UD1. Javascript

OBJETOS Y CLASES

Copiar objetos

- Se puede hacer en ES6 con spreading:

```
const copyOfObject = {...originalObject};

const copyOfArray = [...originalArray];
```

- Se puede hacer con Object.assign()

```
Object.assign({},originalObject)
```

- En caso de necesidad de un ‘deep copy’ se tiene que hacer a mano.

Clonado de un objeto y que al modificar uno no se modifique el otro.

Object Destructuring

```
// Desestructuración de arrays
const foo = ['uno', 'dos', 'tres'];
const [rojo, amarillo, verde] = foo;
console.log(rojo); // "uno"
console.log(amarillo); // "dos"
console.log(verde); // "tres"
```

```
// Desestructuración de objetos
const o = {p: 42, q: true, a: {r: 20, s:
'abc'}};
const {p, q} = o;
console.log(p,q); // 42 true
const {p: foo, q: bar} = o; // Nuevos nombres
console.log(foo,bar); // 42 true
var {a} = o;
var {a: {r: R}} = o; // Objetos anidados y
cambio de nombre
console.log(a,R);
```

Object Literal enhacement

```
const a = 'foo';
const b = 42;
const c = {};
const object1 = { a, b, c }; // No hace falta hacer a: a, b:
b ...
console.log(object1); // Object { a: "foo", b: 42, c: {} }
```

Clases con class (ES6)

- Todas las clases son **funciones**, que son **objetos**.
- Javascript es un lenguaje basado en prototipos o classless. Los objetos no se crean instanciando clases, sino **clonando** otros objetos.
- Cada objeto tiene una propiedad interna llamada Prototype que puede ser utilizada para **extender las propiedades** y **métodos** del objeto.
- La palabra reservada **class** en **ES6** es una comodidad sintáctica.
- Muchos programadores **no** la recomiendan porque esconde lo que realmente está pasando.

```
const x = function() {}
const y = class {}
const constructorFromFunction = new x();
const constructorFromClass = new y();
```

Ejemplo de creación de clases

```
function Hero(name, level) {
   this.name = name;
   this.level = level;
  }
```

```
class Hero {
   constructor(name, level) {
       this.name = name;
       this.level = level;
   }
  }
```

Ejemplo de creación de métodos

```
function Hero(name, level) {
   this.name = name;
   this.level = level;
  }

  Hero.prototype.greet = function()
{
   return `${this.name} says
hello.`;
  }
```

```
class Hero {
   constructor(name, level) {
       this.name = name;
       this.level = level;
   }
   greet() {
   return `${this.name} says hello.`;
   }
  }
```

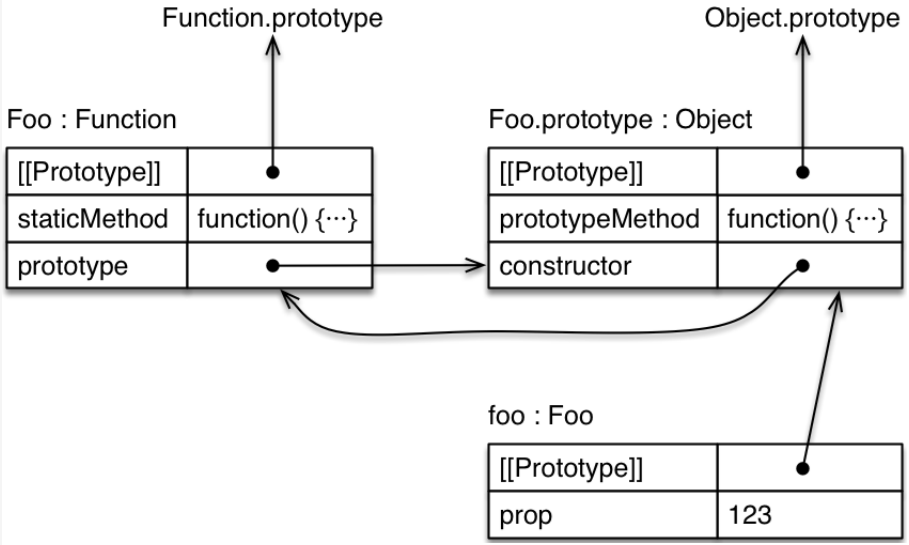
Herencia

```
// Creando un nuevo constructor a
partir de su padre
function Mage(name, level, spell) {
   // Enlazar constructors con call()
   Hero.call(this, name, level);
   this.spell = spell;
  }
```

```
class Mage extends Hero {
   constructor(name, level, spell) {
       // Enlazar constructors con super
       super(name, level);
       this.spell = spell;
   }
  }
```

Atributos estáticos

```
class Foo {
    constructor(prop) {
        this.prop = prop;
    }
    static staticMethod() {
        return 'classy';
    }
    prototypeMethod() {
        return 'prototypical';
    }
}
const foo = new Foo(123);
```



Clases y atributos privados

- Por defecto, en ES6, **todo es público**.
- En ES2019 han incotporado # para hacer las variables internas privadas, pero **no son compatibles** en todos los navegadores a día de hoy.
- Si queremos atributos privados tenemos que hacer uso de las **funciones internas** y los **scopes**.

```
class SmallRectangle {
   constructor() { let width = 20; let height = 10;
                   this.getDimension = () => { return
{width: width, height: height};};
                   this.increaseSize = () => {
width++; height++; }; }
  }
  const rectangle = new SmallRectangle();
  console.log(rectangle.getDimension());
  console.log(rectangle.height);    // => undefined
  console.log(rectangle.width);     // => undefined
```

Closure

- Una forma de hacer variables privadas es imitar el comportamiento de las clases.

  https://www.w3schools.com/js/js_function_closures.asp

  https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Funciones

  https://github.com/getify/You-Dont-Know-JS/blob/2nd-ed/get-started/ch3.md#closure

```
var add = (function () {   // Función autoinvocada
   var counter = 0; // Closure
   return function () {counter += 1; return counter} // add será esta función
  })();
  console.log(add());   // 1
  console.log(add());   // 2
```

Closure

- Hace uso de las funciones anidadas.
- La función interna hereda las variables de la externa.
- Solo podemos acceder a la función interna desde la externa.
- La función externa no puede acceder a los atributos de la función interna.
- Los atributos declarados en this. se quedan en el contexto de ejecución de la función interna.

```
function triangulo(a,b){
   this.a = a;
   this.b = b; // con this se queda en el prototype y es accessible
   console.log(this);
   var hipo = Math.sqrt(this.a*this.a + this.b*this.b); // no es accessible
   this.hipotenusa = function (){
       return `La Hipotenusa es: ${hipo}`; // La función si que tiene acceso
   }
}
var t = new triangle(10,20); // prueba a quitar el new
console.log(t.hipotenusa(),t.hipo); // 22.36 undefined
```

Setters y Getters

- Si tenemos atributos privados o que tienen que ser obtenidos con un control previo podemos utilizar **Setters** y **Getters**.
- Se pueden utilizar com si fuesen **propiedades de la clase**.

```
class Producto {
    constructor(nombre,precio){
        this.nombre = nombre; this. precio = precio;
    }
    set setPreu(precio){
        if(isNaN(precio)) this. precio = 0;
        else this. precio = precio
    }
    get getPrecio(){
        return parseFloat(this.precio);
    }
}
let p1 = new Producto(PC,1000);
p1.setPrecio = 900;
```

This

- En una función representa el contexto de ejecución. El contexto depende de como es llamada.
- Si se ejecuta fuera de un objeto, **this** es el objecto **Window**.
- Ejecutado en modo **strict**, **this** siempre necesita un objeto como contexto.

```
function classroom(teacher) {
   //"use strict";   // prueba el modo estricto
   this.plant = 3;   // sin new, this es window
   console.log(this);
   return function study() {
       console.log(
           `${ teacher } says to study ${ this.topic
} in plant ${this.plant}`
       );};}
let assignment =  classroom("Kyle");  // Prueba a
poner el new
console.log(assignment);
assignment();
let clase = { topic: 'mates',
   plant: '5',   // prueba a comentar esta línea
   assignment: assignment
}
clase.assignment();
```

This según como lo invoquen

- **Invocación simple**: (En el código global) This es el objeto global window o undefined en modo strict.
- **Invocación como método**: (Dentro de un objeto) This es el objeto que contiene el método.
- Invocación indirecta: (**.call()** o **.apply()**) This es el primer argumento de la invocación.
- En el **constructor**: Es el objeto que se está creando.

```
// Simple Invocation
function simpleInvocation() {console.log(this);}
simpleInvocatoin();
// Method Invocation
const methodInvocation = { method(){ console.log(this);}};
methodInvocation.method();
// Indirect Invocation
const context = { value1: 'A', value2: 'B' };
function indirectInvocation() { console.log(this);}
indirectInvocation.call(context);
indirectInvocation.apply(context);
// Constructor Invocation
function constructorInvocation() { console.log(this);}
new constructorInvocation();
```

This i That

- En ocasiones, las funciones anidadas tienen que acceder al this de la función superior y no pueden.
- Algunos programadores utilizan **self** (no recomendado), **this** o **that** para guardar el this.
- Lo más recomendable es utilizar nombres de variables más semánticas.

```
(function () {
 "use strict";
 document.addEventListener("DOMContentLoaded", function () {
   var colours = ["red", "green", "blue"];
   document.getElementById("header")
                  .addEventListener("click", function () {
     // this es una referencia al clicado
    var that = this;
     colours.forEach(function (element, index) {
       console.log(this, that, index, element);
       // this es undefined
       // that es el que se ha clicado
     });   });  }); })();
```

This y funciones flecha

```
function UiComponent() {
   var _this = this;
   var button =
document.getElementById('myButton');
   button.addEventListener('click', function ()
{
       console.log('CLICK');
       _this.handleClick();
   });
}
UiComponent.prototype.handleClick = function ()
{
   ...
};
```

```
function UiComponent() {
   var button =
document.getElementById('myButton');
   button.addEventListener('click', () => {
       console.log('CLICK');
       this.handleClick(); // (A)
   });
}
```

This y funciones flecha

- This es el objeto del contexto externo de la función.
- En modo estricto, this puede dar problemas.
- Son muy útiles para callbacks dentro de métodos, ya que aún hacen referencia al objeto del método. (diapositiva anterior)

https://blog.bitsrc.io/arrow-functions-vs-regular-functions-in-javascript-458ccd863bc1

```
var variable = "Global Level Variable";
let myObject = {
   variable: "Object Level Variable",
   arrowFunction: () => {
       console.log(this.variable);
   },
   regularFunction() {
       console.log(this.variable);
   }
};
myObject.arrowFunction();
myObject.regularFunction();
```