ANTES DE LAS CLASES

FUNCIÓN CONSTRUCTORA OBJETOS

```
//Función constructora
function Punto(coordX,coordY){
    this.x=coordX;
    this.y=coordY;
    this.mostrarCoordenadas=()=> `(${this.x},${this.y})`;
}

let a=new Punto(10,20);
let b=new Punto(-3,6);

console.log(a.mostrarCoordenadas());
console.log(b.mostrarCoordenadas());
```

MÉTODO ANTIGUO DE HERENCIA

6.3.3.1 IDEA DE PROTOTIPO

Desde hace tiempo JavaScript utiliza un concepto muy interesante para conseguir implementar lo que en otros lenguajes se conoce como herencia.

En el lenguaje Java (no JavaScript) y en otros lenguajes, todo objeto pertenece a una clase. Para definir un objeto es obligatorio primero crear una clase. Además, se puede indicar que una clase es **heredera** de otra clase, por lo que esa clase dispondrá de las propiedades y métodos definidos en la clase de la que hereda.

JavaScript no nació con esa idea. La idea es que todos los objetos procedentes del mismo tipo de función constructora, tienen un mismo prototipo con el que enlazan. El prototipo de un objeto es una serie de métodos y propiedades comunes.

En los lenguajes que usan clases, el código de los métodos se copia a los objetos de esa clase. Sin embargo en JavaScript lo que se hace es enlazar con su prototipo. El prototipo es la parte común de los objetos del mismo tipo. Lo interesante en JavaScript es que podemos modificar el prototipo sobre la marcha, y los objetos que enlazan con ese prototipo inmediatamente estarán al día porque el enlace con su prototipo es dinámico.

En el código anterior, la variable ***a*** es un objeto de clase ***Punto***, al igual que la variable ***b***. El acceso al prototipo de un objeto se puede hacer con la propiedad **__proto__** (hay dos guiones al principio y dos al final de la palabra ***proto***). Si mostramos esa propiedad para la variable ***a***:

```
console.log(a.__proto__);
```

Se nos muestra:

```
Punto {}
```

Con ello se nos dice que ***a*** usa el prototipo de clase **Punto**, y que ese prototipo no tiene definido ningún método ni propiedad. Los objetos de tipo punto toman las propiedades y métodos definidos en la función constructora, pero no hay propiedades comunes.

Una forma equivalente de obtener el prototipo es mediante el método **getPrototypeof** que es un método de la clase genérica **Object**. Se usa de esta forma:

```
Object.getPrototypeOf(a)
```

El resultado será el mismo.

6.3.3.2 MODIFICAR PROTOTIPOS

Para modificar prototipos basta con indicar la propiedad **prototype** y después definir propiedades y métodos a voluntad. Esta propiedad, como es lógico, solo está disponible en las funciones

constructoras (en realidad está disponible sobre cualquier función, pero eso es otra cuestión). Así si escribimos este código:

```
console.log(Punto.prototype);
```

Obtendremos el prototipo de la función Punto, que será un objeto vacío porque no hemos definido nada en él.

Para definir, por ejemplo, un nuevo método y una nueva propiedad podemos indicarlos y darles valor. Por ejemplo:

```
Punto.prototype.sumaXY=function(){
    return this.x+this.y;
}
Punto.prototype.z=0;
```

Hemos definido un método llamado ***sumaXY*** y lo hemos definido para el prototipo de los objetos basados en **Punto**. También hemos creado en ese mismo prototipo una propiedad llamada **z** con valor de cero. Si mostramos ahora el prototipo veremos:

```
Punto { sumaXY: [Function], z: 0 }
```

Dará igual que lo hagamos con la expresión **Punto.prototype** o con (si ***p*** es un objeto de tipo ***Punto***) **p.__proto__**

Ahora el prototipo de ***Punto*** ya tiene un método y una propiedad. Lo interesante es que todos los objetos basados en ***Punto*** disponen de esa propiedad y método:

```
let a=new Punto(10,20);
let b=new Punto(-3,6);
console.log(a.sumaXY()); //Muestra 30, resultado de sumar 10+20
console.log(b.sumaXY()); //Muestra 3, resultado de sumar -3+6
console.log(a.z); //Muestra 0
console.log(b.z); //Muestra 0
```

Un detalle muy importante es lo ocurre si modificamos en un objeto la propiedad heredada. Por ejemplo:

```
a.z=7;
```

Ahora la variable ***a***, ya no coge la propiedad **z** del prototipo, tiene un valor propio de esa propiedad. Aunque modifiquemos la propiedad **z** a través del prototipo, la variable ***a*** no lo reflejará porque su propiedad **z** ya es independiente de su prototipo. Sin embargo, todos los demás objetos usarán la propiedad z del prototipo.

Para aclarar este punto observemos este código:

```
console.log(a);
console.log(b);
```

El resultado de este código sería:

```
Punto { x: 10, y: 20, mostrarCoordenadas: [Function], z: 7 }
```

```
Punto { x: -3, y: 6, mostrarCoordenadas: [Function] }
```

La primera línea del resultado muestra las propiedades y métodos de ***a***, la segunda los de ***b***. El objeto ***a*** ha definido una propiedad que no tiene ***b***. No obstante, ***b*** la tiene:

```
console.log(b.z); //Muestra 0
```

La propiedad **z** en el caso de ***b*** la obtiene del prototipo. Lo mismo pasaría con los métodos. Si un objeto redefine un método, entonces, usa su versión de forma prioritaria sobre la del prototipo.

Pero siempre podemos acceder a las propiedades y métodos de los prototipos de un objeto:

```
console.log(a.__proto__.z); //Escribe 0
```

Un detalle muy interesante es que podemos modificar el prototipo incluso de los objetos estándar. Ejemplo:

```
Array.prototype.obtenerPares=function(){
    return this.filter((x)=>(x%2==0));
}

let a=[1,2,3,4,5,6,7,8,9];
console.log(a.obtenerPares());
```

En el código anterior conseguimos que todos los arrays dispongan de un nuevo método llamado **obtenerPares,** el cual devuelve un nuevo array en el que solo quedan los números pares del array original (para ello usa internamente el método **filter** de los arrays).