Unidad 5
Programación Orientada a Objetos

CESUR
Tu Centro Oficial de FP

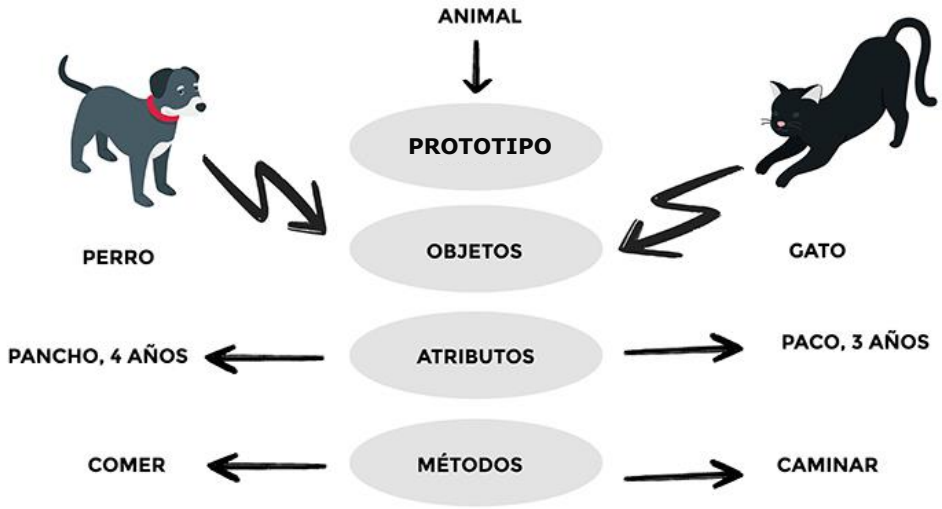
1. Introducción

En Javascript todo es un objeto: las fechas, las expresiones regulares, los arrays, las funciones, etc.

OBJETO – entidad de un programa que tiene una serie de propiedades (rasgos, colores, marcas, medidas….) y también un conjunto de comportamientos (encender, caminar, frenar, abrir, ….)

PROGRAMACIÓN ORIENTADA A OBJETOS – modelo de programación que pretende acercar todo lo posible el mundo real a un sistema informático. Lo que pretende la POO es unificar en un mismo elemento, llamado **OBJETO**, tanto los datos como las funciones que lo manipulan, con esto se logra un mayor grado de modularidad.

Cada entidad de un problema se puede representar como un **OBJETO**, definiendo sus propiedades (**ATRIBUTOS**) y sus acciones o funciones (**MÉTODOS**). El **DESARROLLO DE UN PROGRAMA** consistirá en establecer la relación de comunicación que se crea entre los distintos objetos que intervienen en el problema.

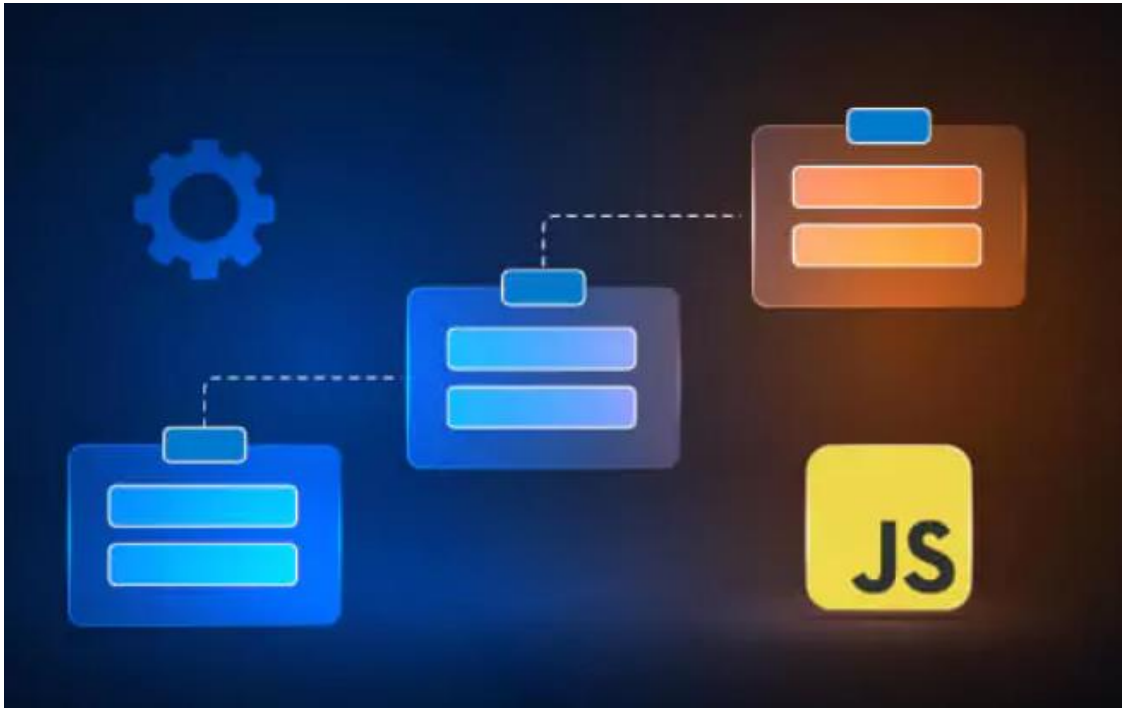


1. Introducción

A diferencia de otros lenguajes orientados a objetos, Javascript no está basado en clases, sino basado en **PROTOTIPOS**.

PROTOTIPO – objeto que se utiliza como una plantilla a partir de la cual se obtiene un conjunto inicial de propiedades de un nuevo objeto.

Todas las aplicaciones de JS usa objetos y basa su lógica en las interacciones entre ellos.



2. JSON

JSON (JavaScript Object Notation) – sintaxis que se emplea para almacenar e intercambiar datos entre aplicaciones. Es una alternativa más sencilla que XML. Se utiliza con mucha frecuencia para el intercambio de datos en aplicaciones web.

CARACTERÍSTICAS DE JSON

- El formato JSON es ampliamente aceptado en las aplicaciones que incluyen arquitectura cliente-servidor para intercambio de datos entre el cliente y el servidor.
- JSON es un formato fácil de usar (tanto de leerlo y escribirlo) y puramente basado en texto, liviano y legible por humanos.
- Aunque la abreviatura comienza con JavaScript, JSON no es un lenguaje de programación, solo es una especificación para la notación de datos.
- Es independiente de la plataforma (Windows, Linux, Mac etc.) y el idioma de programación, está integrado en todos los lenguajes populares como pueden ser JavaScript, PHP, C#, Java, C++, Python, Ruby etc.
- La extensión del nombre del archivo JSON es .json
- Por defecto utiliza codificación UTF-8.

2. JSON

SINTAXIS JSON

JSON se basa en dos estructuras:

1.- OBJETO: Es una colección de pares de nombre/valor - Van encerradas entre llaves: {}

2.- ARRAY: Es una lista ordenada de valores - Van encerrados entre corchetes: []

1.- OBJETO

Un objeto es un conjunto desordenado de pares nombre/valor.

Un objeto comienza con llave izquierda { y termina con una llave a la derecha } , puede haber saltos de línea entre los nombres/valor para una mejor lectura.

Cada nombre debe ir entre comillas dobles y va seguido de : dos puntos y los pares de nombre/valor están separados por , (comas)

```
{
  "nombre": "Cristian",
  "apellido": "Gonzalez",
  "edad": 25,
  "dni": 12345678,
  "direccion": "Av. Siempre Viva 123",
  "telefono": 12345678,
  "email": "cristiang@gmail.com"
}
```

2. JSON

Los valores posibles de los objetos JSON son:

- **Strings:** Las cadenas deben estar encerradas entre comillas dobles y pueden contener caracteres escapados como \n \t etc.
- **Numbers:** Números enteros y flotantes incluyendo la notación exponencial. No podemos indicar octal o hexadecimal.
- **Booleans:** Pueden almacenar solo los valores true o false.
- **nulls:** Podemos almacenar el valor null, indicando que está vacío.
- **Arrays:** Un array es una colección ordenada de valores. Los valores deben estar encerrados entre corchetes.
- **Objetos:** Es decir podemos definir objetos anidados.

NOTA: Observaciones sobre JSON

- Los string deben ir obligatoriamente entre comillas dobles.
- Los datos que no requieren comillas deben ir en minúscula.
- No podemos dejar vacío ni la clave ni el valor
- JSON no incluye una sintaxis para los comentarios
- Es incorrecto poner una coma después del último nombre/valor

```
{
  "nombre": "Cristian",
  "edad": 25,
  "altura": 1.75,
  "casado": true,
  "trabajo": null,
  "hijos": ["Juan", "Pedro", "José"],
  "estudios": {
    "primaria": true,
    "secundaria": false,
    "universidad": false
  }
}
```

2.- ARRAY

Es la segunda estructura válida de un archivo JSON. Van encerrados entre corchetes [].

No es necesario que cada objeto contenido en el array tenga una estructura similar.

```
[
  3,
  {
    "nombre": "Juan",
    "edad": 30,
    "pais": "España"
  },
  {
    "nombre": "Pedro",
    "edad": 35,
    "pais": "España"
  },
  {
    "nombre": "Ana",
    "edad": 25,
    "pais": "Argentina"
  },
  true
]
```

En este ejemplo, el primer elemento es un número (3), los tres siguientes son objetos (formato nombre/valor) y el último es un valor true.

2. JSON

La sintaxis JSON en JS tiene en cuenta 5 reglas:

1. Es un subconjunto de la sintaxis de JavaScript
2. Los datos aparecen como pares nombre/valor
3. Los datos se separan por comas
4. Las llaves { } contienen objetos
5. Los corchetes [ ] contiene arrays

```
{
        "empleados":[
            {"nombre": "Gerardo", "apellidos": "García"},
            {"nombre": "Maryna", "apellidos": "Hernández"},
            {"nombre": "Abel", "apellidos": "Pacheco"}
        ]
}
```

Este ejemplo, define el **OBJETO** empleados con un array que contiene tres registros del tipo empleado.

En JSON hay que tener en cuenta:

- Sólo es posible definir propiedades, no métodos.
- Las propiedades deben ir entre comillas dobles.

2. JSON

VENTAJAS DE JSON

- **Formato ligero de intercambio de datos** – se prescinde al máximo de elementos que sobrecarguen los datos de información innecesaria
- **Independiente del lenguaje** – JSON utiliza la sintaxis JS, pero su formato es sólo texto, al igual que el XML. El texto puede leerse y utilizarse como formato de datos por cualquier lenguaje de programación.
- **Autodescriptivo y fácil de entender**

JSON
https://developer.mozilla.org/es/docs/Learn/JavaScript/Objects/JSON

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Actividad 1</title>
        <script>
            document.write("EJEMPLO DE OBJETOS JSON<br><br>")
            var text={
                "nombre":"IES Mar de Alborán",
                "calle": "Fuente María Gil S/N",
                "teléfono":"951 33 56 45"
            };
            document.write ("NOMBRE: "+ text.nombre + "<br>DIRECCIÓN: " + text.calle + "<br>TELÉFONO: " + text.teléfono);
        </script>
    </head>
    <body>
    </body>
</html>
```

EJEMPLO DE OBJETOS JSON

NOMBRE: IES Mar de Alborán
DIRECCIÓN: Fuente María Gil S/N
TELÉFONO: 951 33 56 45

Sintaxis de JSON
https://www.youtube.com/watch?v=RhxOTqFbI5Q

2. JSON

Las diferencias entre JSON y XML son las siguientes:

- XML tiene que ser analizado con un analizador XML, mientras que JSON puede ser analizado por una función JavaScript estándar.
- JSON no utiliza etiqueta de fin.
- JSON es más corto.
- JSON es más rápido de leer y escribir.
- JSON puede utilizar matrices.
- JSON es más rápido y sencillo que XML.

Las similitudes entre ambos son:

- JSON y XML son autodescriptivos (legible por humanos).
- JSON y XML son jerárquicos (valores dentro de valores).
- JSON y XML pueden ser analizados y utilizados por muchos lenguajes de programación.
- Tanto JSON como XML se pueden obtener con una llamada XMLHttpRequest.

XML vs. JSON

```xml
<?xml version="1.0" encoding="UTF-8"?>
<endereco>
    <cep>31270901</cep>
    <city>Belo Horizonte</city>
    <neighborhood>Pampulha</neighborhood>
    <service>correios</service>
    <state>MG</state>
    <street>Av. Presidente Antônio Carlos, 6627</street>
</endereco>
```

```json
{
    "endereco": {
        "cep": "31270901",
        "city": "Belo Horizonte",
        "neighborhood": "Pampulha",
        "service": "correios",
        "state": "MG",
        "street": "Av. Presidente Antônio Carlos, 6627"
    }
}
```

2. JSON

En una aplicación web muchas veces recibimos los datos en formato de texto pero con una estructura que respeta el formato json, nosotros debemos convertir dicho formato a un objeto de JavaScript. Esta actividad se nos hace muy fácil mediante el método **parse** del objeto JSON.

PARSE

Devuelve el objeto equivalente en JS desde una cadena JSON:

```
const datos=`
{
  "nombre": "Cristian",
  "apellido": "Gonzalez",
  "edad": 25,
  "dni": 12345678,
  "direccion": "Av. Siempre Viva 123",
  "telefono": 12345678,
  "email": "cristiang@gmail.com"
}`

const persona=JSON.parse(datos)

console.log(persona.nombre) // "Cristian"
console.log(persona.apellido) //"Gonzalez"
console.log(persona.edad)  /25
console.log(persona.dni) // 12345678
console.log(persona.direccion) // "Av. Siempre Viva 123"
console.log(persona.telefono) // 12345678
console.log(persona.email) // "cristiang@gmail.com"
```

La variable 'persona' se trata de un objeto creado a partir de los datos del string de la variable 'datos', este string normalmente llega de un servidor web.

Método parse
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse

2. JSON

Podemos también convertir un string que contenga un arreglo:

```
const stringPaises = `[
  {
    "nombre": "Agentina",
    "poblacion": 44
  },
  {
    "nombre": "Brasil",
    "poblacion": 212
  },
  {
    "nombre": "Uruguay",
    "poblacion": 5
  }
]`
const paises=JSON.parse(stringPaises)
console.log(paises[0].nombre) // "Argentina"
console.log(paises[0].poblacion) // 44
console.log(paises[1].nombre) // "Brasil"
console.log(paises[1].poblacion) // 212
console.log(paises[2].nombre) // "Uruguay"
console.log(paises[2].poblacion) // 5
```

Con el formato JSON hay que tener cuidado que las propiedades deben estar encerradas entre comillas dobles, lo mismo que los valores de tipo string, si no tienen las comillas dobles no se puede hacer la transformación.

STRINGIFY

Es el método inverso a parse. Convierte un objeto JS a una cadena en formato JSON.

Este método se le envía un objeto literal de JavaScript y nos retorna un string. Esta acción es muy común cuando hay que enviar un objeto de JavaScript a través de la red de Internet.

```
const usu = {
  usuario: "gustavo",
  clave: "123456"
}
console.log(typeof usu) // object
const datos = JSON.stringify(usu)
console.log(typeof datos) // string
console.log(datos) //'{"usuario":"gustavo","clave":"123456"}'
```

Tenemos un objeto llamado 'usu', luego mediante la llamada al método 'stringify' convertimos dicho objeto en un string.

Método stringify
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify

3. Gestión de objetos

EJEMPLO DE OBJETO:

```
let vector = [4,2,7,9];
console.log(vector.length);
```

Tenemos el **OBJETO** vector.

Con vector.length – accedemos a la **PROPIEDAD** length del objeto vector (longitud del vector).

También utilizamos el **MÉTODO** log() del **OBJETO** console.

Si mostramos el tipo de datos (typeof()) tanto de vector como de console obtenemos que son de tipo **object**:

```
console.log(typeof(vector));
console.log(typeof(console));
```

```
object
object
>
```

De forma genérica, para acceder a las propiedades de un objeto utilizamos el punto (.) y alternativamente la notación de corchetes ([]). Por tanto, son equivalentes:

```
objeto.propiedad
objeto["propiedad"]
```

```
objeto.método(parámetros)
objeto["método"](parámetros)
```

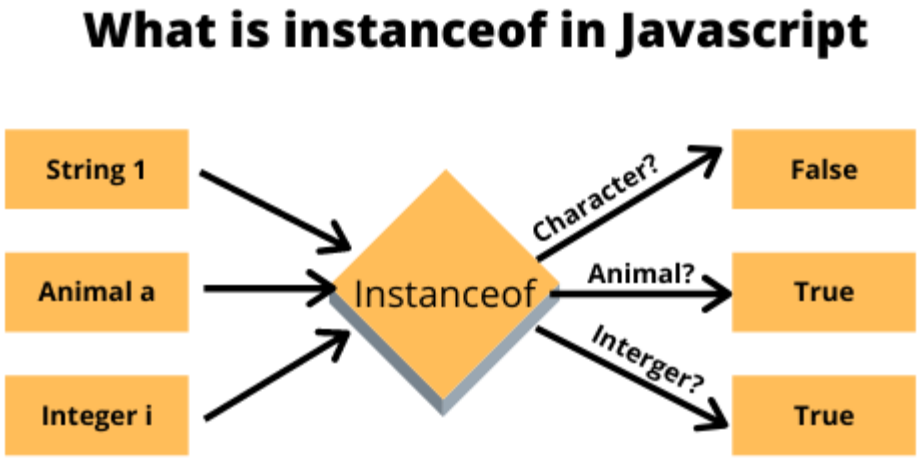
NOTA: en los métodos hay que añadir los paréntesis que permiten pasar parámetros. Recuerda que los métodos son funciones.

3. Gestión de objetos

INSTANCEOF – operador que ayuda a comprobar el tipo de un objeto, es decir, el prototipo del que parte.

Para usarlo, hay que preguntarle si un objeto es de un prototipo concreto y devuelve true o false.

```
let vector = [4,2,7,9];
console.log(vector instanceof Array); // devuelve true
let conjunto = new Set();
console.log(conjunto instanceof Map); // devuelve false
```



3. Gestión de objetos

OBJECT – objeto genérico que representa a los **objetos literales** (**instancias directas**) que son los objetos más sencillos que pueden crear en JS.

No es necesario definirlos previamente, sino que se puede ir creando según se desarrolla el programa.

```
let notas = new Object();
notas.valores = [7,5,3,2,3,9,6];
notas.cantidad = notas.valores.length;
notas.media = notas.valores.reduce((a,b)=>a+b,0)/notas.cantidad;
notas.verMedia = function() {
        console.log(notas.media);
}
notas.verMedia(); // escribe 5
```

Donde:

- **new Object()** – crea un **objeto** vacío que se va completando según avanza la ejecución
- Se definen tres **propiedades**:
  - **valores**
  - **cantidad**
  - **media**
- Se añade un **método**, **verMedia()** que muestra la media de las notas

3. Gestión de objetos

Los objetos también se pueden representar mediante la notación JSON:

```
let viaje={
        origen:"Granada",
        destino:"El Cairo",
        dias:8,
        precio:750,
        mostrar:function(){
                console.log(`${viaje.origen} / ${viaje.destino}`);
                console.log(`durante ${viaje.dias} días: EUR${viaje.precio}`);
        }
};
viaje.mostrar();
```

THIS – objeto que se utiliza para que un método acceda a las propiedades de su propio objeto.

El método anterior mostrar() se podría reescribir de la siguiente forma:

```
mostrar:function(){
        console.log(`${this.origen} / ${this.destino}`);
        console.log(`durante ${this.dias} días: EUR${this.precio}`);
}
```

Objeto this en JS
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/this

Se ha sustituido el objeto de referencia viaje por **this**. En tiempo de ejecución **this** valdrá viaje.

Lo que aporta **this** es seguridad, ya que si se asigna el objeto viaje a otro objeto y se modificaran sus propiedades, esto podría afectar al resultado de nuestro código.

3. Gestión de objetos

Ejemplo 1: OBJETOS

Crea una página que incluya un objeto denominado usuario que permita autenticar a la persona que quiere iniciar la sesión en el sistema

```
LLAMADA CON - usuario:agar007 contraseña: agar007_pass
Sesión iniciada con éxito

LLAMADA CON usuario:agar007 contraseña: agar007pass
Credenciales no válidas
```

```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Actividad 1</title>
        <script>
            const usuario = {
                nombre: 'Antonio García',
                nombreUsuario: 'agar007',
                contraseña: 'agar007_pass',
                login: function(nombreUsuario, contraseña){
                    if (nombreUsuario=== this.nombreUsuario && contraseña === this.contraseña){
                        document.write('Sesión iniciada con éxito');
                    } else {
                        document.write('Credenciales no válidas');
                    }
                }
            }
            document.write('LLAMADA CON - usuario:agar007 contraseña: agar007_pass<br>');
            usuario.login('agar007','agar007_pass');
            document.write('<br><br>LLAMADA CON usuario:agar007 contraseña: agar007pass<br>');
            usuario.login('agar007','agar007pass');
        </script>
    </head>
    <body>
    </body>
</html>
```

4. Constructores

CONSTRUCTOR – Método especial que se utiliza para crear e inicializar de forma personalizada un objeto a partir de una clase.

En JS sólo puede haber un método llamado constructor en cada clase.

El constructor se ejecuta inicializando las propiedades del objeto cuando se utiliza el operador **new**.

Si tenemos una clase heredada de otra, se puede ejecutar el constructor de la clase madre invocando a **super()**.

```
class Viaje{
    constructor(or,des,di,pre){
        this.origen = or;
        this.destino = des;
        this.dias = di;
        this.precio = pre;
    }
    mostrar(){
        console.log(this.origen +'/'+ this.destino);
        console.log('durante ' +this.dias+ ' días: '+this.precio+' EUR');
    }
}
let miViaje = new Viaje ("Barcelona","Ibiza",2,112);
miViaje.mostrar();
```

```
Barcelona/Ibiza
durante 2 días: 112 EUR
```

Ejemplo 2: CONSTRUCTORES I

Crea un programa que incluya una clase para modelar un objeto “teléfono móvil” que tenga al menos estas propiedades:

1) CPU
2) RAM
3) Almacenamiento
4) Ancho
5) Alto
6) numCamaras

Añade también un método llamado **toString()** que muestre en pantalla la información del objeto creado. Crea cuatro objetos con distintos números de parámetros en la creación y muestra en pantalla la información de cada objeto.

```
CPU: Snapdrago 888, RAM: 8 GB, Almacenamiento: 256 GB, Ancho: 6.5, Alto: 12.4, Número de cámaras: 4

CPU: Exynos 2100, RAM: 6 GB, Almacenamiento: 128 GB, Ancho: 6.1, Alto: 11.8, Número de cámaras: 3

CPU: A14 Bionic, RAM: 4 GB, Almacenamiento: 64 GB, Ancho: 5.8, Alto: 11.7, Número de cámaras: 2

CPU: Kirin 9000, RAM: 12 GB, Almacenamiento: 512 GB, Ancho: 6.7, Alto: 13.2, Número de cámaras: 5
```

4. Constructores

Ejemplo 2: CONSTRUCTORES II

CPU: Snapdrago 888, RAM: 8 GB, Almacenamiento: 256 GB, Ancho: 6.5, Alto: 12.4, Número de cámaras: 4

CPU: Exynos 2100, RAM: 6 GB, Almacenamiento: 128 GB, Ancho: 6.1, Alto: 11.8, Número de cámaras: 3

CPU: A14 Bionic, RAM: 4 GB, Almacenamiento: 64 GB, Ancho: 5.8, Alto: 11.7, Número de cámaras: 2

CPU: Kirin 9000, RAM: 12 GB, Almacenamiento: 512 GB, Ancho: 6.7, Alto: 13.2, Número de cámaras: 5

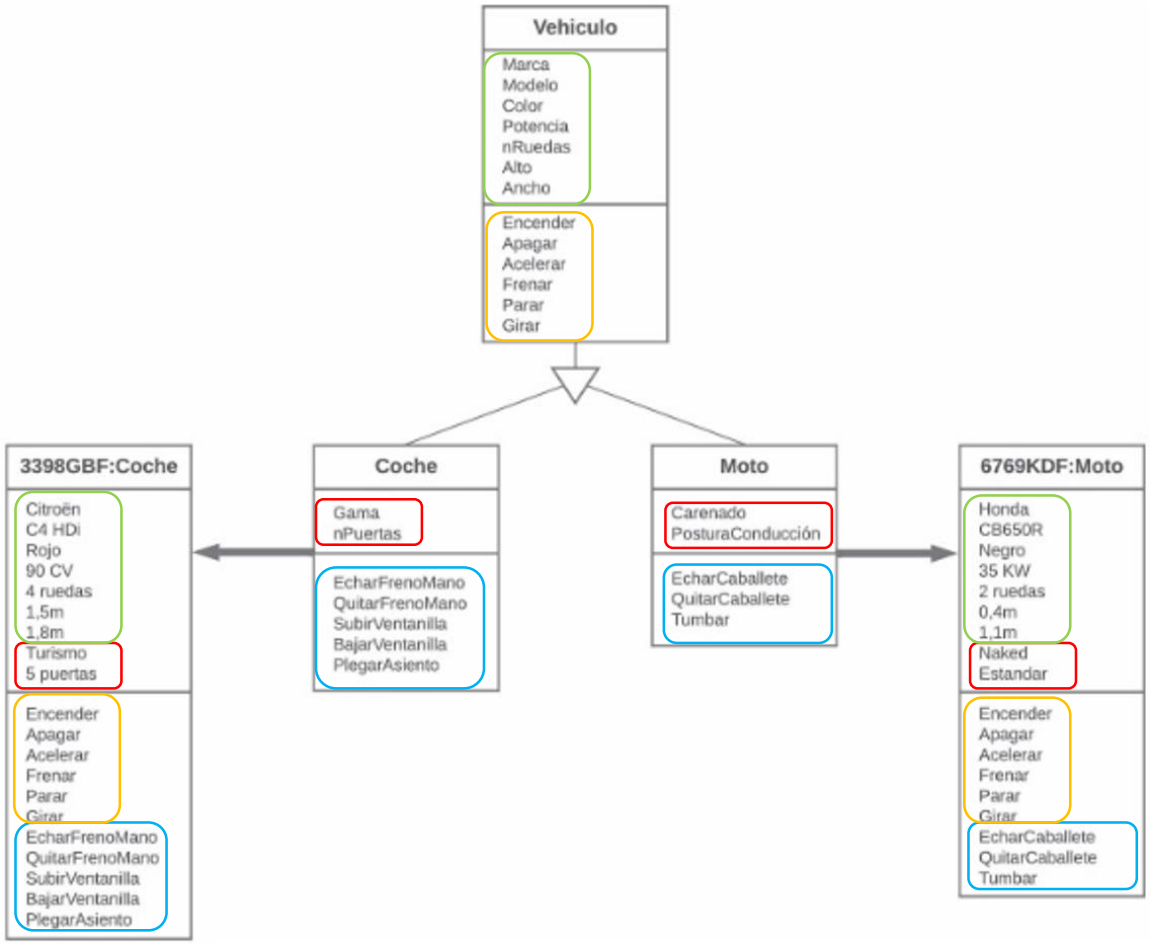
```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Actividad 1</title>
        <script>
            class TelefonoMovil{
                constructor(CPU, RAM, almacenamiento, ancho, alto, numCamaras){
                    this.CPU = CPU;
                    this.RAM = RAM;
                    this.almacenamiento = almacenamiento;
                    this.ancho = ancho;
                    this.alto = alto;
                    this.numCamaras = numCamaras;
                }
                toString(){
                    return 'CPU: ' + this.CPU + ', RAM: ' + this.RAM + ', Almacenamiento: '+ this.almacenamiento + ', Ancho:
                    ' + this.ancho + ', Alto: ' + this.alto + ', Número de cámaras: ' + this.numCamaras + '<br><br>';
                }
            }
            let movil1 = new TelefonoMovil("Snapdrago 888", "8 GB", "256 GB", 6.5, 12.4, 4);
            let movil2 = new TelefonoMovil("Exynos 2100", "6 GB", "128 GB", 6.1, 11.8, 3);
            let movil3 = new TelefonoMovil("A14 Bionic", "4 GB", "64 GB", 5.8, 11.7, 2);
            let movil4 = new TelefonoMovil("Kirin 9000", "12 GB", "512 GB", 6.7, 13.2, 5);

            document.write(movil1.toString());
            document.write(movil2.toString());
            document.write(movil3.toString());
            document.write(movil4.toString());
        </script>
    </head>
    <body>
    </body>
</html>
```

5. Herencia y Polimorfismo

HERENCIA – es el mecanismo por el que una clase (un modelo de objetos), permite heredar características (propiedades y métodos) de otra clase. Esto facilita que se puedan definir nuevas clases basadas en otras existentes, generando una gerarquía de clases dentro de la aplicación.

Si una clase hereda de otra obtiene sus propiedades y métodos pudiendo añadir otros nuevos específicos de esa nueva clase.



Ejemplo: clase Vehículo, sólo contiene propiedades y métodos comunes a una generalidad de vehículos. Hay dos nuevas clases, Coche y Moto que heredan de vehículo, cada una de ellas con sus propias propiedades y métodos únicos. Cuando se crean objetos de clases heredades contienen propiedades y métodos de ambas clases.

5. Herencia y Polimorfismo

POLIMORFRISMO – característica que permite que distintos tipos de objetos (objetos que heredan de clases distintas) tengan métodos con el mismo nombre, de manera que pueden realizar una misma acción de forma diferente.

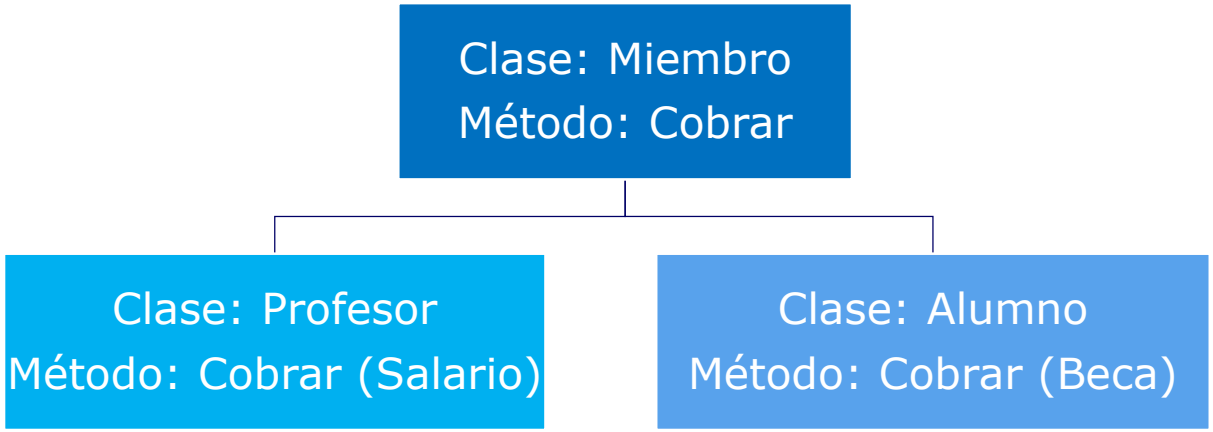
Ejemplo: Tenemos una academia de formación en la que tenemos una clase origen llamada **Miembro** que incluye un único método llamado **Cobrar**.

Por otra parte, tenemos dos clases derivadas llamadas: **Profesor** y **Alumno**.

Tanto profesor como alumno pueden ejecutar el método **Cobrar** y obtener resultados diferentes:

a) La clase **Profesor** al llamar al método **Cobrar** obtiene el salario del Profesor.

b) La clase **Alumno** al llamar al método **Cobrar** obtiene la beca del Alumno.



CESUR
Tu Centro Oficial de FP

Ejemplo 3: HERENCIA Y POLIMORFISMO

Crea un programa que incluya una clase Miembro con las propiedades nombre, alta y estado.

Además, debe incluir un método Cobrar.

Por otra parte, hay que crear dos clases heredadas Profesor y Alumno.

Profesor incluye la propiedad número de alumnos y Alumno la propiedad número de asignaturas.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Actividad 1</title>
        <script>
            class Miembro{
                nombre = " nombre apellido1 apellido2";
                alta = "01/01/2022";
                estado = "vigente";
                constructor(nombre,alta,estado){
                    this.nombre = nombre;
                    this.alta = alta;
                    this.estado = estado;
                }
                cobrar(){document.write('El Miembro '+this.nombre+' ha cobrado <br>');}
            }
            class Profesor extends Miembro{
                nAlumnos = 0;
                constructor(nombre,alta,estado,nAlumnos){
                    super(nombre,alta,estado);
                    this.nAlumnos = nAlumnos;
                }
                cobrar(){document.write('El Profesor '+this.nombre+' ha cobrado <br>');}
            }
            class Alumno extends Miembro{
                nAsignaturas = 0;
                constructor(nombre,alta,estado,nAsignaturas){
                    super(nombre,alta,estado);
                    this.nAsignaturas = nAsignaturas ;
                }
                cobrar(){document.write('El Alumno '+this.nombre+' ha cobrado <br>');}
            }
            let unMiembro = new Miembro("Pepe Ruíz González","12/02/2021","finalizado");
            unMiembro.cobrar();
            let unProfesor = new Profesor("Samuel Orta Pérez","25/06/2021","finalizado",30);
            unProfesor.cobrar();
            let unAlumno = new Alumno("Elena Sánchez Sanz","06/03/2021","finalizado",11);
            unAlumno.cobrar();
        </script>
    </head>
    <body>
    </body>
</html>
```

El Miembro Pepe Ruíz González ha cobrado
El Profesor Samuel Orta Pérez ha cobrado
El Alumno Elena Sánchez Sanz ha cobrado

6. Recorrer un objeto

Para recorre un objeto se recomiendo utilizar el bucle **for..in**, aunque también se puede utilizar un bucle **for..of**.

El bucle va a iterar sobre las propiedades del objeto, pero también lo hará sobre las propiedades que herede en su cadena de prototipos, sacando primero las propiedades de los prototipos más cercanos. Para prevenir este comportamiento y que sólo acceda a las propiedades del objeto en sí, y no de sus prototipos, se puede utilizar el método **getOwnPropertyNames()**.

Hay que tener en cuenta, que la iteración entre propiedades se hace de forma arbitraria, por lo que no se recomienda que se modifiquen las propiedades de un objeto al mismo tiempo que se itera sobre ellas, ya que no hay garantía de que se visite una propiedad que se ha modificado, e incluso se podría visitar una propiedad eliminada dentro del propio recorrido.

getOwnPropertyNames()
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames

6. Recorrer un objeto

Ejemplo 4: RECORRIDOS CON HERENCIA

Utilizando los recorridos muestra las propiedades de los tres objetos creados en el ejemplo 3.

```
PROPIEDADES DE unMiembro:
nombre: Pepe Ruíz González
alta: 12/02/2021
estado: finalizado

PROPIEDADES DE unProfesor:
nombre: Samuel Orta Pérez
alta: 25/06/2021
estado: finalizado
nAlumnos: 30

PROPIEDADES DE unAlumno:
nombre: Elena Sánchez Sanz
alta: 06/03/2021
estado: finalizado
nAsignaturas: 11

PROPIEDADES DE unMiembro COMO ARRAY:
Pepe Ruíz González
12/02/2021
finalizado

PROPIEDADES DE unProfesor COMO ARRAY:
Samuel Orta Pérez
25/06/2021
finalizado
30

PROPIEDADES DE unAlumno COMO ARRAY:
Elena Sánchez Sanz
06/03/2021
finalizado
11
```

```
40
41          //Recorrido for in para mostrar todas las propiedades de cada objeto
42          document.write('<br>PROPIEDADES DE unMiembro:<br>');
43          for (let propiedad in unMiembro){
44              document.write(propiedad+': '+unMiembro[propiedad]+'<br>');
45          }
46          document.write('<br>PROPIEDADES DE unProfesor:<br>');
47          for (let propiedad in unProfesor){
48              document.write(propiedad+': '+unProfesor[propiedad]+'<br>');
49          }
50          document.write('<br>PROPIEDADES DE unAlumno:<br>');
51          for (let propiedad in unAlumno){
52              document.write(propiedad+': '+unAlumno[propiedad]+'<br>');
53          }
54
55          //Recorrido for of para mostrar todas las propiedades de los objetos
56          document.write('<br>PROPIEDADES DE unMiembro COMO ARRAY:<br>');
57          for (let valor of Object.values(unMiembro)){
58              document.write(valor+'<br>');
59          }
60          document.write('<br>PROPIEDADES DE unProfesor COMO ARRAY:<br>');
61          for (let valor of Object.values(unProfesor)){
62              document.write(valor+'<br>');
63          }
64          document.write('<br>PROPIEDADES DE unAlumno COMO ARRAY:<br>');
65          for (let valor of Object.values(unAlumno)){
66              document.write(valor+'<br>');
67          }
```

7. Borrados

La eliminación de propiedades se realiza de forma sencilla. Tan sólo hay que utilizar el operador **delete** sobre una propiedad de un objeto en concreto.

```
Barcelona/Ibiza: durante 2 días, 112 EUR

Recorrido for..in por las propiedades:
origen: Barcelona
destino: Ibiza
dias: 2
precio: 112

Recorrido for..in eliminadas las propiedades precio y días:
origen: Barcelona
destino: Ibiza
```

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Actividad 1</title>
        <script>
            class Viaje{
                constructor(or,des,di,pre){
                    this.origen = or;
                    this.destino = des;
                    this.dias = di;
                    this.precio = pre;
                }
                mostrar(){
                    document.write(this.origen +'/'+ this.destino+': durante ' +this.dias+ ' días, '+this.precio+' EUR<br>');
                }
            }
            let miViaje = new Viaje ("Barcelona","Ibiza",2,112);
            miViaje.mostrar();
            document.write('<br>Recorrido for..in por las propiedades:<br>');
            for (elemento in miViaje){
                document.write(elemento + ": "+miViaje[elemento]+'<br>');
            }
            delete miViaje.precio;
            delete miViaje.dias;
            document.write('<br>Recorrido for..in eliminadas las propiedades precio y días:<br>');
            for (elemento in miViaje){
                document.write(elemento + ": "+miViaje[elemento]+'<br>');
            }
        </script>
    </head>
    <body>
    </body>
</html>
```

8. Prototipos

JS no utiliza el esquema conceptual de clase sino **PROTOTIPOS**.

En JS todos los objetos proceden de un prototipo (conjunto de propiedades y métodos comunes).

Cuando se crean objetos se enlazan con su prototipo.

La ventaja que presenta esta estructura es que los prototipos se pueden modificar al vuelo y los objetos que lo enlazan estarán actualizados en tiempo real.

PROTOTIPOS – mecanismo por el que los objetos de JS heredan características entre sí.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Actividad 1</title>
        <script>
            function Viaje(origen,destino,dias,precio){
                this.origen = origen;
                    this.destino = destino;
                    this.dias = dias;
                    this.precio = precio;
                    this.mostrar = function(){
                        console.log(this.origen / this.destino);
                        console.log(': durante ' +this.dias+ ' dias, '+this.precio+' EUR');
                    }
                }
                let viaje1 = new Viaje ("Barcelona","Ibiza",2,112);
                console.log(viaje1);
        </script>
    </head>
    <body>
    </body>
</html>
```

```
▼ Viaje {origen: 'Barcelona', destino: 'Ibiza', dias: 2, precio: 112, mostrar: f}
    destino: "Ibiza"
    dias: 2
  ▶ mostrar: ƒ ()
    origen: "Barcelona"
    precio: 112
  ▼ [[Prototype]]: Object
    ▶ constructor: ƒ Viaje(origen,destino,dias,precio)
    ▼ [[Prototype]]: Object
      ▶ constructor: ƒ Object()
      ▶ hasOwnProperty: ƒ hasOwnProperty()
      ▶ isPrototypeOf: ƒ isPrototypeOf()
      ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
      ▶ toLocaleString: ƒ toLocaleString()
      ▶ toString: ƒ toString()
      ▶ valueOf: ƒ valueOf()
```

8. Prototipos

En la consola, al desplegar el contenido del prototipo se ven todas las propiedades y los métodos que se pueden utilizar.

Por ejemplo, **valueOf** retorna el valor del objeto sobre el que se llama. Si Viaje reescribirier valueOf():

- El navegador comprobaría si viaje1 tiene un método valueOf()
- Si no, el navegador comprobaría si el objeto prototipo de viaje1 (constructor Viaje()) tiene un método valueOf()
- Si no, el navegador comprobaría si Object() prototipo del objeto prototipo del constructor tiene un método valueOf() disponible.

Recuerda

Las propiedades y los métodos no se copian de un objeto a otro en la cadena del prototipo. Son accedidos subiendo por la cadena.

8. Prototipos

En JS, se puede modificar cualquier prototipo con la propiedad **prototype** actuando sobre el constructor.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Actividad 1</title>
        <script>
            class Viaje{
                constructor(or,des,di,pre){
                    this.origen = or;
                    this.destino = des;
                    this.dias = di;
                    this.precio = pre;
                }
                mostrar(){
                    console.log(this.origen +'/'+ this.destino+': durante ' +this.dias+ ' días, '+this.precio+' EUR');
                }
            }
            let miViaje = new Viaje ("Barcelona","Ibiza",2,112);
            console.log(Viaje.prototype);
            Viaje.prototype.costeDiario = function(){
                return this.precio/this.dias;
            };
            Viaje.prototype.descuento='20%';
            console.log(Viaje.prototype);
        </script>
    </head>
    <body>
    </body>
</html>
```

```
▶ {constructor: ƒ, mostrar: ƒ}
▶ {descuento: '20%', costeDiario: ƒ, constructor: ƒ, mostrar: ƒ}
```

9. Objetos predefinidos

OBJETOS PREDEFINIDOS – son objetos que ya incorpora JS y que se pueden utilizar libremente.

1) STRING – cadena de caracteres. Los métodos más utilizados son:

Método	Utilidad
charAt(*posición*)	Devuelve el carácter que ocupa esa ***posición.***
charCodeAt(*posición*)	Variante que devuelve el código Unicode.
toUpperCase()	Devuelve la cadena en mayúsculas.
toLowerCase()	Devuelve la cadena en minúsculas.
indexOf(*texto*)	Devuelve la posición del texto buscado.
lastIndexOf(*texto*)	Devuelve la posición de la última ocurrencia del texto.
endsWith(*textoABuscar*)	Devuelve **true** si el texto finaliza con lo buscado.
startsWith(*textoABuscar*)	Devuelve true si el texto empieza con lo buscado.
replace(*txtAnt,txtNuevo*)	Reemplaza el texto buscado por un nuevo texto.
trim()	Elimina los espacios en blanco del inicio y del final.
slice(*inicio,fin*)	Extrae la cadena desde la posición ***inicio*** hasta la posición ***fin***, sin incluir ***fin.***
substr(*inicio,n*)	Extrae *n* caracteres desde la posición ***inicio.***
split(*delimitador*)	Rompe un *string* usando un carácter ***delimitador*** y construye un *array* con las piezas generadas.

```
let cadena = " Bolonia ";
console.log("1." + cadena.charAt(1));
console.log("2." + cadena.toUpperCase());
console.log("3." + cadena.toLowerCase());
console.log("4." + cadena.indexOf("n"));
console.log("5." + cadena.lastIndexOf("o"));
console.log("6." + cadena.replace("B","C"));
console.log("7." + cadena.trim());
console.log("8." + cadena.slice(1,3));
console.log("9." + cadena.substr(1,3));
console.log("10." + cadena.split("o"));
```

```
1.B
2. BOLONIA
3. bolonia
4.5
5.4
6. Colonia
7.Bolonia
8.Bo
9.Bol
10. B,l,nia
>
```

9. Objetos predefinidos

Nota técnica

Los **métodos estáticos** son aquellos que deben ser llamados sin instanciar su clase. Por ejemplo, para usar un método estático del prototipo **String()** no puede crearse un ***objeto*** = **new String()** y hacer luego ***objeto*.metodo(),** sino que hay que invocarlo directamente como **String.*metodo*().**

El objeto String también incluye 3 métodos estáticos:

Método	Utilidad
String.fromCharCode(*num1*[,...])	Crea una cadena usando una secuencia de valores Unicode.
String.fromCodePoint(*num1* [,...])	Crea una cadena utilizando la secuencia de puntos de código especificada.
String.raw()	Crea una cadena a partir de una plantilla literal sin formato: no escapa caracteres.

Objeto String
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/String

9. Objetos predefinidos

2) DATE – representa un momento fijo en el tiempo que se puede representar en numerosos formatos.

NOTA: una fecha en JS se especifica como el número de milisegundos que han transcurrido desde el 1/1/1970, UTC.

El objeto Date puede crearse sin parámetros, pero también indicando su lista completa (año, mes, día, hora, minutos, segundos, milisegundos) o un número parcial de ellos.

```
let fechaSinParametros = new Date();
let fechaTodosParametros = new Date(2022,8,17,13,59,49,0);
let fechaTresParametros = new Date(2022,8,17);
let fechaUnParametro = new Date(1000);
```

```
Sun Jul 17 2022 19:30:37 GMT+0200 (hora de verano de Europa central)
Sat Sep 17 2022 13:59:49 GMT+0200 (hora de verano de Europa central)
Sat Sep 17 2022 00:00:00 GMT+0200 (hora de verano de Europa central)
Thu Jan 01 1970 01:00:01 GMT+0100 (hora estándar de Europa central)
>
```

9. Objetos predefinidos

Los métodos más utilizados son:

Método	Utilidad
getDate()	Devuelve el día del mes de la fecha (de 1 a 31).
getDay()	Obtiene el día de la semana de la fecha (de 0 a 6).
getFullYear()	Obtiene el año (con cuatro dígitos).
getHours()	Obtiene la hora de la fecha (número de 0 a 23).
getMilliseconds()	Obtiene los milisegundos en la fecha actual.
getMinutes()	Obtiene los minutos de la fecha.
getMonth()	Obtiene el número del mes sabiendo que enero es 0.
getSeconds()	Obtiene los segundos de la fecha.
getTime()	Obtiene el valor en milisegundos de la fecha.
setDate(*día*)	Modifica el día de la fecha.
setFullYear(*año*)	Modifica el año de la fecha.
setHours(*hora*)	Modifica la hora de la fecha.
setMilliseconds(*milisegundos*)	Modifica los milisegundos de la fecha.
setMinutes(*minutos*)	Modifica los minutos de la fecha.
setMonth(*mes*)	Modifica el mes de la fecha.
setSeconds(*segundos*)	Modifica los segundos de la fecha.
setTime(*milisegundos*)	Establece la fecha a partir de un valor en milisegundos.
toDateString()	Convierte la fecha a un formato más amigable.
toLocaleString([*params*])	Muestra la fecha en texto en formato local.
toLocaleDateString()	Muestra la fecha sin la hora en formato local.
toTimeString()	Muestra la hora sin la fecha en formato local.
toString()	Muestra la fecha en texto al estilo JavaScript.
toJSON()	Muestra la fecha en formato JSON.

9. Objetos predefinidos

El objeto Date también incluye métodos estáticos:

Método	Utilidad
Date.now()	Devuelve el valor numérico correspondiente al actual número de milisegundos transcurridos desde el 1 de enero de 1970, 00:00:00 UTC, ignorando los segundos intercalares.
Date.parse(*objeto*)	Transforma la cadena que representa una fecha y retorna el número de milisegundos transcurridos desde el 1 de enero de 1970, 00:00:00 UTC, ignorando los segundos intercalares.
Date.UTC(*año, mes, día, horas, minutos, segundos, milisegundos*)	Acepta los mismos parámetros de la forma extendida del constructor (por ejemplo, del 2 al 7) y retorna el número de milisegundos transcurridos desde el 1 de enero de 1970, 00:00:00 UTC, ignorando los segundos intercalares.

Objeto Date
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Date

9. Objetos predefinidos

3) MATH – objeto para realizar operaciones matemáticas.

No es un objeto de función, no se puede editar y todas su propiedades y métodos son estáticos.

Constantes matemáticas definidas en el objeto Math:

Constante	Utilidad
Math.E	Constante de Euler, aproximadamente 2,718.
Math.LN10	Logaritmo natural de 10, aproximadamente 2,303.
Math.LN2	Logaritmo natural de 2, aproximadamente 0,693
Math.LOG10E	Logaritmo de E con base 10, aproximadamente 0,434
Math.LOG2E	Logaritmo de E con base 2, aproximadamente 1,443.
Math.PI	Ratio de la circunferencia con respecto a su diámetro, aproximadamente 3,14159.
Math.SQRT1_2	Raíz cuadrada de ½, aproximadamente 0,707.
Math.SQRT_2	Raíz cuadrada de 2, aproximadamente 1,414.

Objeto Math
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Math

9. Objetos predefinidos

Los métodos más utilizados son:

Método	Utilidad
Math.abs(*n*)	Valor absoluto de *n*.
Math.acos(*n*)	Arcocoseno de *n*.
Math.asin(*n*)	Arcoseno de *n*.
Math.atan(*n*)	Arcotangente de *n*.
Math.ceil(*n*)	Redondea *n* (decimal) a su entero superior.
Math.cos(*n*)	Coseno de *n*.
Math.exp(*n*)	e^n.
Math.floor(*n*)	Redondea *n* (decimal) a su entero inferior.
Math.log(*n*)	Logaritmo decimal de *n*.
Math.max(*a*,*b*)	Devuelve el mayor de dos números, *a* y *b*.
Math.min(*a*,*b*)	Devuelve el menor de dos números, *a* y *b*.
Math.pow(*a*,*b*)	a^b.
Math.random()	Devuelve un número aleatorio entre 0 y 1.
Math.round(*n*)	Redondea *n* a su entero más próximo.
Math.sin(*n*)	Seno de *n*.
Math.sqrt(*n*)	Raíz cuadrada de *n*.
Math.tan(*n*)	Tangente de *n*.
Math.trunc(*n*)	Devuelve la parte entera de *n*, eliminando los decimales.

con 5 se va al superior, menor a 5 baja al inferior

9. Objetos predefinidos

4) BOOLEAN – objeto contenedor de un valor booleano, es decir, un valor lógico true o false. No deben confundirse los valores booleanos primitivos (true y false) con los valores true y false del objeto Boolean

El valor pasado a Boolean como primer parámetro se convierte en un valor booleano, si es necesario. Si el valor se omite o tiene uno de los siguientes valores: 0, -0, null, false, NaN, undefined, “”, el objeto tiene un valor inicial de **false**. Para todos los demás valores, se obtendrá inicialmente el valor de **true**.

```
let b1 = new Boolean(NaN);
let b2 = new Boolean(undefined);
let b3 = new Boolean("");
let b4 = new Boolean([]);
let b5 = new Boolean("false");
```

```
▸ Boolean {false}
▸ Boolean {false}
▸ Boolean {false}
▸ Boolean {true}
▸ Boolean {true}
>
```

NOTA: Cuidado con los valores Boolean en las expresiones condicionales:

```
let logico1 = false;
let logico2 = new Boolean(false);
console.log(logico1);
console.log(logico2);
if (logico1)
        console.log("logico1 entra");
if (logico2)
        console.log("logico2 entra");
```

```
false
▸ Boolean {false}
logico2 entra
>
```

La salida tiene un comportamiento inesperado, aunque inicialmente ambas definiciones de variables se muestran como false, no se evalúan como tal cuando se utilizan en expresiones condicionales. El objeto creado a partir de Boolean se ha evaluado como true.

REGLA: no utilizar un objeto Boolean como sinónimo de un booleano primitivo.

si se van a usar booleanos,
se utilizan tipos de datos booleanos,
no objetos Boolean

Objeto Boolean
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Boolean

9. Objetos predefinidos

5) EXPRESIONES REGULARES – patrón que se utiliza para buscar coincidencias en las cadenas de texto, de manera que puedan resolver tareas frecuentes como la validación de datos.

Se pueden construir expresiones regulares de dos formas:

1) Usando una expresión regular literal (las barras delimitan la expresión): `let erLiteral = /[0-9]/;`

2) Llamando a la función constructora del objeto **RegExp**: `let erObjeto = new RegExp('[0-9]');`

Por ejemplo, con las expresiones anteriores serán validadas todas las cadenas que tengan algún número: "101 Dálmatas", "875", etc. Pero no serán válidas: "Faro" o "xMssT_pol#xhN".

La forma de comprobar si una cadena cumple con los criterios del patrón establecido en la expresión regular es a través del método **test()**.

Test() recibe como parámetro una cadena a comprobar y devuelve true en caso de superar la validación o false en caso contrario

```
let erObjeto = new RegExp('[0-9]');
console.log(erObjeto.test("a"));
console.log(erObjeto.test("almamia"));
console.log(erObjeto.test("alma66Mia"));
console.log(erObjeto.test("987"));
```

la salida de la pieza de código anterior será **false false true true.**

Expresiones regulares
https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Regular_expressions

9. Objetos predefinidos

Debido a la gran variedad de expresiones que se pueden indicar en una cadena de caracteres y las estrategias para validarlas, JS ha desarrollado una sintaxis que facilita su uso. Algunos de los modificadores más utilizados son:

I. ***MODIFICADOR i*** – no distingue entre mayúsculas y minúsculas:

```
let er = /a/;
console.log(er.test("pizza")); // Escribe true
console.log(er.test("TACO")); // Escribe false
let er2 = /a/i;
console.log(er2.test("pizza")); // Escribe true
console.log(er2.test("TACO")); // Escribe true
```

II. ***MODIFICADOR ^*** - fuerza que la cadena empiece por el carácter inmediatamente posterior:

```
let er = /^a/;
console.log(er.test("pizza")); // Escribe false
console.log(er.test("TACO")); // Escribe false
console.log(er.test("armario")); // Escribe true
```

III. ***MODIFICADOR $*** - fuerza a que la cadena termine por el carácter inmediatamente anterior:

```
let er = /pon$/;
console.log(er.test("ponderado")); // Escribe false
console.log(er.test("posicion")); // Escribe false
console.log(er.test("tapon")); // Escribe true
```

9. Objetos predefinidos

IV. ***MODIFICADOR .*** – representa un carácter cualquiera:

```
let er = /ar.on/;
console.log(er.test("arcon")); // Escribe true
console.log(er.test("arpon")); // Escribe true
console.log(er.test("Aaron")); // Escribe false
```

V. ***MODIFICADOR []*** – establecen caracteres opcionales. La expresión la cumpliría cualquier cadena que contenga alguno de los elementos indicados entre corchetes:

```
let er = /[aeiou]/;
console.log(er.test("SOS")); // Escribe false
console.log(er.test("col")); // Escribe true
console.log(er.test("Pfff!")); // Escribe false
```

VI. ***MODIFICADOR [^expresión]*** - ^ como primer elemento de unos corchetes indica un carácter no permitido. Por ejemplo, para que sólo se validen las cadenas que no sean completamente numéricas:

```
let er = /[^0-9]/;
console.log(er.test("cabo")); // Escribe true
console.log(er.test("526")); // Escribe false
console.log(er.test("bueno")); // Escribe true
console.log(er.test("p4ssw0rd")); // Escribe true
```

9. Objetos predefinidos

VII. MODIFICADORES DE CARDINALIDAD – permiten configurar repeticiones de expresiones. Los más usados son:

Método	Utilidad
exp?	Halla ninguna o una vez el elemento *exp*. Por ejemplo, **/a?sa?/** coincide con «as» en «pecas» y «asa» en «casados». Si se usa inmediatamente después de cualquiera de los cuantificadores ***, +, ?,** o **{},** hace que el cuantificador no sea maximalista (es decir, que coincida con el mínimo número de veces), a diferencia del predeterminado, que es maximalista (coincide con el máximo número de veces).
exp*	Concuerda cero o más veces con el elemento *exp*. Por ejemplo, **/ho*/** coincide con «muchoooo» en «Hace muchoooo calor» y «h» en «El vehículo está ardiendo», pero nada en «Para de ladrar».
exp+	Encuentra una o más veces el elemento *exp*, equivalente a **{1,}**. Por ejemplo, **/r+/** coincide con la letra «r» en «Brody» y con todas las letras «r» en «Brrrrrr! ».
exp{n}	Donde *n* es un número entero positivo, concuerda exactamente con *n* apariciones del elemento *exp*. Por ejemplo, **/r{2}/** no coincide con la «r» de «Brody», pero coincide con todas las «r» de «carro» y las dos primeras «r» en «Brrrrrr! ».
exp{n,}	Donde *n* es un número entero positivo, concuerda con al menos *n* apariciones del elemento *exp*. Por ejemplo, **/a{2,}/** no coincide con las «a» en «maracaná», pero coincide con todas las «a» en «maamaa» y en «caaaaaaaraaaamelo».
exp{m,n}	Donde *n* es 0 o un número entero positivo, *m* es un número entero positivo y $m > n$ coincide con al menos *n* y como máximo *m* apariciones del elemento *exp*. Por ejemplo, **/e{1,3}/** no coincide con nada en «castaña», la «e» en «mesa», las dos «e» en «peero» y las tres primeras «e» en «seeeeeguro».

9. Objetos predefinidos

VIII. ***MODIFICADOR ()*** – permiten agrupar expresiones, aumentando la complejidad del patrón. Por ejemplo, para validad la cadena maria#5jorge#9 que tiene el patrón: 5 letras de la 'a' a la 'z', después #, después un número del '0' al '9' y después repetir lo anterior, sería:

```
let er = /([a-z]{5}#[0-9]){2}/;
console.log(er.test("maria#5jorge#9")); // Escribe true
console.log(er.test("maria#5jorge#")); // Escribe false
```

IX. ***MODIFICADOR |*** - indica una opción, es decir, valida lo que está a su derecha o a su izquierda. Por ejemplo, para validar un móvil en España (nueve dígitos que comienzan por 6, 7 u 8):

```
let er = /(6|7|8)([0-9]{8})/;
console.log(er.test("615833678")); // Escribe true
console.log(er.test("715833678")); // Escribe true
console.log(er.test("815833678")); // Escribe true
console.log(er.test("515833678")); // Escribe false
console.log(er.test("915833678")); // Escribe false
console.log(er.test("61583678")); // Escribe false
```

9. Objetos predefinidos

X. MODIFICADORES ABREVIADOS se trata de un conjunto de símbolos a los que precede la barra invertida \.

Funcionan muy bien con Unicode y permite escribir expresiones de una forma más ágil.

Los símbolos más usados son:

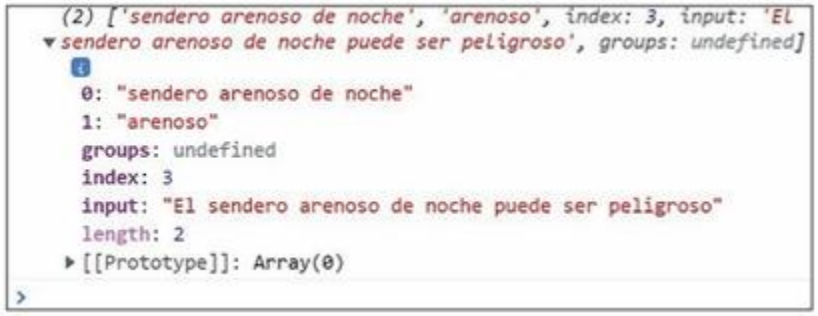
Símbolo	Utilidad
\d	Cualquier dígito numérico.
\D	Cualquier carácter salvo los dígitos numéricos.
\s	Espacio en blanco.
\S	Cualquier carácter salvo el espacio en blanco.
\w	Cualquier carácter alfanumérico: [a-zA-Z0-9].
\W	Cualquier carácter que no sea alfanumérico: [^a-zA-Z0-9].
\0	Carácter nulo.
\n	Carácter de nueva línea.
\t	Carácter tabulador.
\\	El símbolo \.
\"	Comillas dobles.
\'	Comillas simples.
\c	Escapa el carácter *c*.
\ooo	Carácter Unicode empleando la notación octal.
\xff	Carácter ASCII empleando la notación hexadecimal.
\uffff	Carácter Unicode empleando la notación hexadecimal.

9. Objetos predefinidos

XI. ***MÉTODO EXEC()*** – se utiliza para realizar una búsqueda sobre las coincidencias de una expresión regular en una cadena específica, devolviendo un array en caso de éxito o null en caso contrario. Es una alternativa más potente que test() y que muestra más información de las coincidencias.

```
let exreg = /sendero\s(arenoso).+?noche/ig;
let res = exreg.exec('El sendero arenoso de noche puede ser peligroso');
```

Busca "sendero arenoso" seguido de "noche", ignorando los caracteres que encuentre en medio, y además ignora las mayúsculas y minúsculas. La información del objeto devuelto es:



La salida se interpreta:

- **0:** primer elemento que cumple con la expresión regular
- **index:** número que indica la primera posición en la que se encontrón el texto
- **input:** almacena el texto original donde se realizó la búsqueda
- **índices mayores que 0:** indican las coincidencias con las subcadenas buscadas en agrupaciones con paréntesis

ACTIVIDAD 6: VALIDACIÓN DE CORREOS

Crea un programa que incluya una función que haciendo uso de las expresiones regulares permita validar el formato de una dirección de correo electrónico.

Las normas para validar un email son:

- @ es obligatoria, separa la primera parte (izquierda) de la segunda (derecha)
- Primera parte:
  - Acepta mayúsculas y minúsculas, caracteres numéricos y los caracteres especiales: # * + & ‘ ! % @ ? { ^ } “
  - Acepta todos los caracteres punto (.) que se deseen, pero no puede ser ni el primer ni el último carácter y tampoco pueden ir seguidos
- Segunda parte: acepta puntos, dígitos, guiones y letras

Por ejemplo:

hola@tu.casa.net – es válido

mi.email.140dominio.com – no es válido

Gracias