Javascript properties are enumerable, writable and configurable

November 3, 2014 9:30 am 17 comments

Objects are one of the main parts of Javascript. JS syntax for Objects is really concise and easy to use, so we are constantly creating objects and using them as hashmaps effortlessly.

```
// My beloved object ob
var ob = {a: 1};

// Accessing to a property
ob.a; // => 1

// Modifying the value of a property
ob.a = 0;
ob.a; // => 0;

// Creating a new property
ob.b = 2;
ob.b; // => 2

// Deleting a property
delete ob.b;
ob.b; // => undefined
```

But, do you know that all the object properties in the example above are enumerable, writable and configurable? I mean:

- **Enumerable**: I can access to all of them using a `for..in` loop. Also, enumerable property keys of an object are returned using `Object.keys` method.
- **Writable**: I can modify their values, I can update a property just assigning a new value to it: `ob.a = 1000;`
- **Configurable**: I can modify the behavior of the property, so I can make them non-enumerable, non-writable or even non-cofigurable if I feel like doing so. Configurable properties are the only ones that can be removed using the `delete` operator.

I bet that you knew about the two first features of `Object`'s properties, but there are less developers that know that they can **create and update them to be non-enumerable or immutable** using the `Object`'s method called defineProperty.

```
// Adding a property to ob using Object.defineProperty
Object.defineProperty( ob, 'c', {
  value: 3,
  enumerable: false,
  writable: false,
  configurable: false
});

ob.c; // => 3

Object.getOwnPropertyDescriptor( ob, 'c' );
// => {value: 3, enumerable: false, writable: false, configurable: false}
```

I reckon that the syntax is not as friendly as usual one, but having this kind of properties can be really handy for some purposes. The object that define the property is called **descriptor**, and you can have a look at the descriptor of any property using `Object.getOwnPropertyDescriptor` method.

It is funny that **the default option values** for `Object.defineProperty` **are completely the opposite** to the ones applied when adding a property by assigment: The default property by assignment is non-enumerable,

non-writable and non-configurable.

```
// The 'f' property will be non-enumerable. non-writable and non-configurable
Object.defineProperty( ob, 'f', {value: 6} );
```

It is also possible to define the properties on object creation if you instantiate it using the method `Object.create( prototype, properties )`. It accepts an object with property descriptors as the second parameter, and it can be used as follows

```
var ob = Object.create(Object.prototype, {
  a: { writable:true, enumerable:true, value: 1 },
  b: { enumerable: true, value: 2 }
});

ob; // => {a:1, b:2}
```

Object's non-enumerable properties

As I said before, enumerable properties are accessible using `for...in` loops, so, non-enumerable ones aren't. Basically, non-enumerable properties won't be available using most of the functions that handle `Objects` as hashmaps.

- They won't be in `for..in` iterations.
- They won't appear using `Object.keys` function.
- They are not serialized when using `JSON.stringify`

So they are kind of *secret* variables, but you can always access to them directly.

```
var ob = {a:1, b:2};

ob.c = 3;
Object.defineProperty(ob, 'd', {
  value: 4,
  enumerable: false
});

ob.d; // => 4

for( var key in ob ) console.log( ob[key] );
// Console will print out
// 1
// 2
// 3

Object.keys( ob );  // => ["a", "b", "c"]

JSON.stringify( ob ); // => "{a:1,b:2,c:3}"

ob.d; // => 4
```

Since this kind of properties are not serialized, I found them really useful when handling data model objects. I can add handy information to them using non enumerable properties.

```
// Imagine the model that represent a car, it has a reference
// to its owner using owner's id in the owner attribute

var car = {
  id: 123,
  color: red,
  owner: 12
};

// I also have fetched the owner from the DB
// Of course, the car is mine :)
var owner = {
```

```
 id: 12,
 name: Javi
}

// I can add the owner data to the car model
// with a non-enumerable property, maybe it can
// be useful in the future
Object.defineProperty( car, 'ownerOb', {value: owner} );

// I need the owner data now
car.ownerOb; // => {id:12, name:Javi}

// But if I serialize the car object, I can't see me
JSON.stringify( car ); // => '{id: 123, color: "red", owner: 12}'
```

Can you think how useful can this be to create a ORM library for example?

In case that you need to know all properties in an object, enumerable and non-enumerable ones, the method `Object.getOwnPropertyNames` returns an array with all the names.

Object's non-writable properties

While the world waits for ES6 to finally arrive with the desired `const` statement, non-writable properties are the **most similar thing to a constant** that we have in Javascript. Once its value is defined, **it is not possible to change it using assignments**.

```
var ob = {a: 1};

Object.defineProperty( ob, 'B', {value: 2, writable:false} );

ob.B; // => 2

ob.B = 10;

ob.B; // => 2
```

As you can see, the assignment didn't affect the value of `ob.B` property. You need to be careful, because **the assignment always returns the value assigned**, even if the property is non-writable like the one in the example. In strict mode, trying to modifying a non-writable property would throw an `TypeError` exception:

```
var ob = {a: 1};
Object.defineProperty( ob, 'B', {value: 2, writable:false} );

// Assingments returns the value
ob.B = 6; // => 6
ob.B = 1000; // => 1000

// But the property remains the same
ob.B; => 2;

function updateB(){
  'use strict';
  ob.B = 4; // This would throw an exception
}

updateB(); // Throws the exception. I told you.
```

It is also needed to keep in mind that **if the non-writable property contains an object**, the reference to the object is what is not writable, but **the object itself can be modified yet**:

```
var ob = {a: 1};
Object.defineProperty( ob, 'OB', {value: {c:3}, writable:false} );

ob.OB.c = 4;
ob.OB.d = 5;
```

```
6
7  ob.OB; // => {c:4, d:5}
8
9  ob.OB = 'hola';
10
11 ob.OB; // => {c:4, d:5}
```

If you want to have a property with an completely non-writable object, you can use the function `Object.freeze`. `freeze` will make impossible to add, delete or update any object's property, and you will get a `TypeError` if you try so in strict mode.

```
1  var ob = { a: 1, b: 2 };
2
3  ob.c = 3;
4
5  // Freeze!
6  Object.freeze( ob ); // => {a:1,b:2,c:3}
7
8  ob.d = 4;
9  ob.a = -10;
10 delete ob.b;
11
12 Object.defineProperty( 'ob', 'e', {value: 5} );
13
14 // Every modification was ignored
15 ob; // => {a:1,b:2,c:3}
```

Object's non-configurable properties

You can update the previous behaviors of the properties if they are defined as configurable. You can use `defineProperty` once and again to change the property to writable or to non-enumerable. But once you have defined the property as non-configurable, there is only one behaviour you can change: If the property is writable, you can convert it to non-writable. Any other try of definition update will fail throwing a `TypeError`.

```
1  var ob = {};
2  Object.defineProperty( ob, 'a', {configurable:false, writable:true} );
3
4  Object.defineProperty(ob, 'a', { enumerable: true }); // throws a TypeError
5  Object.defineProperty(ob, 'a', { value: 12 }); // throws a TypeError
6  Object.defineProperty(ob, 'a', { writable: false }); // This is allowed!!
7  Object.defineProperty(ob, 'a', { writable: true }); // throws a TypeError
```

An important thing to know about the non-configurable properties is that they can't be removed from the object using the operator `delete`. So if you create a property non-configurable and non-writable you have a *frozen* property.

```
1  var ob = {};
2
3  Object.defineProperty( ob, 'a', {configurable: true, value: 1} );
4
5  ob; // => {a:1}
6  delete ob.a; // => true
7  ob; // => {}
8
9  Object.defineProperty( ob, 'a', {configurable: false, value: 1} );
10
11 ob; // => {a:1}
12 delete ob.a; // => false
13 ob; // => {a:1}
```

Conclusion

`Object.defineProperty` was introduced with ES5, and you can start using it right now, **it is supported by all modern browsers**, including IE 9 (and even IE 8, but only for DOM objects). It is always fun to play with javascript basics in a different way that we are used to, and it is easy to learn new stuff just observing how javascript core objects work.

`Object.defineProperty` also give us the chance of creating customized getters and setters for the properties, but I won't write about that today. If you want to learn more, have a look at the always amazing Mozilla's documentation.

Computer are useless. They can only give you answers.

— Pablo Picasso