JavaScript

Sesión 2 - **JavaScript Orientado a Objetos**

Índice

2.1 Trabajando con Objetos

- Todo en *JavaScript* es un objeto
- Una cadena es un objeto
  - propiedad `length`, ...
  - método `toUpperCase()`, ...
- Los tipos primitivos son inmutables
- Los objetos en JavaScript son colecciones de claves mutables.
  - arrays (`Array`), funciones (`Function`), fechas (`Date`), expresiones regulares (`RegExp`), y los objetos (`Object`)
- Para crear un objeto → `new Object()`

```
var obj = new Object();
var str = new String();
```

Propiedades

- Un objeto es un contenedor de propiedades
- Cada propiedad tiene un nombre y un valor
- Un objeto puede contener otros objetos → grafo, árbol
- Operador `.`
- Se definen dinámicamente

```
var persona = new Object();
persona.nombre = "Aitor";
persona.apellido1 = "Medrano";
```

- Para saber si un objeto contiene una propiedad → operador `in`
- Para eliminar una propiedad → operador `delete`

```
console.log('nombre' in persona); // true
delete persona.nombre;
console.log('nombre' in persona); // false
```

Métodos

- Operaciones de un objeto
- Función anónima asignada a una propiedad
- Dentro de los métodos, `this` referencia al objeto

```
persona.getNombreCompleto = function() {
  return this.nombre + " " + this.apellido1;
}
console.log(persona.getNombreCompleto());
```

2.2 Objetos literales

- Notación más sencilla → similar a JSON
- Par de llaves que rodean 0 o más parejas de *clave:valor* separadas por comas
  - cada clave puede ser una propiedad o un método

```
var nadie = {};
var persona = {
  nombre : "Aitor",
  apellido1 : "Medrano",
  getNombreCompleto : function() {
    return this.nombre + " " + this.apellido1;
  }
};
```

- No olvidar el **punto y coma tras la definición** del objeto literal

Accediendo a los campos

- Para acceder a un campo, operador punto → `.`
- También podemos usar notación de corchete → `[]`
  - Permite acceder a propiedades mediante variables que contienen una cadena

```
var nom = persona.nombre;
var ape1 = persona["apellido1"];
var nombreCompleto = persona.getNombreCompleto();
var nombreCompletoCorchete = persona[“getNombreCompleto”]();
```

- Un objeto puede contener otros objetos como propiedades
  - `variable.objeto.objeto.objeto.propiedad`
  - `variable['objeto']['objeto']['objeto']['propiedad']`
- **Encadenamiento de objetos** (*object chaining*)

Objetos anidados

- Cualquier propiedad puede a su vez ser un objeto
  - Agrupa información

```
var cliente = {
  nombre: "Bruce Wayne",
  email: "bruce@wayne.com",
  direccion: {
    calle: "Mountain Drive",
    num: 1007,
    ciudad: "Gotham"
  }
};
```

```
var cliente = {};
cliente.nombre = "Bruce Wayne";
cliente.email = "bruce@wayne.com";
cliente.direccion = {};
cliente.direccion.calle = "Mountain Drive";
cliente.direccion.num = 1007;
cliente.direccion.ciudad = "Gotham";
```

Propiedades sin Valor

- Al acceder a una propiedad que no existe → `undefined`
- Podemos evitarlo con operador `||`

```
var nada = cliente.formaPago; // undefined
var pagoPorDfecto = cliente.formaPago || "Efectivo";
```

- Al obtener un valor de una propiedad `undefined` → `TypeError`
- Podemos evitarlo con operador `&&`

```
var cliente = {};
cliente.direccion; // undefined
cliente.direccion.calle; //   lanza TypeError
cliente.direccion && cliente.direccion.calle; // undefined
```

2.3 Función Factoría

- 2 personas → *copy/paste* → **NO**
- Mejor crear una función que devuelva un objeto → patrón *Factoría*

```
function creaPersona(nom, ape1) {
  return {
    nombre : nom,
    apellido1 : ape1,
    getNombreCompleto : function() {
      return this.nombre + " " + this.apellido1;
    }
  };
}

var persona = creaPersona("Aitor", "Medrano"),
  persona2 = creaPersona("Domingo", "Gallardo");
```

Métodos que usan objetos

- *JavaScript* es débilmente tipado → comprobar que el objeto recibido es del tipo esperado
  - consultando la propiedad que vayamos a usar

```
function creaPersona(nom, ape1) {
  return {
    nombre : nom, apellido1 : ape1,
    getNombreCompleto : function() {
      return this.nombre + " " + this.apellido1;
    },
    saluda: function(persona) {
      if (typeof persona.getNombreCompleto !== "undefined") {
        return "Hola " + persona.getNombreCompleto();
      } else {
        return "Hola colega";
      }
    }
  };
};
```

http://jsbin.com/tirawe/1/edit?js

```
var persona = creaPersona("Aitor", "Medrano"),
  persona2 = creaPersona("Domingo", "Gallardo");
persona.saluda(persona2); // Hola Domingo Gallardo
persona.saluda({}); // Hola colega
persona.saluda({ getNombreCompleto: "Bruce Wayne" });
// TypeError, la propiedad no es una función
```

Función Constructor

- Parecido a una clase Java
  - Nombre del objeto comienza por mayúsculas
- Al crear el objeto mediante una función constructor podemos usar `instanceof`

```
var Persona = function(nombre, apellido1) {
  this.nombre = nombre;
  this.apellido1 = apellido1;

  this.getNombreCompleto = function() {
    return this.nombre + " " + this.apellido1;
  };

  this.saluda = function(persona) {
    if (persona instanceof Persona) {
      return "Hola " + persona.getNombreCompleto();
    } else {
      return "Hola colega";
    }
  };
}
```

http://jsbin.com/peliq/1/edit?js

`instanceof`

- Permite averiguar si un objeto es una instancia de una función constructor
- `true` cada vez que le preguntemos si un objeto es una instancia de `Object`
  - todos los objetos heredan del constructor `Object()`.

```
var heroe = { nombre: "Batman" };
console.log(heroe instanceof Object); // true
```

- `false` con los tipos primitivos que envuelven objetos
- `true` si se crean con el operador `new`

```
console.log("Batman" instanceof String);  // false
console.log(new String("Batman") instanceof String); // true
```

`new`

- Permite instanciar una función Constructor

```
var persona = new Persona("Aitor", "Medrano"),
  persona2 = new Persona("Domingo", "Gallardo");
persona.saluda(persona2); // Hola Domingo Gallardo
persona.saluda({}); // Hola colega
persona.saluda({ getNombreCompleto: "Bruce Wayne" }); // Hola colega
```

- Si olvidamos `new`, no creará el objeto, sino que asignará las propiedades y métodos al objeto `window`.
- Solución → comprobar el valor de `this` en el constructor de la función

```
var Persona = function(nombre, apellido1) {
  if (this === window) {
    return new Persona(nombre, apellido1);
  }
```

http://jsbin.com/peliq/1/edit?js

2.4 Invocación Indirecta

- Permite invocar al método de un objeto tomando como instancia a otro objeto
  - Permite reutilizar funciones entre objetos
  - Redefine el valor de `this` mediante `apply()` o `call()`

http://jsbin.com/necohu/1/edit?js

```
var heroe = {
  nombre: "Superheroe",
  saludar: function() {
    return "Hola " + this.nombre;
  }
};

var batman = { nombre: "Batman" };
var spiderman = { nombre: "Spiderman" };

console.log(heroe.saludar()); // Hola Superheroe
console.log(heroe.saludar.apply(batman)); // Hola Batman
console.log(heroe.saludar.call(spiderman)); // Hola Spiderman
```

`call()` vs `apply()`

- Forman parte del prototipo de todos los objetos
- Varían en la manera que tienen de añadir parámetros
  - **`apply`**`(método, `**`arrayArgumentos`**`)`
  - **`call`**`(método, `**`arg1, arg2, arg3,`**` ...)`

http://jsbin.com/sulamu/1/edit?js

```
var heroe = {
  nombre: "Superheroe",
  saludar: function() {
    return "Hola " + this.nombre;
  },
  despedirse: function(enemigo1, enemigo2) {
    var malos = enemigo2 ? (enemigo1 + " y " + enemigo2) : enemigo1;
    return "Adios " + malos + ", firmado:" + this.nombre;
  }
};

var batman = { nombre: "Batman" }; var spiderman = { nombre: "Spiderman" };

console.log(heroe.despedirse()); // Adios undefined, firmado:Superheroe
console.log(heroe.despedirse.apply(batman, ["Joker", "Dos caras”]));
// Adios Joker y Dos caras, firmado:Batman
console.log(heroe.despedirse.call(spiderman, "Duende Verde", "Dr Octopus”));
// Adios Duende Verde y Dr Octopus, firmado:Spiderman
```

`bind()`

- *ECMAScript* 5
- En vez de realizar la llamada al método, devuelve la función con el contexto modificado

```
var funcionConBatman = heroe.despedirse.bind(batman);
var despedidaBind = funcionConBatman("Pingüino");
console.log(despedidaBind); // Adios Pingüino, firmado:Batman
```

- Se emplea al usar *callbacks* y en vez de guardar una referencia a `this` en una variable auxiliar (normalmente nombrada como `that`), hacemos uso de `bind` para pasarle `this` al *callback*.

```
var that = this;
function callback(datos) {
  that.procesar(datos);
}
ajax(callback);
```

```
function callback(datos) {
  this.procesar(datos);
}
ajax(callback.bind(this));
```

2.5 Descriptores de Propiedades

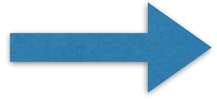
- En un objeto literal, las propiedades se pueden leer y modificar
- Los descriptos de propiedades permiten restringir el acceso.
- *ECMAScript* 5
- Descriptor de **datos**
  - Propiedades que almacenan un valor → `Object.defineProperty()` / `Object.defineProperties()`
- Descriptor de **acceso**
  - Definen dos funciones, para los métodos `get` y `set`

- **IMPORTANTE**: No se pueden combinar, una propiedad es un descriptor de datos o de acceso, no las dos al mismo tiempo.

`Object.defineProperty()`

- `Object.defineProperty(`*`objeto`*`, `*`propiedad`*`, `*`atributos`*`)`
  - `value` → nombre de la propiedad
  - `writable` → booleano que indica si se puede modificar. Por defecto, `false`

```
function creaPersona(nom, ape1) {
  return {
    nombre : nom,
    apellido1 : ape1,
    getNombreCompleto : function() {
      return this.nombre + " " +
             this.apellido1;
    }
  };
}
```



```
function creaPersona(nom, ape1) {
  var persona = {};

  Object.defineProperty(persona, "nombre", {
    value: nom,
    writable: true
  });
  Object.defineProperty(persona, "apellido1", {
    value: ape1,
    writable: false
  });

  return persona;
}
```

Object.defineProperties()

- `Object.defineProperties(objeto, propiedades)`
  - `propiedad: atributos`

http://jsbin.com/loqofe/1/edit?js

```
function creaPersona(nom, ape1) {
  var persona = {};

  Object.defineProperty(persona, "nombre", {
    value: nom,
    writable: true
  });
  Object.defineProperty(persona, "apellido1", {
    value: ape1,
    writable: false
  });

  return persona;
}
```

```
function creaPersona(nom, ape1) {
  var persona = {};

  Object.defineProperties(persona, {
    nombre: {
      value: nom,
      writable: true
    },
    apellido1: {
      value: ape1,
      writable: false
    }
  });

  return persona;
}
```

Autoevaluación: Descriptores de Propiedades

```
function creaPersona(nom, ape1) {
  var persona = {};

  Object.defineProperties(persona, {
    nombre: {
      value: nom,
      writable: true
    },
    apellido1: {
      value: ape1,
      writable: false
    }
  });

  return persona;
}
```

http://jsbin.com/loqofe/1/edit?js

```
var batman = creaPersona("Bruce", "Wayne");
console.log(batman);
batman.nombre = "Bruno";
batman.apellido1 = “Díaz";
console.log(batman);
```

Object.getOwnPropertyDescriptor()

- Si queremos obtener una propiedad → `objeto.propiedad`
- Para obtener la información de una propiedad → `Object.getOwnPropertyDescriptor(objeto, propiedad)`

```
var batman = creaPersona("Bruce", "Wayne");
console.log(Object.getOwnPropertyDescriptor(batman, "nombre"));
```

```
[object Object] {
  configurable: false,
  enumerable: false,
  value: "Bruce",
  writable: true
}
```

`get` y `set`

- Sustituyen a los métodos que modifican las propiedades no asociadas a un dato.
- Dentro de los atributos de una propiedad sin valor
  - Método de acceso → `get`
  - Método de modificación → `set`
- Permite acceder al descriptor como propiedad en vez de como método

```
function creaPersona(nom, ape1) {
  var persona = {};
  Object.defineProperties(persona, {
    nombre: {
      value: nom, writable: true
    },
    apellido1: {
      value: ape1, writable: false
    },
    nombreCompleto: {
      get: function() {
        return this.nombre + " " + this.apellido1;
      },
      set: function(valor) {
        this.nombre = valor; this.apellido1 = valor;
      }
    }
  });
  return persona;
}
```

Uso de `get` y `set`

```
var batman = creaPersona("Bruce", "Wayne");
console.log(batman.nombreCompleto); // Bruce Wayne
batman.nombreCompleto = "Bruno Díaz";
console.log(batman.nombreCompleto); // Bruno Díaz Wayne
```

```
> batman
< ▼ Object {nombre: "Bruce", apellido1: "Wayne", nombreCompleto: "Bruce Wayne"}
    apellido1: "Wayne"
    nombre: "Bruce"
    nombreCompleto: (...)
  ▼ get nombreCompleto: function () { // <1>
      arguments: null
      caller: null
      length: 0
      name: ""
    ▶ prototype: Object.defineProperties.nombreCompleto.get
    ▶ __proto__: function Empty() {}
    ▶ <function scope>
  ▼ set nombreCompleto: function (valor) {
      arguments: null
      caller: null
      length: 1
      name: ""
    ▶ prototype: Object.defineProperties.nombreCompleto.set
    ▶ __proto__: function Empty() {}
    ▶ <function scope>
  ▶ __proto__: Object
```

Iterando sobre las propiedades

- Si necesitamos recorrer todas las propiedades de un objeto → `for ... in`

```
for (var prop in batman) {
  console.log(batman[prop]);
}
```

- O utilizar el método `Object.keys(objeto)`

```
var propiedades = Object.keys(batman);
```

- Las propiedades definidas mediante descriptores no se visualizan al recorrerlas.

`enumerable`

- atributo que indica si la propiedad se puede recorrer
- Si no la indicamos, su valor por defecto es `false`.

```
function creaPersona(nom, ape1) {
  var persona = {};
  Object.defineProperties(persona, {
    nombre: {
      value: nom, enumerable: true
    },
    apellido1: {
      value: ape1, enumerable: true
    },
    nombreCompleto: {
      get: function() {
        return this.nombre + " " + this.apellido1;
      },
      enumerable: false
    }
  });
  return persona;
}
var batman = creaPersona("Bruce", "Wayne");
console.log(Object.keys(batman));
```

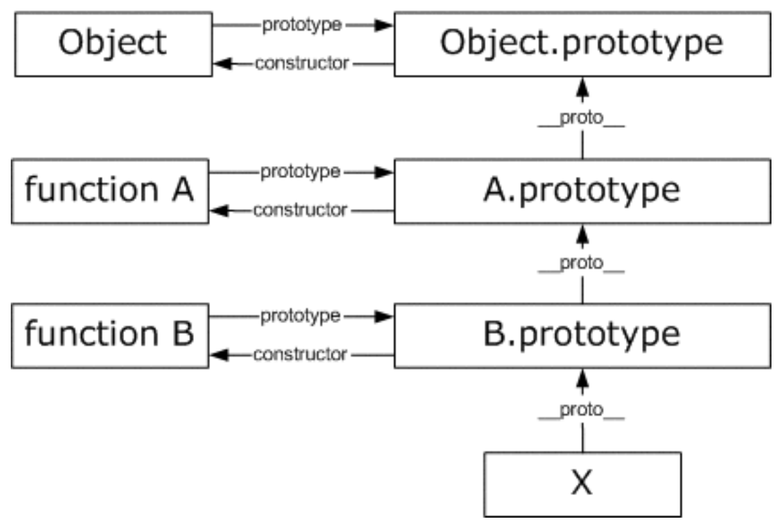
2.6 Prototipos

- Permiten definir tipos de objetos
- Define propiedades y funciones que se aplicarán a todas las instancias de un objeto
- Todos los objetos tienen una propiedad `prototype`
- Por defecto, todos los objetos tienen como prototipo raíz a `Object.prototype`
- `Object.getPrototypeOf(objeto)` → obtiene el prototipo de un objeto

```
console.log(Object.getPrototypeOf({}) == Object.prototype); // true
console.log(Object.getPrototypeOf(Object.prototype)); // null
```

Constructores y `prototype`

- Al llamar a una función mediante la instrucción `new` se invoca como un constructor.
  - El constructor asocia la variable `this` al objeto creado, y a menos que se indique, la llamada devolverá este objeto.
- Este objeto se conoce como una instancia de su constructor.
- Cada instancia creada con este constructor tendrá este objeto como su prototipo.
- Para añadir nuevos métodos al constructor → añadirlos como propiedades del prototipo.
  - **objeto.`prototype`.método**



Ejemplo prototipo

```
var Persona = function(nombre, apellido1) {
  this.nombre = nombre;
  this.apellido1 = apellido1;
}

Persona.prototype.getNombreCompleto = function() {
  return this.nombre + " " + this.apellido1;
};
Persona.prototype.saluda = function(persona) {
  if (persona instanceof Persona) {
    return "Hola " + persona.getNombreCompleto();
  } else {
    return "Hola colega";
  }
};
var persona = new Persona("Aitor", "Medrano"),
  persona2 = new Persona("Domingo", "Gallardo");
persona.saluda(persona2); // Hola Domingo Gallardo
persona.saluda({}); // Hola colega
persona.saluda({ getNombreCompleto: "Bruce Wayne" }); // Hola colega
```

`__proto__`

- Todo objeto contiene una propiedad `__proto__` que incluye todas las propiedades que hereda nuestro objeto

```
var objeto = {};
console.dir(objeto);
```

```
▼ Object i
  ▼ __proto__: Object
    ▶ __defineGetter__: function __defineGetter__() { [native code] }
    ▶ __defineSetter__: function __defineSetter__() { [native code] }
    ▶ __lookupGetter__: function __lookupGetter__() { [native code] }
    ▶ __lookupSetter__: function __lookupSetter__() { [native code] }
    ▶ constructor: function Object() { [native code] }
    ▶ hasOwnProperty: function hasOwnProperty() { [native code] }
    ▶ isPrototypeOf: function isPrototypeOf() { [native code] }
    ▶ propertyIsEnumerable: function propertyIsEnumerable() { [native code] }
    ▶ toLocaleString: function toLocaleString() { [native code] }
    ▶ toString: function toString() { [native code] }
    ▶ valueOf: function valueOf() { [native code] }
    ▶ get __proto__: function __proto__() { [native code] }
    ▶ set __proto__: function __proto__() { [native code] }
```

- Para recuperar este prototipo podemos usar el método `Object.getPrototypeOf()` o directamente navegar por la propiedad `__proto__` (***deprecated***).

`prototype` vs `__proto__`

- Al invocar un método o acceder a una propiedad, el motor busca dentro de las propiedades o métodos del propio objeto.
- Si el motor JS no lo encuentra, seguirá la referencia `__proto__` y buscará el miembro en el prototipo del objeto

- `Array.__proto__`: prototipo de Array →el objeto del que hereda la función constructor `Array`.
- `Array.prototype`: objeto prototipo de todos los arrays → contiene los métodos que heredarán todos los arrays.

http://jsbin.com/sicivi/2/edit?js

```
function Heroe(){
  this.malvado = false;
  this.getTipo = function() {
    return this.malvado ? "Malo" : "Bueno";
  };
}
Heroe.prototype.atacar = function() {
  return this.malvado ? "Ataque con Joker" :
    "Ataque con Batman";
}

var robin = new Heroe();
console.log(robin.getTipo());   // Bueno
console.log(robin.atacar());     // Ataque con Batman

var lexLuthor = new Heroe();
lexLuthor.malvado = true;
console.log(lexLuthor.getTipo());   // Malo
console.log(lexLuthor.atacar());     // Ataque con Joker

var policia = Object.create(robin);
console.log(policia.getTipo());     // Bueno
console.log(policia.__proto__.atacar());  // Ataque con Batman
```

2.7 Herencia

- *JavaScript* es un lenguaje de **herencia prototipada** → un objeto puede heredar directamente propiedades de otro objeto, sin necesidad de crear clases.
- Propiedad `prototype` → asocia atributos y métodos al prototipo de nuestras funciones constructor.

- Para mejorar el uso de la memoria y reducir la duplicidad de los métodos:
  - Crear descriptores de acceso para las propiedades.
  - Llevar los métodos al prototipo de la función constructor.

```
var Persona = function(nombre, apellido1) {
  this.nombre = nombre;
  this.apellido1 = apellido1;

  this.nombreCompleto = function() {
    return this.nombre + " " + this.apellido1;
  };

  this.saluda = function(persona) {
    if (persona instanceof Persona) {
      return "Hola " + persona.getNombreCompleto();
    } else {
      return "Hola colega";
    }
  };
}
```

http://jsbin.com/racexa/1/edit?js

```
var Persona = function(nombre, apellido1) {
  this.nombre = nombre;
  this.apellido1 = apellido1;
}

Object.defineProperties(Persona.prototype, {
  nombreCompleto: {
    get: function() {
      return this.nombre + " " + this.apellido1;
    },
    enumerable: true
  }
});

Persona.prototype.saluda = function(persona) {
  if (persona instanceof Persona) {
    return "Hola " + persona.nombreCompleto;
  } else {
    return "Hola colega";
  }
};

var batman = new Persona("Bruce", "Wayne");
console.log(batman.nombreCompleto); // Bruce Wayne
console.log(batman.saluda()); // Hola colega
var superman = new Persona("Clark", "Kent");
console.log(batman.saluda(superman)); // Hola Clark Kent
```

`Object.create(prototipo)`

- Para crear un objeto que utilice el prototipo de otro hemos de hacer uso del método `Object.create(`*`objetoPrototipo`*`)`.
- El *objetoPrototipo* se convierte en el prototipo del objeto devuelto, y así podemos acceder al objeto padre mediante el método `Object.getPrototypeOf(objeto)`
  - o la propiedad `__proto__` (*deprecated*)

```
var Empleado = Object.create(Persona);
console.log(Empleado.hasOwnProperty('nombreCompleto'));  // false
console.log(Empleado.__proto__ === Persona);  // true
console.log(Object.getPrototypeOf(Empleado) === Persona);  // true
```

Herencia de Constructor

- Al usar funciones constructor, podemos realizar herencia de constructor
  - El hijo comparte las mismas propiedades que el padre.
- El hijo debe realizar una llamada al padre y definir sus propios atributos.

```
var Empleado = function(nombre, apellido1, cargo) {
  Persona.call(this, nombre, apellido1);
  this.cargo = cargo;
}
```

Herencia de Prototipo

- Se hereda el prototipo para compartir los métodos y si fuese el caso, sobreescribirlos.
- Mediante `Object.create(`*prototipo*`, `*propiedades*`)` definir los métodos del hijo y si quisiéramos sobrescribir los métodos que queramos del padre.

```
Empleado.prototype = Object.create(Persona.prototype, {
  saluda: {   // sobreescribimos los métodos que queremos
    value: function(persona) {
      if (persona instanceof Persona) {
        return Persona.prototype.saluda.call(this) +
           " (desde un empleado)";
      } else {
        return "Hola trabajador";
      }
    },
    writable: false,
    enumerable: true
  },
  nombreCompleto: {
    get: function() {
      var desc = Object.getOwnPropertyDescriptor(
          Persona.prototype, ”nombreCompleto");
      return desc.get.call(this) + ", " + this.cargo;
    },
    enumerable: true
  }
});
```

http://jsbin.com/ledavu/1/edit?js

2.8 `this` y el Patrón Invocación

- En *JavaScript*, la variable `this` toma diferentes valores dependiendo de cómo se invoque la función o fragmento donde se encuentra.

- 4 maneras de llamar a una función → **Patrón Invocación**:
  - El patrón de invocación como método
  - El patrón de invocación como función
  - El patrón de invocación como constructor
  - El patrón de invocación con apply y call

Invocación como Método

- Un método es una función que se almacena como propiedad de un objeto
- `this` se inicializa con el objeto al que pertenece la función

```
var obj = {
  valor = 0,
  incrementar: function(inc){
    this.valor += inc;
  }
};

obj.incrementar(3);
console.log(obj.valor); // 3
```

Invocación como Función

- Cuando una función no es una propiedad de un objeto, se invoca como función.
- `this` se inicializa con el objeto global (al trabajar con un navegador, el objeto `window`).

```
function suma(a,b) {
   console.log(a+b); // 8
   console.log(this); // Window {top: Window, window: Window, location: Location, ... }
}
suma(3,5);
```

- Al llamar a una función dentro de otra, `this` sigue referenciando al objeto global
- Si queremos acceder al `this` de la función padre tenemos que almacenarlo antes en una variable.

```
var obj = {
  valor: 0,
  incrementar: function(inc) {
    var that = this;
    function otraFuncion(unValor) {
      that.valor += unValor;
    }
    otraFuncion(inc);
  }
};
obj.incrementar(3);
console.log(obj.valor); // 3
```

Invocación como Constructor

- Al invocar una función mediante `new` se crea un objeto con una referencia al valor de la propiedad `prototype` de la función (es decir, la función constructor)
- `this` referencia a este nuevo objeto.

```
var Persona = function() { // constructor
  this.nombre = 'Aitor';
  this.apellido1 = "Medrano";
}

Persona.prototype.getNombreCompleto = function(){
  return this.nombre + " " + this.apellido1;
}

var p = new Persona();
console.log(p.getNombreCompleto()); // Aitor Medrano
```

Invocación con `apply` / `call`

- `apply` permite reescribir el valor de `this` en tiempo de ejecución.
- `apply` recibe 2 parámetros, el primero es el valor para `this` y el segundo es un array de parámetros.

```
var Persona = function() { // constructor
  this.nombre = "Aitor";
  this.apellido1 = "Medrano";
}
Persona.prototype.getNombreCompleto = function(){
  return this.nombre + " " + this.apellido1;
}
var otraPersona = {
  nombre: "Rubén",
  apellido1: "Inoto"
}

var p = new Persona();
console.log(p.getNombreCompleto()); // Aitor Medrano
console.log(p.getNombreCompleto().apply(otraPersona)); // Rubén Inoto
```

Autoevaluación - `this`

```
var nombre = 'Bruce Wayne';
var obj = {
  nombre: 'Bruno Díaz',
  prop: {
    nombre: 'Aitor Medrano',
    getNombre: function() {
      return this.nombre;
    }
  }
};

console.log(obj.prop.getNombre());
var test = obj.prop.getNombre;
console.log(test());
```

http://jsbin.com/xuzuhu/1/edit?js

2.9 Arrays

- En JavaScript, un array es un objeto
- Puede contener tipos diferentes
- Puede tener huecos
- Permite añadir elementos en caliente
  - Si añadimos elementos en posiciones mayores al tamaño del array, se rellena con `undefined`
- *0-index*
- longitud → propiedad `length`
- Para borrar un elemento → `delete[posicion]`
  - Deja un hueco

```
var cosas = new Array();
var tresTipos = new Array(11, "hola", true);
var longitud = tresTipos.length; // 3
var once = tresTipos[0];
```

```
var tresTipos = [11, "hola", true];
var once = tresTipos[0];
```

```
tresTipos[3] = 15;
tresTipos[tresTipos.length] = "Bruce";
var longitud2 = tresTipos.length; // 5
tresTipos[8] = "Wayne";
var longitud3 = tresTipos.length; // 9
var nada = tresTipos[7]; // undefined
// ¿tresTipos?
```

http://jsbin.com/zegepa/2/edit?js

Manipulación Individual

Métodos	Propósito
pop()	Extrae y devuelve el último elemento del array
push(*elemento*)	Añade el *elemento* en la última posición
shift()	Extrae y devuelve el primer elemento del array
unshift(*elemento*)	Añade el *elemento* en la primera posición

http://jsbin.com/nepazi/4/edit?js

```
var notas = ['Suspenso', 'Aprobado', 'Bien', 'Notable', 'Sobresaliente'];

notas.push('Matrícula de Honor');
var matricula = notas.pop(); // "Matrícula de Honor"
var suspenso = notas.shift(); // "Suspenso"
notas.unshift('Suspendido');
```

Manipulación conjunto

Métodos	Propósito
concat(*array2*[,… , *arrayN*])	Une dos o más arrays
join(*separador*)	Concatena las partes de un array en una cadena, indicándole como parámetro el *separador* a utilizar
reverse()	Invierte el orden de los elementos del array, mutando el array
sort()	Ordena los elementos del array alfabéticamente, mutando el array
slice(*inicio*, *fin*)	Devuelve un nuevo array con una copia con los elementos comprendidos entre *inicio* y *fin* (con índice `0`, y sin incluir *fin*).
splice(*índice*, *cantidad*, *elem1*[, … , *elemN*])	Modifica el contenido del array, añadiendo nuevos elementos mientras elimina los antiguos seleccionando a partir de *índice* la *cantidad* de elementos indicados. Si *cantidad* es 0, sólo inserta los nuevos elementos.

Ejemplos Manipulación Array

http://jsbin.com/kefexa/6/edit?js

```
var notas = ['Suspenso', 'Aprobado', 'Bien', 'Notable', 'Sobresaliente'];
notas.reverse();
console.log(notas);  // ["Sobresaliente", "Notable", "Bien", "Aprobado", "Suspenso"]
notas.sort();
console.log(notas);  // ["Aprobado", "Bien", "Notable", "Sobresaliente", "Suspenso"]
notas.splice(0, 4, "Apto");
console.log(notas); // ["Apto", "Suspenso"]
```

```
var nombreChicos = ["Juan", "Antonio"];
nombreChicos.push("Pedro");
var nombreChicas = ["Ana", "Laura"];
var nombres = nombreChicos.concat(nombreChicas);
var separadoConGuiones = nombres.join("-");
nombres.reverse();
var alfa = nombres[3];
nombres.sort();
var iguales = (alfa == nombres[2]);
```

http://jsbin.com/gawaju/1/edit?js

Búsqueda

Método	Propósito
indexOf(*elem*[, *inicio*])	Devuelve la primera posición (0..n-1) del elemento comenzando desde el principio o desde *inicio*
lastIndexOf(*elem*[, *inicio*])	Igual que `indexOf` pero buscando desde el final hasta el principio

```
var frutas = ["naranja", "pera", "manzana", "uva", "fresa", "naranja"];
var primera = frutas.indexOf("naranja");
var ultima = frutas.lastIndexOf("naranja");
```

Iteración

Método	Propósito
forEach(*función*)	Ejecuta la función para cada elemento del array
map(*función*)	Devuelve un nuevo array. Ejecuta la función para cada elemento del array, y el nuevo valor se inserta como un elemento del nuevo array que devuelve.
every(*función*)	Verdadero si la función se cumple para todos los valores. Falso en caso contrario (Similar a una conjunción → Y)
some(*función*)	Verdadero si la función se cumple para al menos un valor. Falso si no se cumple para ninguno de los elementos (Similar a un disyunción → O)
filter(*función*)	Devuelve un nuevo array con los elementos que cumplen la función
reduce(*función*)	Ejecuta la función para un acumulador y cada valor del array (de inicio a fin) se reduce a un único valor

- `funcion(valor, indice, array)` → valor del elemento del array, indice del elemento, el propio array

map y forEach

```
var heroes = ["Batman", "Superman", "Ironman", "Thor"];
function mayus(valor, indice, array) {
  return valor.toUpperCase();
}
var heroesMayus = heroes.map(mayus);
console.log(heroesMayus); // ["BATMAN", "SUPERMAN", "IRONMAN", "THOR"]
```

```
var heroes = ["Batman", "Superman", "Ironman", "Thor"];
heroes.forEach(function(valor, indice) {
  console.log("[", indice, "]=", valor);
});
```

```
var numeros = [1, 3, 5, 7, 9];
var suma = 0;
numeros.forEach(function(valor) {
  suma += valor
});
```

every, filter, reduce

```
var frutas = ["naranja", "pera", "manzana", "uva", "fresa", "naranja"];
function esCadena(valor, indice, array) {
  return typeof valor === "string";
}
console.log(frutas.every(esCadena)); // true
```

```
var mezcladillo = [1, "dos", 3, "cuatro", 5, "seis"];
console.log(mezcladillo.filter(esCadena)); // ["dos", "cuatro", "seis"]
```

```
var numeros = [1, 3, 5, 7, 9];
var suma = 0;
numeros.reduce(function(anterior, actual) {
  return anterior + actual
});
```

¿Preguntas?