

Funciones y Objetos

Objetos

- Son entidades mutables que almacenan colecciones de propiedades. Una propiedad es un par "**clave: valor**" donde clave es una cadena y el valor puede ser cualquier dato primitivo, array, objeto o función.
- La forma más sencilla de crear un objeto es con la notación literal: "{...}" (es equivalente a utilizar **new Object()** y luego asignar las propiedades):

```
// Notación literal:
let persona1 = {
  nombre: "Ana",
  edad: 20
};

// Equivalente a:
let persona2 = new Object();
persona2.nombre = "Paco";
persona2.edad = 29;
```

```
✓ persona1: Object {nombre: "Ana", edad: 20}
  edad: 20
  nombre: "Ana"
  > __proto__: Object {constructor: , __defineG...

✓ persona2: Object {nombre: "Paco", edad: 29}
  edad: 29
  nombre: "Paco"
  > __proto__: Object {constructor: , __defineG...
```

Objetos. Propiedades.

```
let empleado = {  
  nombre: "Ana",  
  edad: 20,  
  "Trabajador temporal": true,  
  domicilio: {  
    calle: "Mayor, 2",  
    poblacion: "Valladolid",  
  },  
};  
empleado.sexo = "Masculino";  
console.log(empleado.sexo); // Masculino  
console.log(empleado.nombre); // Ana  
empleado["Trabajador temporal"] = false;  
console.log(empleado["Trabajador temporal"]); // false  
console.log(empleado.domicilio.calle); // Mayor, 2  
delete empleado.edad;  
console.log(empleado.edad); // undefined
```

- El nombre de las propiedades puede contener espacios, en cuyo caso hay que delimitarlas por “ ”.
- Las propiedades (nombre:valor) finalizan con una coma. Incluso la última propiedad puede finalizar con coma.
- El acceso a la propiedad se puede hacer con dos notaciones:
 - **obj.pro** -> Más sencilla
 - **obj[“pro”]** -> Válida para nombres de propiedad con espacios.
- Se puede borrar una propiedad con el operador “**delete**”. (ej: **delete o.nombre**)
- Si una propiedad no existe o no tiene valor se devuelve “undefined”.
- Todas las propiedades son públicas.

Objetos. Propiedades calculadas.

- Son propiedades cuyo nombre se obtiene a partir del contenido de variables o expresiones. Ejemplos:

```
const propiedad1 = 'edad';  
const valor = 20;  
const propiedad2 = 'trabajador';  
const persona = { nombre: 'Pepe', [propiedad1]: valor, [propiedad2]: true };
```

```
{nombre: 'Pepe', edad: 20, trabajador: true}
```

```
const propiedades = ['nombre', 'edad', 'trabajador'];  
const valores = ['Luis', 43, true];  
const persona = {};  
for (const key in propiedades) {  
  | persona[propiedades[key]] = valores[key];  
}
```

```
{nombre: 'Luis', edad: 43, trabajador: true}
```

Objetos. Iterar y copiar.

```
const cliente = { nombre: "Pepe", saldo: 1000, credito: true };
for (const key in cliente)
  console.log(`Propiedad: ${key} Valor: ${cliente[key]}`);
console.log("nombre" in cliente);
console.log("apellido" in cliente);
const copia1 = { ...cliente };
console.log(copia1);
const copia2 = { apellido: "García", ...cliente };
console.log(copia2);
let copia3 = Object.assign(cliente);
console.log(copia3);
let copia4 = {};
Object.assign(copia4, cliente);
console.log(copia4);
```

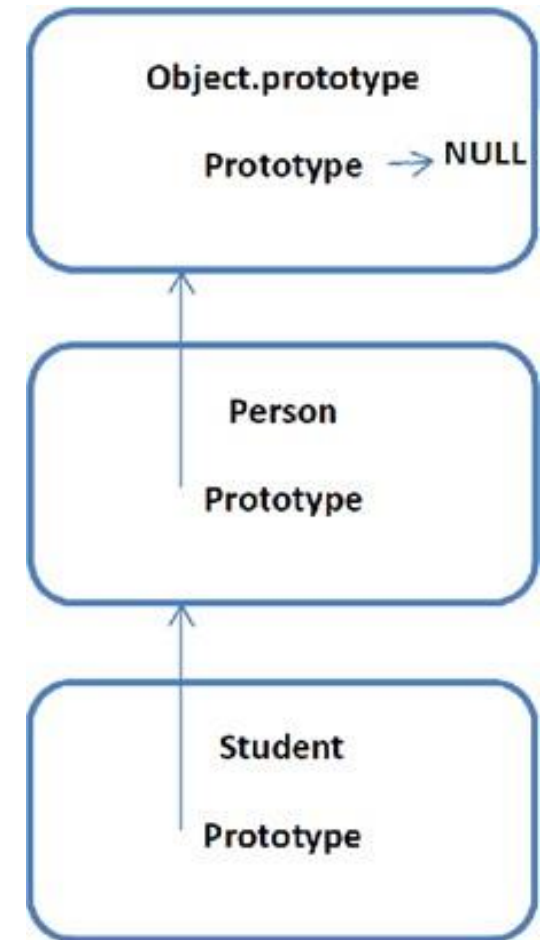
```
Propiedad: nombre Valor: Pepe
Propiedad: saldo Valor: 1000
Propiedad: credito Valor: true
true
false
{ nombre: 'Pepe', saldo: 1000, credito: true }
{ apellido: 'García', nombre: 'Pepe', saldo: 1000, credito: true }
{ nombre: 'Pepe', saldo: 1000, credito: true }
{ nombre: 'Pepe', saldo: 1000, credito: true }
```

- **for..in** permite enumerar las propiedades de un objeto.
- **for..of** solo se admite en objetos de tipo iterador (no todo es un iterador)
 - Un iterador es aquel que implementa "next()", "done" y "value".
 - Para iterar con for..of también se pueden utilizar los métodos:
 - **Object.entries(obj1)**
 - **Object.keys(obj1)**
 - **Object.values(obj1)**
- **in** permite comprobar si un objeto tiene una propiedad.
- Dos formas fáciles de clonar objetos son (hay otras más que veremos):
 - El operador spread.
 - Utilizando **Object.assign()** -> compone objetos a partir de uno o varios objetos.

Objetos. Prototipos.

- En JavaScript todo objeto tiene una referencia a su objeto raíz o prototipo.
- Por defecto Object es el objeto raíz de todos los objetos y su prototipo es null.
- Los objetos delegados heredan las propiedades de su prototipo (ej: student es un objeto delegado de person).
- Cuando se accede a una propiedad de un objeto se buscará dicha propiedad en toda la jerarquía (de abajo a arriba) hasta llegar al Object y si no se encuentra se devuelve el valor “undefined”.
- Para acceder al prototipo de un objeto es posible utilizar:
 - La propiedad **__proto__** del objeto (no es estándar, se usa más en código legacy (obsoleto))
 - El método **Object.getPrototypeOf(obj)** : devuelve el objeto prototipo de “obj”.
 - El método **Object.setPrototypeOf(target,proto)**: asigna a “target” el prototipo “proto”.

Se recomienda utilizar **Object.getPrototypeOf(obj)** y **Object.setPrototypeOf(target,proto)**

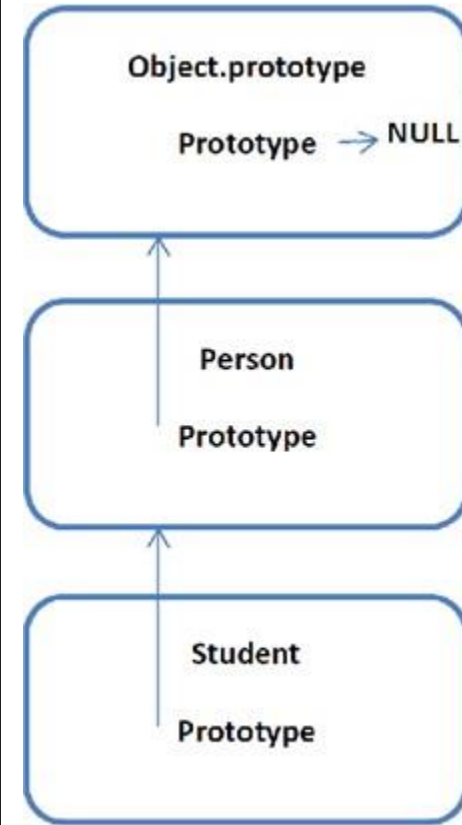


Objetos. Prototipos.

- Ejemplo:

```
const person = { name: 'Paco', age: 30 };
const student = { course: 'A' };
// El prototipo de los dos objetos es el mismo:
console.log(Object.getPrototypeOf(person) === Object.getPrototypeOf(student)); // true
// "student" no tiene la propiedad "name"
console.log(student.name); // undefined
// Asigna a "student" el prototipo "Person"
Object.setPrototypeOf(student, person);
// Ahora "student" tiene la propiedad "name" (en el prototipo)
console.log(student.name); // Paco

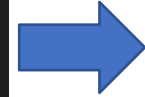
// Creamos una nueva propiedad "name" en "student"
student.name = 'Pepe';
console.log(student.name); // Pepe <-- la propiedad "name" sombrea la del prototipo
console.log(person.name); // Paco <-- "name" del prototipo permanece invariable
```



- Las propiedades se buscan en el objeto actual y en la jerarquía (de abajo a arriba).
- Las propiedades de los objetos son públicas (no hay propiedades privadas).
- Se pueden “sombrear” propiedades del prototipo utilizando nombres coincidentes.

Objetos. Iterar por las propiedades.

```
const person = { name: 'Paco', age: 30 };
const student = { course: 'A' };
// "student" es el delegado de "person":
Object.setPrototypeOf(student, person);
// Itera por las propiedades (incluye los prototipos):
console.log('>>>>> for in:');
for (const k in student) {
  console.log(`${k}: ${student[k]}`);
}
// Itera por las propiedades (solo las propias del objeto):
console.log('>>>>> for of:');
for (const k of Object.keys(student)) {
  console.log(`${k}: ${student[k]}`);
}
// Itera por las propiedades (solo las propias del objeto):
console.log('>>>>> forEach');
Object.entries(student).forEach(e => {
  console.log(`${e[0]}: ${e[1]}`);
});
```



```
>>>>> for in:
course: A
name: Paco
age: 30
>>>>> for of:
course: A
>>>>> forEach
course: A
```

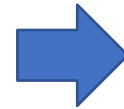
- Para determinar si una propiedad es propia o heredada se puede utilizar el método “hasOwnProperty”. Ejemplo:

```
for (const key in student) {
  if (Object.hasOwnProperty.call(student, key)) {
    console.log(student[key]);
  }
}
```


Objetos. Más formas de crear objetos.

- Además de los objetos literales y los creados mediante el operador spread, se pueden crear objetos con **Object.create()** pasando como parámetro el prototipo deseado.
- **Object.create()** especialmente útil cuando se desean crear objetos delegados. Ejemplo:

```
const person = { name: 'Pepe', age: 23 };  
const student = Object.create(person);  
student.course = 'A';  
const customer = Object.create(person);  
customer.sales = 1230;  
console.log(person);  
console.log(student);  
console.log(customer);
```



```
{name: 'Pepe', age: 23}  
{course: 'A'}  
{sales: 1230}
```

- **Object.assign()** permite crear objetos y añadir propiedades de otros objetos. Ejemplo:

```
const person = { name: 'Pepe', age: 23 };  
const data = { height: 160 };  
const student = Object.assign({}, person, data);  
console.log(person);  
console.log(student);
```




```
{name: 'Pepe', age: 23}  
{name: 'Pepe', age: 23, height: 160}
```

Objetos y métodos

- Las funciones dentro de los objetos se denominan “métodos”.
- Existen dos formas sencillas de añadir un método a un objeto:

```
// Opción 1:  
let u1 = {  
  nombre: "Pepe",  
  ver: function() {  
    console.log(this.nombre);  
  },  
};  
u1.ver();
```

ver: function() {...}
se puede escribir
también como
ver() {...}



```
// Opción 2:  
let u2 = Object.create(null);  
u2.nombre = "Paco";  
u2.ver = function() {  
  console.log(this.nombre);  
};  
u2.ver();
```

- Dentro de los métodos “this” permite acceder al contexto de ejecución actual (en estos dos casos el contexto es el propio objeto).
- Este enfoque tiene un problema: tendríamos que añadir el método a todos los objetos y esto es repetitivo e ineficiente >>>

Objetos y métodos

- Es posible crear un objeto que podemos llamar “padre” con la definición de un método y luego crear los objetos delegados “hijos”:

```
const persona = {  
  visualizar: function () {  
    console.log(this);  
  }  
};  
  
const estudiante = Object.create(persona);  
estudiante.nombre = 'Pedro';  
estudiante.visualizar(); // {nombre: 'Pedro'}
```

← También se puede omitir la palabra function en la definición del método.

- Este enfoque se basa en la composición: “persona” es un objeto “padre” que delega en diferentes objetos hijos (ej: “estudiante”) utilizando Object.create().
- El método “visualizar()” está disponible para todos los objetos delegados.

Objetos. Función constructora.

- En javascript la palabra “function” se utiliza para crear una función o un método pero también se puede utilizar para crear objetos:

```
function Persona (nombre, edad) {  
  this.nombre = nombre;  
  this.edad = edad;  
}  
  
const persona = new Persona('Pepe', 23);  
console.log(persona); // {nombre: 'Pepe', edad: 23}
```

- En este caso el nombre de la función debe comenzar por mayúscula.
- Para crear un nuevo objeto invocaremos a la función constructora precedida de la palabra reservada “**new**”. Invocar directamente la función no tiene sentido.
- Con “**this.<propiedad>=<valor>**” se crean las propiedades del objeto.
- El resultado final es el mismo que si hubiésemos creado un objeto literal.

Objetos. Función constructora.

- ¿Cómo se pueden añadir métodos cuando se usa la función constructora?

```
function Persona (nombre, edad) {  
  this.nombre = nombre;  
  this.edad = edad;  
  this.visualizar = function () {  
    console.log(`${nombre} ${edad}`);  
  };  
}  
  
const persona = new Persona('Pepe', 23);  
persona.visualizar();  
console.log(persona);
```

El método “visualizar” se crea dentro de la instancia (objeto “persona”)



```
Pepe  23  
▼ Persona {nombre: 'Pepe', edad: 23, visualizar: f}  
  edad: 23  
  nombre: 'Pepe'  
> visualizar: f () {\r\n    console.log(`${nombre}  
> __proto__: Object
```

- Este enfoque no es eficiente porque tendríamos el método visualizar en cada uno de los objetos contruidos por la función constructora. >>>

Objetos. Función constructora.

- La función constructora proporciona la propiedad “prototype” que permite acceder al prototipo para añadir métodos. Ejemplo:

```
function Persona (nombre, edad) {  
  this.nombre = nombre;  
  this.edad = edad;  
}  
Persona.prototype.visualizar = function () {  
  console.log(this.nombre);  
};  
const persona = new Persona('Pepe', 23);  
persona.visualizar();  
console.log(persona);
```



La propiedad “prototype” solo está disponible en funciones constructoras.

```
Pepe  
▼ Persona {nombre: 'Pepe', edad: 23}  
  edad: 23  
  nombre: 'Pepe'  
  > __proto__: Object
```

- Utilizando “prototype” también es posible establecer el objeto prototipo de la función constructora y así crear objetos delegados. Ejemplo:

```
const serVivo = { vivo: true };  
Persona.prototype = serVivo;
```



Los objetos creados con el constructor “Persona” serán delegados de “serVivo”.

Ejercicio 2 (ej2.js)

- Como fase inicial de una simulación para un juego de ajedrez se desea crear los siguientes elementos:
 - Un array “tablero” de dos dimensiones: 8x8
 - Un objeto “figura” con las propiedades “x” e “y” (coordenadas de la posición en tablero) y el método “moverA(x,y)” que permite cambiar “x” e “y”.
 - Una array con las 16 figuras disponibles. Cada figura es un objeto delegado de “figura” con la propiedad “tipo” que debe ser uno de los siguientes strings: “K”->Rey, “Q”->Reina, “T”->Torre, “A”->Alfil, “C”->Caballo y “P”->Peón.
 - Una función “colocarPieza(figura, tablero)” que permite colocar cada figura en un lugar aleatorio del tablero.
- Finalmente se debará:
 - Colocar todas las figuras en el tablero.
 - Motrar el tablero con “console.table(tablero)”:

(index)	0	1	2	3	4	5	6	7
0		'P'						
1						'A'		'P'
2	'P'		'T'		'P'			
3	'A'			'P'			'T'	
4					'P'			'C'
5	'P'							'P'
6			'Q'					
7					'C'			'K'

Objetos y clases

- Además de la función constructora a partir de ES6 se pueden crear objetos mediante clases:

```
function Persona (nombre) {  
  this.nombre = nombre;  
}  
Persona.prototype.visualizar = function () {  
  console.log(this);  
};  
const p1 = new Persona('Pedro');  
p1.visualizar(); // Persona {nombre: "Pedro"}
```

```
class Ciudadano {  
  constructor(nombre) {  
    this.nombre = nombre;  
  }  
  visualizar() {  
    console.log(this);  
  }  
}  
let c1 = new Ciudadano("Pedro");  
c1.visualizar(); // Ciudadano {nombre: "Pedro"}
```

- El ejemplo de la izquierda es el utilizado ES5 y anteriores.
- El ejemplo de la derecha es el utilizado ES6. En este caso “class” es “azúcar sintáctico”. Internamente la herencia se basa en la función constructora y en prototipos (como el caso de la izquierda).

Herencia en clases

```
class Persona {
  constructor (nombre) {
    this.nombre = nombre;
  }

  saludar () {
    return `Hola soy ${this.nombre}`;
  }
}

class Programador extends Persona {
  constructor (nombre, lenguaje) {
    super(nombre);
    this.lenguaje = lenguaje;
  }

  saludar () {
    return super.saludar() + ` y programo en ${this.lenguaje}`;
  }
}

const pe = new Persona('Pepe');
console.log(pe.saludar());
const pr = new Programador('Paco', 'javascript');
console.log(pr.saludar());
```

- Se permite herencia mediante la palabra “extends”.
- Se permite acceder la clase padre con “super”.
- El método “saludar” de “Programador” sombrea a “saludar” de “Persona”.
- Internamente todo se define mediante prototipos y objetos delegados.



Hola soy Pepe

Hola soy Paco y programo en javascript

Miembros estáticos

```
class Persona {  
  constructor (nombre) {  
    this.nombre = nombre;  
  }  
  
  static comparar (personaA, personaB) {  
    return personaA.nombre.localeCompare(personaB.nombre);  
  }  
  
  static crearPersona (nombre) {  
    return new this(nombre);  
  }  
}  
  
const a = [new Persona('Pepe'), new Persona('Paco')];  
console.log(Persona.comparar(a[0], a[1])); // 1  
a.sort(Persona.comparar);  
console.log(a.map(e => e.nombre).join(',')); // Paco,Pepe  
console.log(Persona.comparar(a[0], a[1])); // -1  
const persona = Persona.crearPersona('Luis');  
console.log(persona.nombre); // Luis
```

- La palabra “static” permite definir miembros exclusivos de una clase.
- Solo son accesibles mediante la clase (no mediante la instancia).
- No forman parte del prototipo.
- En el ejemplo “crearPersona” es una factoria que utiliza “this” para llamar al propio constructor.



```
1  
Paco,Pepe  
-1  
Luis
```

Getters y Setters

```
class Persona {  
  constructor (nombre) {  
    this._nombre = nombre;  
  }  
  
  set nombre (value) {  
    this._nombre = value;  
  }  
  
  get nombre () {  
    return this._nombre;  
  }  
}  
  
const p = new Persona('Paco');  
console.log(p.nombre); // Paco  
p.nombre = 'Raul';  
console.log(p.nombre); // Raul
```

- Igual que en otros lenguajes podemos declarar getters y setters para encapsular miembros.
- Por convención el miembro privado se declara anteponiendo el símbolo “_”.
- Realmente el campo es accesible pero no debe utilizarse directamente sino por medio de los getters y setters:

```
p._nombre = 'Pedro';  
console.log(p.nombre); // Pedro
```



No es correcto

- Si solo se define el getter será un miembro de solo lectura y si solo se define el setter de solo escritura.

Funciones setInterval y setTimeout

- La función “setTimeout” permite ejecutar una función “callback” después de un cierto tiempo.
 - Formato: **setTimeout(<función>,<ms>)**
 - Donde “función” representa el nombre de la función y “ms” los milisegundos.
- La función “setInterval” permite ejecutar una función “callback” cada cierto tiempo.
 - Formato: **setInterval(<función>,<ms>)**
 - Donde: “función” representa el nombre de la función y “ms” los milisegundos.
 - setInterval devuelve un numero que se puede utilizar posteriormente para detener el bucle mediante **clearInterval(<numero>)**.

```
setTimeout(() => console.log('hola'), 1000);
setInterval(function () {
  console.log(Math.random() * 10);
}, 3000);
```

```
let contador = 0;
const id = setInterval(function () {
  console.log(Math.random() * 10);
  if (contador++ === 5) {
    clearInterval(id);
  }
}, 3000);
```

Ejercicio 3 (ej3.html y ej3.js)

- Se desean crear las siguientes clases con los getters y setters apropiados:
 - **Vehiculo**. Atributos: pasajeros.
 - **Turismo** (derivada de Vehiculo). Atributos: color.
 - **Camion** (derivada de Vehiculo). Atributos: tara.
- Para crear la simulación de los vehículos que circulan por una carretera se desean las siguientes funciones:
 - **capturaReloj()**: devuelve la hora actual (hh:mm:ss).
 - **generarVehiculos()**: genera objetos aleatoriamente que se almacenaran en un array global:
 - Entre uno y cuatro Turismos con un nº de pasajeros aleatorio (entre 1 y 7) y un color aleatorio (entre azul, rojo y verde).
 - Entre uno y cuatro Camiones con un nº de pasajeros aleatorio (entre 1 y 7) y una tara aleatoria (entre 0 y 9999).
 - **mostrarVehiculos()**: genera el código html para visualizar los vehículos en una página web (mediante document.write()).

Hora: 12:20:37	Tipo: Turismo	Pasajeros: 1	Color: red
Hora: 12:20:37	Tipo: Turismo	Pasajeros: 6	Color: blue
Hora: 12:20:37	Tipo: Camion	Pasajeros: 4	Tara: 6744
Hora: 12:20:37	Tipo: Camion	Pasajeros: 3	Tara: 7154
Hora: 12:20:39	Tipo: Turismo	Pasajeros: 1	Color: blue
Hora: 12:20:39	Tipo: Camion	Pasajeros: 1	Tara: 2543
Hora: 12:20:39	Tipo: Camion	Pasajeros: 1	Tara: 6142
Hora: 12:20:41	Tipo: Turismo	Pasajeros: 7	Color: green
Hora: 12:20:41	Tipo: Turismo	Pasajeros: 2	Color: red
Hora: 12:20:41	Tipo: Camion	Pasajeros: 3	Tara: 1222
Hora: 12:20:41	Tipo: Camion	Pasajeros: 1	Tara: 506
Hora: 12:20:43	Tipo: Turismo	Pasajeros: 5	Color: blue
Hora: 12:20:43	Tipo: Turismo	Pasajeros: 2	Color: blue
Hora: 12:20:43	Tipo: Turismo	Pasajeros: 7	Color: green
Hora: 12:20:43	Tipo: Camion	Pasajeros: 4	Tara: 8215
Hora: 12:20:43	Tipo: Camion	Pasajeros: 2	Tara: 8064
Hora: 12:20:45	Tipo: Turismo	Pasajeros: 5	Color: blue
Hora: 12:20:45	Tipo: Turismo	Pasajeros: 2	Color: red
Hora: 12:20:45	Tipo: Camion	Pasajeros: 3	Tara: 4167



Formato de visualización

Las tres funciones se invocarán un total de 10 veces con un intervalo de 2 segundos.