

Objetos

Tipos de datos

Javascript es un lenguaje débilmente tipado.

No hace falta declarar el tipo de las variables en su declaración.

El tipo de estas variables es determinado en tiempo de ejecución.

En Javascript tenemos siete tipos primitivos de datos.

Tipos de datos

Boolean: dos valores. `true` y `false`.

Number: valor de doble precisión de 64 bits, `-Infinity`, `Infinity` y `NaN`

String: cadenas de texto. Entre `“”` o `‘’`.

Tipos de datos

Null: `null`.

Undefined: `undefined`.

Symbol: único e inmutable. Usado para propiedades anónimas de objetos o privadas de clase. `Symbol()`

Tipos de datos

```
typeof undefined      // undefined
```

```
typeof null           // object
```

```
null === undefined    // false
```

```
null == undefined     // true
```

```
false == 0            // true
```

```
false === 0           // false
```

```
true === !0           // true
```

```
false === !!0         // true
```

Tipos de datos

Symbol: único e immutable. Usado para propiedades anónimas de objetos o privadas de clase

```
var miSimbolo = Symbol("Julian"); // Symbol(Julian)
Symbol.keyFor(Symbol.for("Julian")); // "Julian"
var obj = {};
obj[miSimbolo] = "Julian";
JSON.stringify(obj) // {}
Object.keys(obj) // []
Object.getOwnPropertyNames(obj) // []
Object.getOwnPropertySymbols(obj) // [ Symbol() ]
```

Objetos

Conjunto de propiedades propias + heredadas de otro objeto (prototipos)

¡Qué no cunda el pánico!

Objetos

Dinámicos

```
var obj = {};  
obj.nuevaPropiedad = 1;  
obj['nuevaPropiedad'] = 1;  
  
delete obj.nuevaPropiedad;
```


Objetos

Sets de strings

```
var strset = {  
    hola: true,  
    adios: true  
};  
"hola" in strset;
```

Objetos

Referencias

```
var p1 = {x: 1},  
    p2 = p1;
```

```
p1 === p2; // true  
p1.x = 5;  
p2.x; // 5
```

Objetos

Todo son objetos excepto: strings, números, booleans o undefined

Strings, números, null y booleans se comportan como objetos inmutables

Objetos

Pueden ser argumento y resultado de ejecución de funciones. Contener valores primitivos u otros objetos, incluyendo funciones.

```
(function (obj) {  
    return {b: 2};  
}) ({a: 1});
```

```
var obj = {  
    f: function() {  
        console.log("hola");  
    }  
};  
obj.f();
```

Objetos

Literales:

```
{  
  un: "objeto",  
  literal: true  
};
```

Construidos:

```
new NombreDeUnaFuncion();
```

Objetos: Mensajes

Teniendo:

```
var obj = {  
  nombre: "Pepito",  
  saludo: function () {  
    return "Hola, Mundo!";  
  };  
};
```

¿Qué significa esto?

```
obj.nombre;
```

Objetos: Mensajes

Teniendo:

```
var obj = {  
  nombre: "Pepito",  
  saludo: function () {  
    return "Hola, Mundo!";  
  };  
};
```

¿Y esto?

```
obj.saludo;
```

Objetos: Mensajes

Teniendo:

```
var obj = {  
  nombre: "Pepito",  
  saludo: function () {  
    return "Hola, Mundo!";  
  };  
};
```

¿Y esto otro?

```
obj["saludo"]();
```


Objetos: Mensajes

Teniendo:

```
var obj = {  
  nombre: "Pepito",  
  saludo: function () {  
    return "Hola, Mundo!";  
  };  
};
```

¿Es lo mismo?

```
var fn = obj["saludo"];  
fn();
```

NO

No, no es lo mismo

Objetos: Mensajes

Una función se puede ejecutar de 4 maneras:

- Invocando directamente la función

- Enviando un mensaje a un objeto (método)

```
(function() { alert("Hey!"); })();
```

```
objeto.metodo();
```

```
new MiConstructor();
```

```
fn.call({}, "param");
```

Objetos: Mensajes

Una función se puede ejecutar de 4 maneras:

- Invocando directamente la función

- Enviando un mensaje a un objeto (método)

```
(function() { alert("Hey!"); })();
```

```
objeto.metodo();
```

Objetos: Mensajes

Un mensaje se envía a un receptor:

```
var obj = {};  
obj.toString(); // [object Object]  
  
"hola don Pepito".toUpperCase();
```

Objetos: Mensajes

```
var fn = obj.metodo;  
fn()
```

Accede a la propiedad “metodo” de obj

Supongo que es una función y la invoco

NO hay receptor

```
obj.metodo();
```

Envía un mensaje “metodo” a obj

Si existe, obj se encarga de ejecutar la función

obj es el receptor

Objetos: Mensajes

Un error típico:

```
$("#elemento").click(objeto.clickHandler);
```

Lo que se intenta decir: “Al hacer click sobre #elemento, envía el mensaje clickHandler a objeto”

Lo que se dice en realidad: “Accede al valor de la propiedad clickHandler de objeto y ejecútalo al hacer click sobre #elemento”

Objetos: Mensajes

¿Por qué tanto lío con el receptor del mensaje?

¡Porque el receptor es **this**!

La metáfora mensaje/receptor aclara su (escurridizo) significado.

this = “receptor de este mensaje”

Objetos: Mensajes

this

- Su significado es dinámico
- Se decide en el momento (y según la manera) de ejecutar la función
- Se suele llamar “el contexto de la función”
- Cuando no hay receptor, apunta al objeto global

Objetos: Mensajes

Cuando no hay receptor, es el objeto global

```
var nombre = "Sonia"
var obj = {
  nombre: "Pepito",
  saludo: function() {
    alert("hola " + this.nombre)
  }
}
var fn = obj["saludo"];
fn();
```

Objetos: Mensajes

Cuando no hay receptor, es el objeto global

```
var obj = {  
  nombre: "Pepito",  
  saludo: function() {  
    alert("hola " + this.nombre);  
  }  
};  
  
var maria = {  
  nombre: "María"  
};
```

```
maria.saludo = obj.saludo;  
maria.saludo();
```

Objetos: Mensajes

Semánticamente, es un parámetro oculto

```
function ([this]) {  
    alert("hola" + this.nombre);  
}
```

Que el receptor se encarga de proveer

```
obj.saludo(); => saludo([obj]);
```

Objetos: Mensajes

Otro error común:

```
var obj = {  
  clicks: 0,  
  init: function() {  
    $("#element").click(function() {  
      this.clicks += 1;  
    });  
  }  
};  
obj.init();
```

Objetos: Mensajes

Otro error común:

```
var obj = {  
  clicks: 0,  
  init: function() {  
    var that = this;  
    $("#element").click(function() {  
      that.clicks += 1;  
    });  
  }  
};  
obj.init();
```

Objetos: Mensajes

Una función se puede ejecutar de 4 maneras:

- Invocando directamente la función

- Enviando un mensaje a un objeto (método)

```
(function() { alert("Hey!"); })();
```

```
objeto.metodo();
```

```
new MiConstructor();
```

```
fn.call({}, "param");
```

Objetos: Mensajes

```
fn.call({}, "param");
```


El receptor: this

- Las funciones son objetos
- Se pueden manipular como cualquier otro objeto
 - Asignar valores a propiedades
 - Pasar como parámetros a otras funciones
 - Ser el valor de retorno
 - Guardarse en variables u otros objetos

Tienen métodos

```
var fn = function() { alert("Hey!"); };  
fn.toString();
```

El receptor: this

Dos métodos permiten manipular el receptor (contexto):

```
fn.call(context [, arg1 [, arg2 [...]]])
```

```
var a = [1, 2, 3];  
Array.prototype.slice.call(a, 1, 2); // [2]
```

```
fn.apply(context, arglist)
```

```
var a = [1, 2, 3];  
Array.prototype.slice.apply(a, [1, 2]); // [2]
```

El receptor: this

```
var nombre = "Objeto Global";
```

```
function saluda() {  
  alert("Hola! Soy " + this.nombre);  
}
```

```
var alicia = {  
  nombre: "Alicia"  
};
```

```
saluda();  
saluda.call(alicia);
```

arguments

El otro parámetro oculto

Contiene una lista de todos los argumentos

NO es un Array

```
function echoArgs() {  
    alert(arguments); // [object Arguments]  
}  
echoArgs(1, 2, 3, 4);
```

arguments

Se comporta más o menos como un array, pero no del todo

```
function echoArgs() {  
    return arguments[0] // 1  
}
```

```
echoArgs(1, 2, 3, 4);
```

arguments

Se comporta más o menos como un array, pero no del todo

```
function echoArgs () {  
    return arguments.slice(0, 1) // Error!  
}
```

```
echoArgs(1, 2, 3, 4);
```

Ejercicio mental

¿Qué hace esta función?

```
function misterio(ctx, fn) {  
  return function() {  
    return fn.apply(ctx, arguments);  
  }  
}
```

Ejercicio mental

¿Qué hace esta función?

```
function misterio(ctx, fn) {  
    return function() {  
        return fn.apply(ctx, arguments);  
    }  
}  
  
var algo = misterio();  
typeof algo; //???  
typeof algo(); //???
```


Ejercicio mental

¿Qué hace esta función?

```
function misterio(ctx, fn) {  
    return function() {  
        return fn.apply(ctx, arguments);  
    }  
}  
  
var algo = misterio({}, function() {  
    return this;  
});  
  
typeof algo(); //???
```

Ejercicio mental

¿Qué hace esta función?

```
function misterio(ctx, fn) {  
    return function() {  
        return fn.apply(ctx, arguments);  
    }  
}  
  
var obj = {}  
var algo = misterio(obj, function() {  
    return this;  
});  
  
obj === algo(); //???
```

Ejercicio mental

¿Qué hace esta función?

```
function misterio(ctx, fn) {  
    return function() {  
        return fn.apply(ctx, arguments);  
    }  
}  
  
var obj = {}  
var algo = misterio({}, function() {  
    return this;  
});  
  
obj === algo(); //???
```

Ejercicio mental

¿Qué hace esta función?

```
function misterio(ctx, fn) {  
  return function() {  
    return fn.apply(ctx, arguments);  
  }  
}  
  
var obj = {name: "Barbara"}  
var algo = misterio(obj, function() {  
  return this.name;  
});  
algo(); //???
```

Ejercicio mental

¿Qué hace esta función?

```
function misterio(ctx, fn) {  
  return function() {  
    return fn.apply(ctx, arguments);  
  }  
}  
  
var obj = {name: "Barbara"}  
var algo = misterio(obj, function(saludo) {  
  return saludo + ", " + this.name;  
});  
  
algo("Hola"); //???
```

Ejercicio mental

¿Qué hace esta función?

```
function misterio(ctx, fn) {  
  return function() {  
    return fn.apply(ctx, arguments);  
  }  
}  
  
var barbara = {name: "Barbara"}, carlos = {name: "Carlos"}  
var algo = misterio(barbara, function(saludo) {  
  return saludo + ", " + this.name;  
});  
algo.call(carlos, "Hola"); //???
```

Ejercicio mental

bind: una función que fija el contexto

```
function bind(ctx, fn) {  
    return function() {  
        return fn.apply(ctx, arguments);  
    }  
}
```

Volviendo al problema

```
var obj = {  
  clicks: 0,  
  init: function() {  
    //MAL  
    $("#element").click(function() {  
      this.clicks += 1;  
    });  
  }  
};  
obj.init();
```


Volviendo al problema

Apaño:

```
var obj = {  
  clicks: 0,  
  init: function() {  
    var that = this;  
    $("#element").click(function() {  
      that.clicks += 1;  
    });  
  }  
};  
obj.init();
```

Volviendo al problema

bind al rescate:

```
var obj = {  
  clicks: 0,  
  incClicks: function() {  
    this.clicks += 1;  
  }  
  init: function() {  
    $("#element").click(this.incClicks.bind(this));  
  }  
};
```

Ejercicio mental

¿Qué hace esta otra función?

```
function enigma(fn) {  
  //   var slice = Array.prototype.slice,  
    args = slice.call(arguments, 1);  
  return function() {  
    var newargs = slice.call(arguments);  
    return fn.apply(this, args.concat(newargs));  
  };  
}
```

Ejercicio mental

¿Qué hace esta otra función?

```
function enigma(fn) {  
  var slice = Array.prototype.slice,  
    args = slice.call(arguments, 1);  
  return function() {  
    var newargs = slice.call(arguments);  
    return fn.apply(this, args.concat(newargs));  
  };  
}
```

```
var cosa = enigma();  
typeof cosa; //???  
typeof cosa(); // ???
```

Ejercicio mental

¿Qué hace esta otra función?

```
function enigma(fn) {  
  var slice = Array.prototype.slice,  
    args = slice.call(arguments, 1);  
  return function() {  
    var newargs = slice.call(arguments);  
    return fn.apply(this, args.concat(newargs));  
  };  
}  
  
var cosa = enigma(function() {  
  return "Hola!";  
});  
  
cosa(); // ???
```

Ejercicio mental

¿Qué hace esta otra función?

```
function enigma(fn) {  
  var slice = Array.prototype.slice,  
    args = slice.call(arguments, 1);  
  return function() {  
    var newargs = slice.call(arguments);  
    return fn.apply(this, args.concat(newargs));  
  };  
}  
  
function saluda(nombre) {  
  return "Hola, " + nombre + "!";  
}  
  
var cosa = enigma(saluda);  
cosa("Mundo"); // ???
```

Ejercicio mental

¿Qué hace esta otra función?

```
function enigma(fn) {  
  var slice = Array.prototype.slice,  
      args = slice.call(arguments, 1);  
  return function() {  
    var newargs = slice.call(arguments);  
    return fn.apply(this, args.concat(newargs));  
  };  
}  
  
function saluda(nombre) {  
  return "Hola, " + nombre + "!";  
}  
  
var cosa = enigma(saluda, "Mundo");  
cosa(); // ???
```

Ejercicio mental

¿Qué hace esta otra función?

```
function enigma(fn) {  
  var slice = Array.prototype.slice,  
      args = slice.call(arguments, 1);  
  return function() {  
    var newargs = slice.call(arguments);  
    return fn.apply(this, args.concat(newargs));  
  };  
}  
  
function saluda(saludo, nombre) {  
  return saludo + ", " + nombre + "!";  
}  
  
var cosa = enigma(saluda, "Hola", "Mundo");  
cosa(); // ???
```


Ejercicio mental

¿Qué hace esta otra función?

```
function enigma(fn) {  
  var slice = Array.prototype.slice,  
      args = slice.call(arguments, 1);  
  return function() {  
    var newargs = slice.call(arguments);  
    return fn.apply(this, args.concat(newargs));  
  };  
}  
  
var dario = {nombre: "Dario"}  
function saluda(saludo) {  
  return saludo + ", " + this.nombre + "!";  
}  
  
var cosa = enigma(saluda, "Qué pasa");  
cosa.call(dario); // ???
```

Ejercicio mental

Curry: aplicación parcial de funciones

```
function curry(fn) {  
  var slice = Array.prototype.slice,  
      args = slice.call(arguments, 1);  
  return function() {  
    var newargs = slice.call(arguments);  
    return fn.apply(this, args.concat(newargs));  
  };  
}
```

Rizar el rizo

```
function getter(prop) { return this[prop]; }  
function setter(prop, value) { this[prop] = value; }
```

```
var manuel = {  
  nombre: "Manuel",  
  edad: 32  
};
```

```
var edadDeManuel = bind(manuel, curry(getter, "edad"));  
edadDeManuel(); // ???
```

Objetos estandar 'built-in'

Son objetos que el lenguaje nos ofrece por defecto, incluyendo métodos y propiedades.

Cada uno tiene sus operaciones y sus peculiaridades

Por valor: Infinity, NaN, undefined...

Propiedades de función: parseFloat, escape...

Objetos fundamentales: Function, Object, Symbol

Objetos de control de abstracción: Promise, Generator...

Y muchos mas: Number, Math, String, Array, RegExp...

Maps

Los mapas son diccionarios clave valor.

```
const map = new Map()  
map.set("course", "ES6")  
console.log(map.get("course")) // ES6
```

```
const otherMap = new Map([["course", "ES6"], ["author",  
"Miguel"]])  
console.log(otherMap.size) // 2  
otherMap.forEach((key, value, map) => {  
  console.log(key, value)  
})
```

Sets

Los sets son colecciones de elementos únicos de cualquier tipo.

```
const set = new Set([5, "5", {}, {}, 5])
```

```
console.log(set.size) // 4  
console.log(set.has(5)) // true  
console.log(set.has({})) // false
```

Sets

Una forma sencilla de eliminar duplicados

```
function removeDuplicates(items) {  
    return [...new Set(items)]  
}
```

```
const duplicates = [1,2,2,3,3,3,4,5,5]  
const noDuplicates = removeDuplicates(duplicates)
```

```
console.log(noDuplicates) // [1,2,3,4,5]
```

WeakSets

En los WeakMaps y WeakSets las referencias no son enumerables y permiten al GC borrar referencias no usadas.

```
let set = new WeakSet()  
let key = {a: 1}  
  
set.add(key)  
console.log(set.has(key)) // true  
key = undefined  
console.log(set.has(key)) // false
```


WeakSets

En los WeakMaps y WeakSets las referencias no son enumerables y permiten al GC borrar referencias no usadas.

```
const node = document.querySelector('body')  
  
const map = new WeakMap([[node, {p1: 1, p2: 2}]])  
  
console.log(map.get(node)) // {p1: 1, p2: 2}  
  
node.parentNode.removeChild(node)  
  
console.log(map.get(node)) // undefined
```

Getters y setters

Las usamos para computar propiedades de objetos.

Un *getter* es un método que extrae el valor de una propiedad.

Un *setter* es un método que establece el valor de una propiedad.

Pueden ser eliminados usando el operador *delete*.

La sintaxis de su creación puede ser utilizada en objetos literales, con *Object.create* o *Object.defineProperty*.

Getters y setters

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

Getters y setters

```
var objeto = {  
  a: 7,  
  get b() {  
    return this.a + 1;  
  },  
  set c(x) {  
    this.a = x / 2;  
  }  
};  
objeto.a; //???  
objeto.b; //???  
objeto.c = 50;  
objeto.a; //???
```

Arrow functions

Es una forma especial y más corta de expresar funciones anónimas con algunas diferencias notables.

```
function (x) { return x+1; }
```

Equivale a:

```
x => x+1;
```

O

```
(x) => x+1;
```

Arrow functions

No vincula *this* ni *arguments*. No pueden ser usadas como constructores.

Estos valores los obtiene del entorno en el que está englobada.

```
function (x, y) { return x+y; }
```

Equivale a:

```
(x, y) => x+y;
```

Arrow functions

¿Qué pasa si quiero bindear el `this` de un arrow function con `call` o `apply`?

```
var fn = () => {console.log(this);}  
var triangulo = {  
  a: {x: 1, y: 1},  
  b: {x: 2, y: 1},  
  c: {x: 3, y: 1}  
}  
fn.call(triangulo)
```

Arrow functions

¿Qué pasa si quiero utilizar una arrow function como método?

```
var triangulo = {  
  a: {x: 1, y: 2},  
  b: {x: 2, y: 1},  
  c: {x: 3, y: 3},  
  area: function() {  
    return Math.abs(  
      (this.b.x - this.a.x) * (this.b.y - this.a.y) -  
      (this.c.x - this.a.x) * (this.c.y - this.a.y)) / 2  
    )  
  }  
}  
triangulo.area() //??
```


Arrow functions

¿Qué pasa si quiero utilizar una arrow function como método?

```
var triangulo = {  
  a: {x: 1, y: 2},  
  b: {x: 2, y: 1},  
  c: {x: 3, y: 3},  
  area: () => Math.abs(  
    (this.b.x - this.a.x) * (this.b.y - this.a.y) -  
    (this.c.x - this.a.x) * (this.c.y - this.a.y)) / 2  
}  
triangulo.area() //??
```

Arrow functions

¿Qué pasa con arguments?

```
var arguments = 42;  
var arr = () => arguments;
```

```
arr();
```

```
function foo() {  
  var f = () => arguments[0];  
  return f(2);  
}
```

```
foo(1);
```

Arrow functions

¿Qué pasa aquí?

```
const sumador = {  
  resultado: 0,  
  sumar: (numeros) => {  
    numeros.forEach(n => {  
      this.resultado += n;  
    });  
  }  
  
};  
sumador.sumar([1, 2, 3]);
```