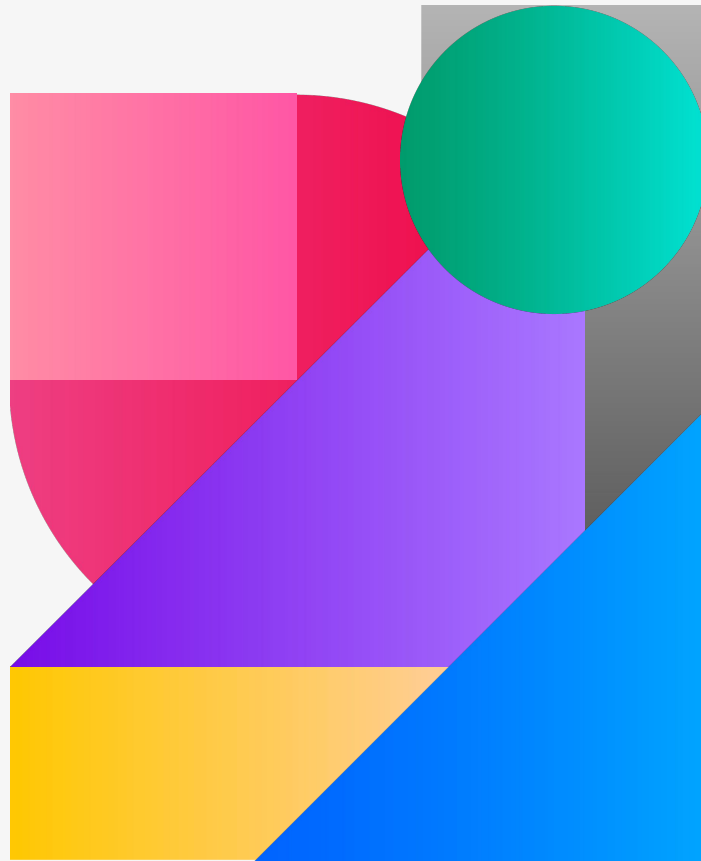


Temas varios

Cordero Hernández, Marco R.





CONTENIDOS

01

**Almacenamiento local y
redirección**

02

Closures

03

Clases y prototipos





01

ALMACENAMIENTO LOCAL Y REDIRECCIÓN

Persistir información en el navegador

Es posible almacenar información de manera local a través del navegador. Esto habilita la posibilidad de crear sesiones, recordar preferencias, *sugerir contenido basado en vistas anteriores*, etc. Esto se logra a través del *frontend*.

- **Cookies** - `document.cookie`
- **Local Storage** - `localStorage.setItem(key, value)`
- **Session Storage** - `sessionStorage.setItem(key, value)`



Detalles de cada método

	Cookies	LocalStorage	SessionStorage
Capacidad	4KB	5-10MB (Depende del navegador)	5MB
Accesibilidad	Todas las pestañas	Todas las pestañas	Solo en la pestaña donde se define
Expiración	Configurada manualmente	Nunca expira	Al cerrar la pestaña
Transferible en solicitudes	Sí	No	No
Almacenamiento	Navegador y Servidor	Solo en Navegador	Solo en Navegador

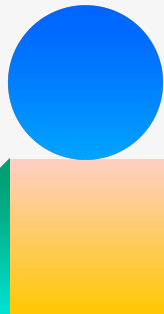


Redirección

Usando **JS** es posible hacer uso de dos mecanismos para redireccionar a otras URL's:

- **`window.location.href`** - Propiedad; Simula un click en algún link
- **`window.location.replace(url)`** - Método; Simula una redirección

La diferencia entre los métodos es que la función ***replace*** sustituye la URL desde donde se ejecuta sin guardarla, es decir, al retroceder en el historial de navegación, la página previa ya no será accesible.



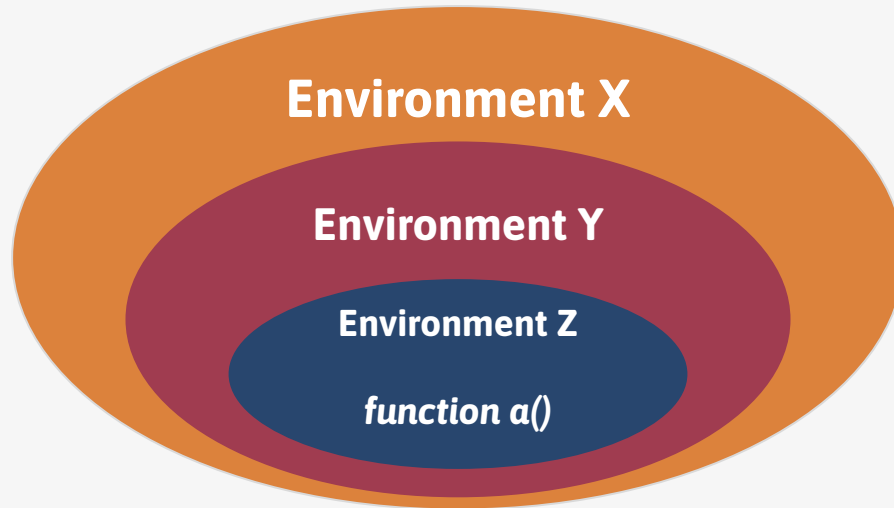


02

CLOSURES

Concepto

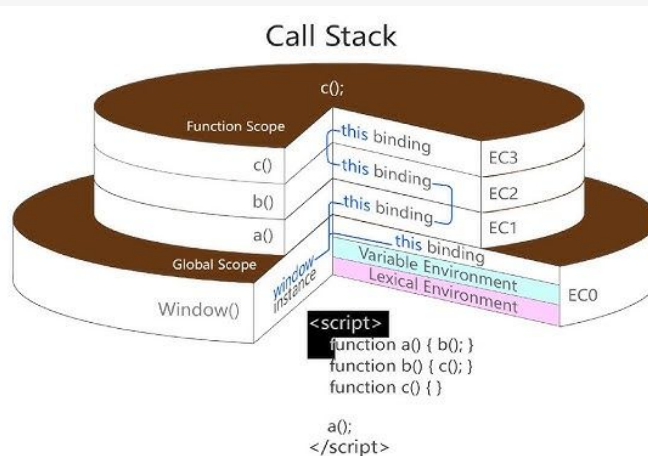
En **JS**, las *closures* (cerraduras) son bloques de código que recuerdan el lugar en el que fueron creados y tienen acceso a las variables que existían en su mismo nivel o **environment** (ambiente) al momento de su creación y sus niveles externos .



Concepto

El término para especificar el alcance de una función se le conoce como **Lexical Environment**. Se compone de dos elementos:

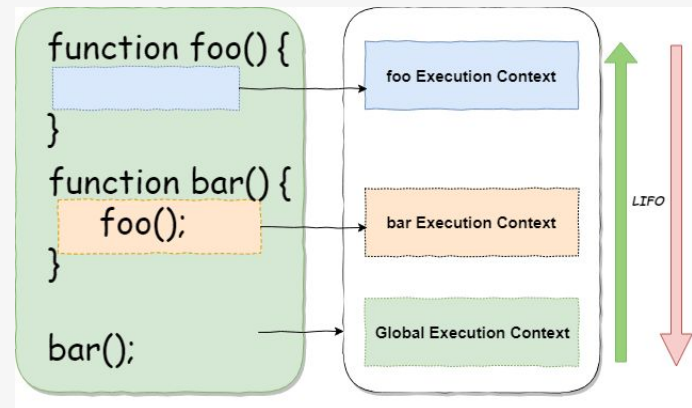
- **Environmental Record**
- Referencia al **lexical environment** inmediatamente exterior



Concepto

Todas las funciones y variables declaradas se almacenan en el **environment record**.

La referencia al **Lexical Environment** ayuda a “marcar la línea” que se sigue para encontrar funciones y variables a las que se hace referencia en un script.



Ejemplo

¿Qué imprimiría este código?

- 12
- 7
- 2
- Ninguna de las anteriores

```
let c = 10;

function sumaC(a) {
    return c + a;
}

c = 5;
c = sumaC(2);
console.log(c);
```

Ejemplo más complejo

¿En qué idioma se verá el saludo?

- En español
- En inglés
- En italiano
- Ninguna de las anteriores

```
function salutare() {  
  let greeting = "Hola mundo!";  
  
  return function() {  
    console.log(greeting);  
  }  
}  
  
let greeting = 'Hello world!';  
let greet = salutare();  
greet();
```

Vista abstracta

```
function salutare() {  
  let greeting = "Hola mundo!";  
  
  return function() {  
    console.log(greeting);  
  }  
}  
  
let greeting = 'Hello world!';  
let greet = salutare();  
greet();
```

Global Environment

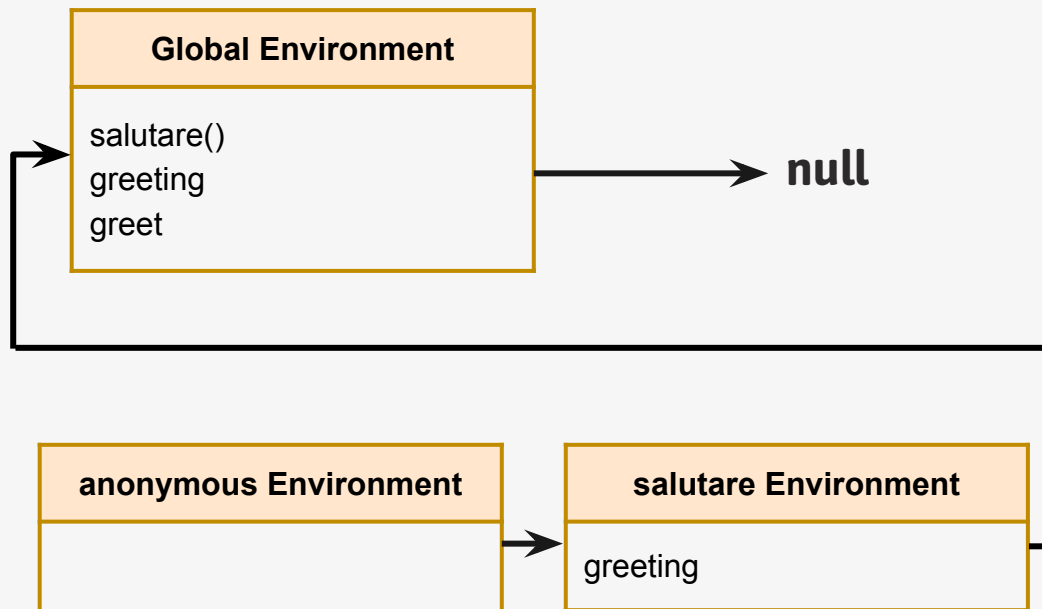
salutare()
greeting
greet

→ null

Vista abstracta

```
function salutare() {  
  let greeting = "Hola mundo!";  
  
  return function() {  
    console.log(greeting);  
  }  
}
```

```
let greeting = 'Hello world!';  
let greet = salutare();  
greet();
```



¿Qué otras cosas crean Closures?

- Estructuras de selección y control
 - if, else, for, while, do...while
- Bloques de código delimitados por llaves
- **IIFE** (Immediately-Invoked **F**unction **E**xpressions)

```
(function() {  
    let greeting = "Hello world!";  
  
    console.log(greeting);  
})();
```



03

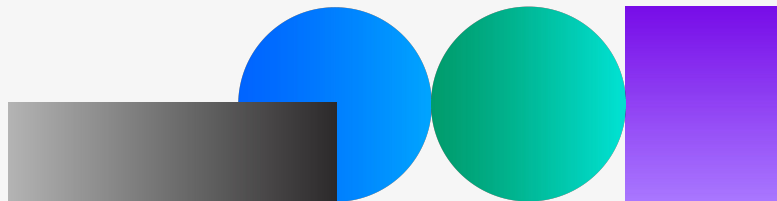
CLASES Y PROTOTIPOS

Prototipos

Objetos que pueden ser usados como referencia para otros objetos a través de su propiedad **[[Prototype]]**. Funcionan como una base abstracta que posteriormente se puede extender.

La propiedad **[[Prototype]]** de un objeto puede ser **null** o puede apuntar a otro objeto.

Esta propiedad no se puede utilizar directamente, pero se puede usar la propiedad **__proto__** para modificarla.

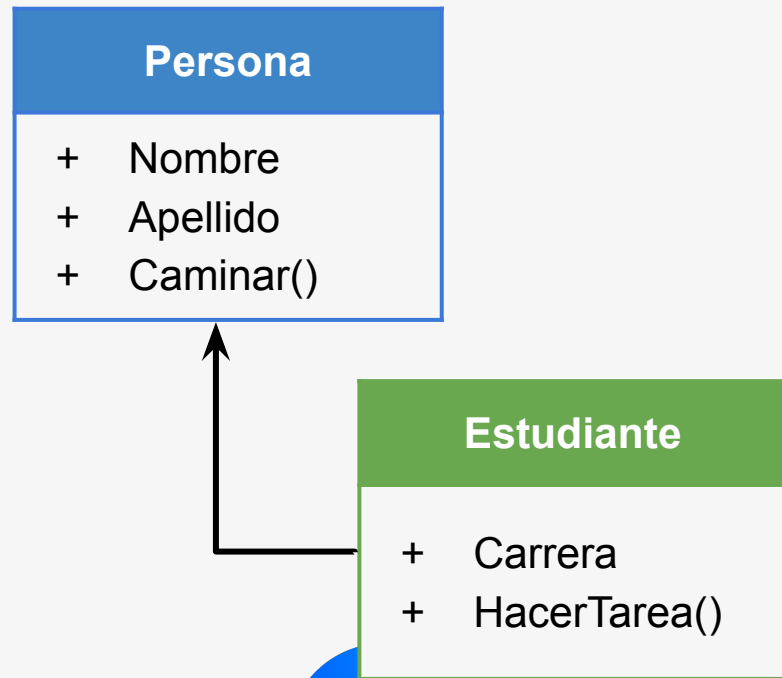


Prototipos

```
// Prototipo (Clase padre)
let Persona = {
  "Nombre": null,
  "Apellido": null,
  "Caminar": function () {
    console.log('La persona está caminando');
  }
};

// Clase extendida
let Estudiante = {
  "Carrera": null,
  "HacerTarea": function () {
    console.log("El estudiante está haciendo tarea");
  }
};

// Asignación de prototipo
Estudiante.__proto__ = Persona;
```

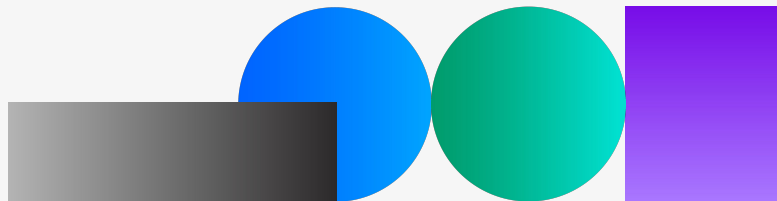


Prototipos

También es posible crear objetos utilizando el operador **new** y una *función*.

La función puede tener un prototipo, y este será el que se pasará al objeto creado como base.

Caso de uso: Cuando se crea un arreglo usando **new Array()**, internamente se “heredan” objetos más abstractos como **Array.prototype** y este a su vez hereda de **Object.prototype**

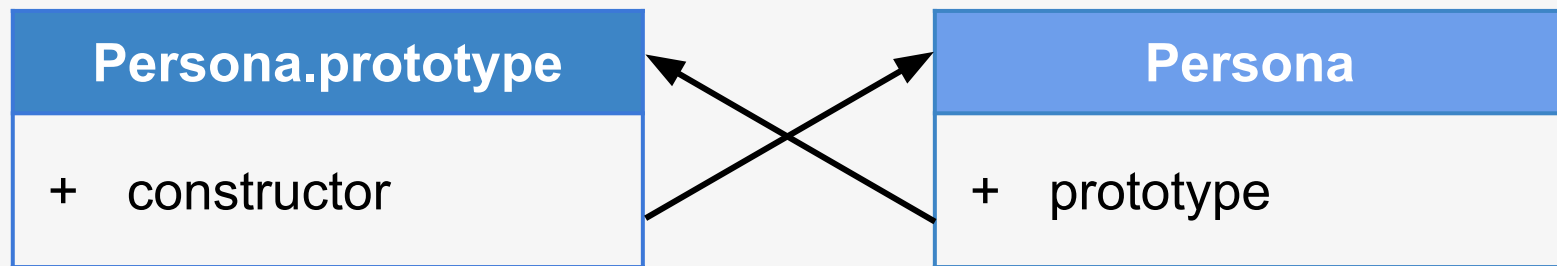


Prototipos

```
let Persona = {  
  "Nombre": null,  
  "Apellido": null,  
  "Caminar": function () {  
    console.log('La persona está caminando');  
  }  
};  
  
function Estudiante() {}  
Estudiante.prototype = Persona;  
  
let estudiante = new Estudiante();  
  
estudiante.Caminar();
```

Prototipos

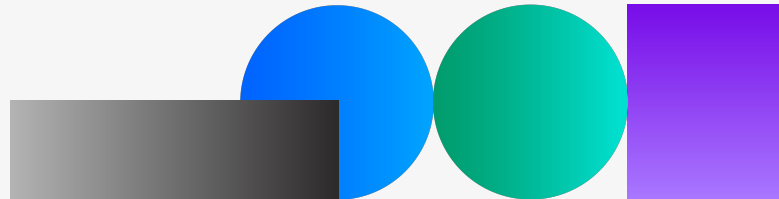
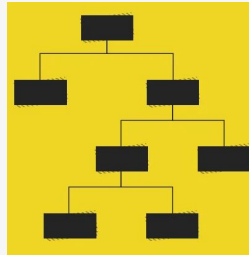
Por defecto, toda función tiene un prototipo *aunque no se especifique manualmente*. A su vez, este prototipo tiene un constructor que apunta de regreso a la función.



Clases y Prototipos

En los **lenguajes orientados a objetos**, las clases pueden ser vistas como plantillas para crear objetos.

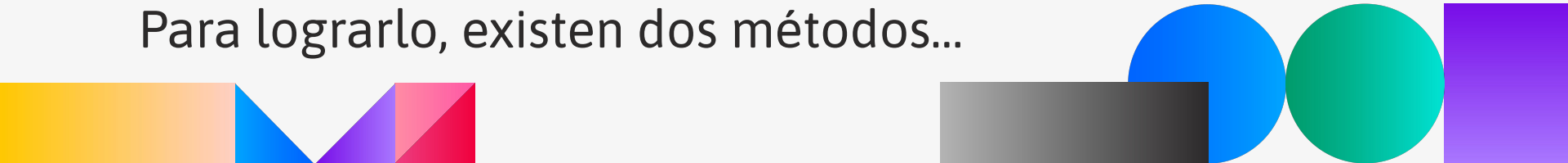
Contienen **atributos** y **métodos** que se “copian” a todos los objetos que se crean a partir de una clase base.



Clases y Prototipos

Lo anterior se aproxima a lo que se ha visto hasta el momento con el manejo de **prototipos**: se puede crear un objeto utilizando el operador **new** y una función, de tal manera que se obtiene un objeto que contendrá lo mismo que la función que se utilice, además de lo definido en el prototipo.

Para lograrlo, existen dos métodos...



Clases y Prototipos

```
/* Método 1 */  
function Estudiante() {  
  this.Carrera = null;  
  this.HacerTarea = function () {  
    console.log('El estudiante  
    está haciendo tarea');  
  };  
}  
  
let estudiante = new Estudiante();  
estudiante.HacerTarea();
```

```
/* Método 2 */  
function Estudiante() {  
  Carrera = null;  
  function HacerTarea() {  
    console.log('El estudiante  
    está haciendo tarea');  
  }  
  
  return {  
    Carrera, HacerTarea  
  };  
}  
  
let estudiante = Estudiante();  
estudiante.HacerTarea();
```


Clases y Prototipos

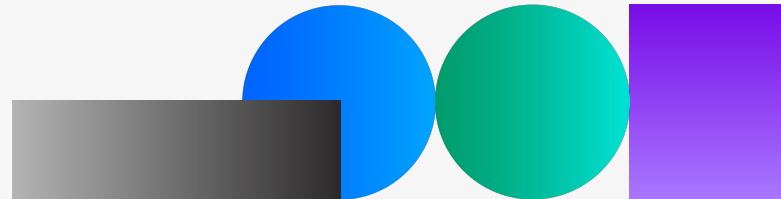
Ambos métodos crean objetos a partir de una función.

En el primero se utiliza el operador **new**.

En el segundo *no porque la función ya regresa un objeto con las propiedades necesarias.*

Esta forma de crear objetos *copia* todas las propiedades de la función en cada nuevo objeto.

La forma *óptima* de manejar la creación de objetos a partir de funciones (**clases**) es utilizando **prototipos**.



Clases y Prototipos

```
// Clase padre
function Persona() {
  this.Nombre = null;
  this.Apellido = null;
}

// Método a heredar
Persona.prototype.Caminar =
function () {
  console.log('La persona
  está caminando');
}

// Clase hijo
function Estudiante() {
  this.Carrera = null;
}
```

```
// Método especializado
Estudiante.prototype.HacerTarea =
function () {
  console.log('El estudiante
  está haciendo tarea');
}

// Herencia
Estudiante.prototype.__proto__ =
Persona.prototype;

// Instanciar clase
let estudiante = new Estudiante();
estudiante.Caminar();
estudiante.HacerTarea();
```

Clases y Prototipos

La secuencia en la que se buscarían propiedades en los objetos sería la siguiente:

- Primero se buscaría en **Estudiante**
- Posteriormente en **Estudiante.prototype**
- Luego en **Persona.prototype**
- Finalmente en **Object.prototype** (que es lo que correspondería a **Persona.prototype.__proto__**)

Es posible usar esto.... O simplemente usar **extends** y **super** a partir de ES6.

