// Dos variables, un solo objeto
var fruit = { name: 'apple' };
var fruitbear = fruit; // Asigna la referencia del objeto fruit a fruitbear

// Aquí fruit y fruitbear apuntan al mismo objeto
fruit == fruitbear; // devuelve true
fruit === fruitbear; // devuelve true

fruit.name = 'grape';
console.log(fruitbear); // Produce: { name: "grape" }, en lugar de { name: "appl
```

Para obtener más información sobre los operadores de comparación, consulta Operadores de comparación (en-US).

- Para profundizar más, lee sobre los detalles del modelo de objetos de JavaScript.
- Para obtener más información sobre las clases de ECMAScript 2015 (una forma alternativa de crear objetos), lee el capítulo Clases de JavaScript.