Unidad 3 - Trabajo avanzado con funciones y objetos

Iterables y array-like

- **Iterables**: Objetos que se pueden recorrer, como arrays y strings, utilizando `for..of`.

```
let array = [1, 2, 3, 4];

console.log("Recorrer el array: " + array);

for (elemento of array) {
    console.log("Elemento: " + elemento);
}

let cadena = "Hola caracola";

console.log("Recorrer la cadena: " + cadena);

for (caracter of cadena) {
    console.log("Caracter: " + caracter);
}
```

- **Array-like**: Objetos con propiedad `.length` e indexables (accedibles con `objeto[i]`). Incluye arrays y strings.
- **Array.from()**: Convierte iterables o array-like en arrays.

```
let arr = Array.from(iterable-o-array-like);

// Ejemplo:
let arr = Array.from("Hola");
```

Objetos Map

- **Map**: Permite almacenar pares clave-valor donde las claves son únicas y pueden ser de cualquier tipo.

```
v = new Map()
let listaCompra = new Map();

// v = new Map([[clave1, valor1], [clave2, valor2], [clave3, valor3],   ..
let listaCompra = new Map([["ajos", 5], ["tomates", 3], ["cebollas", 1],])
```

- **Métodos**:

- `set(clave, valor)`: Añade o actualiza una clave. Devuelve un puntero al propio map, permitiendo que se puedan encadenar.

```
listaCompra.set("ajos", 5);
listaCompra.set("tomate", 3);
listaCompra.set("cebollas", 1);

listaCompra.set("ajos", 5).set("tomate", 3).set("cebollas", 1);
```

- `size`: Obtiene el número de pares clave-valor del map.

```
console.log(listaCompra.size); // 3
```

- `has(clave)`: Verifica si una clave existe.

```
console.log(listaCompra.has("tomate") ? "Hay tomate" : "No hay tomate")
```

- `get(clave)`: Devuelve el valor asociado a una clave. Si no existe, devuelve `undefined`.

```
let numTomates = listaCompra.get("tomate"); // 3
console.log(numTomates); // 3
```

- `delete(clave)`: Elimina un par.

```
listaCompra.delete("tomate");
console.log(listaCompra.size); // 2
console.log(listaCompra.has("tomate") ?"Hay tomate" : "No hay tomate");
```

- `clear()`: Elimina todos los pares.

```
listaCompra.clear();
console.log(listaCompra.size); // 0
```

- **Recorrido**:
  - `for of`: En cada iteración se tomará como valor un array de dos elementos (clave y valor).

```
for (let item of listaCompra) {
    console.log("qué compro: " + item[0] + ", unidades: " + item[1]);
}
```

  - `.forEach((valor, clave) => {})`: En cada iteración, se puede acceder al valor del par (primer parámetro) y a su calve (segundo parámetro).

```
listaCompra.forEach((valor, clave) => {
    console.log("qué compro: " + clave + ", unidades: " + valor);
});
```

- `.entries()`: Devuelve un iterable de arrays de elementos clave y valor.

```
for (let item of listaCompra.entries()) {
    console.log("qué compro: " + item[0] + ", unidades: " + item[1]);
}
```

- `.keys()`: Devuelve un iterable con las claves del map.

```
for (let verdura of listaCompra.keys()) {
    console.log("qué compro: " + verdura);
}
```

- `.values()`: Devuelve un iterable con los valores del map.

```
for (let unidades of listaCompra.values()) {
    console.log("unidades: " + unidades);
}
```

```
// Crear el map
let listaCompra = new Map();

// Añadir pares: map.set
listaCompra.set("ajos", 5);
listaCompra.set("tomate", 3);
listaCompra.set("cebollas", 1);

// Obtener el número de pares: map.size
console.log(listaCompra.size); // 3

// Comprobar si está una clave: map.has()
console.log(listaCompra.has("tomate") ? "Hay tomate" : "No hay tomate"); // H

// Obtener el valor de una clave: map.get()
let numTomates = listaCompra.get("tomate"); // 3
console.log(numTomates); // 3

// Eliminar un par: map.delete
listaCompra.delete("tomate");
console.log(listaCompra.size); // 2
console.log(listaCompra.has("tomate") ?"Hay tomate" : "No hay tomate");

// Eliminar todos los pares: map.clear()
listaCompra.clear();
console.log(listaCompra.size); // 0

// Volver a llenar el map, utilizando .set encadenados
listaCompra.set("ajos", 5).set("tomate", 3).set("cebollas", 1);
```

```
// Recorrer un map con for of
for (let item of listaCompra) {
    console.log("qué compro: " + item[0] + ", unidades: " + item[1]);
}

// Recorrer un map con .foreach()
listaCompra.forEach((valor, clave) => {
    console.log("qué compro: " + clave + ", unidades: " + valor);
});

// Acceder a .entries()
for (let item of listaCompra.entries()) {
    console.log("qué compro: " + item[0] + ", unidades: " + item[1]);
}

// Acceder a .keys()
for (let verdura of listaCompra.keys()) {
    console.log("qué compro: " + verdura);
}

// Acceder a .values()
for (let unidades of listaCompra.values()) {
    console.log("unidades: " + unidades);
}
```

Objetos Set

- **Set**: Objeto que almacena valores únicos.

```
let verduras = new Set();

let verduras = new Set(["ajos", "tomates", "cebollas"]);
```

- **Métodos**:
  - `add(valor)`: Añade un valor único. Devuelve un puntero al propio set, permitiendo que se puedan encadenar.

```
// let s2 = s.add(valor)

verduras.add("ajos");
verduras.add("tomate");
verduras.add("cebollas");

verduras.add("ajos").add("tomate").add("cebollas");
```

  - `size`: Obtiene la cantidad de valores.

```
console.log(verduras.size); // 3
```

- `has(valor)`: Verifica si un valor existe.

```
console.log(verduras.has("tomate") ? "Hay tomate" : "No hay tomate"); /
```

- `delete(valor)`: Elimina un valor.

```
verduras.delete("tomate");
console.log(verduras.size); // 2
console.log(verduras.has("tomate") ?"Hay tomate" : "No hay tomate");
```

- `clear()`: Elimina todos los valores.

```
verduras.clear();
console.log(verduras.size); // 0
```

- **Recorrido**:
  - `for of`: En cada iteración se tomará un valor del set.

```
for (let valor of verduras) {
    console.log("qué compro: " + valor );
}
```

  - `.forEach((valor) => {})`: En cada iteración, se puede acceder a un valor del set.

```
verduras.forEach((valor) => {
    console.log("qué compro: " + valor);
});
```

  - `.entries()`: Devuelve un iterable con arrays de dos elementos: clave y valor.

```
for (let item of verduras.entries()) {
    console.log("qué compro: " + item[0] + ", qué compro: " + item[1]);
}
```

  - `.keys()`: Devuelve un iterable con los valores del set.

```
for (let verdura of verduras.keys()) {
    console.log("qué compro: " + verdura);
}
```

  - `.values()`: Devuelve un iterable con los valores del set.

```
for (let unidades of verduras.values()) {
    console.log("unidades: " + unidades);
```

```
}
```

```
// Crear el set
let verduras = new Set();

// Añadir valores
verduras.add("ajos");
verduras.add("tomate");
verduras.add("cebollas");

// Obtener el número de valores
console.log(verduras.size); // 3

// Comprobar si está una valor: set.has()
console.log(verduras.has("tomate") ? "Hay tomate" : "No hay tomate"); // Hay

// Eliminar un valor: set.delete
verduras.delete("tomate");
console.log(verduras.size); // 2
console.log(verduras.has("tomate") ?"Hay tomate" : "No hay tomate"); // No ha

// Eliminar todos los valores: set.clear()
verduras.clear();
console.log(verduras.size); // 0

// Volver a llenar el set, utilizando .add encadenados
verduras.add("ajos").add("tomate").add("cebollas");

// Recorrer un set con for of
for (let valor of verduras) {
    console.log("qué compro: " + valor);
}

// Recorrer un set con .foreach()
verduras.forEach((valor) => {
    console.log("qué compro: " + valor);
});

// Acceder a .entries()
for (let item of verduras.entries()) {
    console.log("qué compro: " + item[0] + ", qué compro: " + item[1]);
}

// Acceder a .keys()
for (let verdura of verduras.keys()) {
    console.log("qué compro: " + verdura);
}
```

```
// Acceder a .values()
for (let unidades of verduras.values()) {
    console.log("unidades: " + unidades);
}
```

Métodos de Object (`Object.keys()`, `Object.values()`, `Object.entries()`)

Los métodos estáticos de `Object` como `Object.keys()`, `Object.values()`, y `Object.entries()` permiten iterar sobre las propiedades de un objeto, devolviendo arrays en lugar de iterables, a diferencia de los métodos de `Map` y `Set`.

Método	Descripción	Ejemplo
`Object.keys(objeto);`	Devuelve un array con el nombre de las propiedades de objeto.	`Object.keys({a:3, b:2}); // => ['a', 'b']`
`Object.values(objeto);`	Devuelve un array con los valores de objeto.	`Object.values({a:3, b:2}); // => [3, 2]`
`Object.entries(objeto);`	Devuelve array con los pares propiedad-valor de objeto.	`Object.entries({a:3, b:2}); // => [['a',3], ['b',2]]`

```
let user = {
    name: "John",
    age: 30,
};

/*
    Object.keys(user) = ["name", "age"]
    Object.values(user) = ["John", 30]
    Object.entries(user) = [ ["name","John"], ["age",30] ]
 */

// bucle sobre los valores
for (let valor of Object.values(user)) {
    console.log(valor); // John, luego 30
}
```

`Object.fromEntries(iterable)`

Convierte un iterable de pares clave-valor (como un `Map` o un array de arrays) en un objeto.

```
let listaCompra = new Map();

listaCompra.set("ajos", 5).set("tomate", 3).set("cebollas", 1);

l = Object.fromEntries(listaCompra);
```

Objetos `Date`

`Date` almacena la fecha en milisegundos desde el 1 de enero de 1970 a las 00:00:00 UTC, permitiendo el uso de fechas anteriores mediante números negativos. Los meses y días de la semana están numerados, comenzando por 0 (enero y domingo, respectivamente).

Crear objetos `Date`

- **Fecha y hora actual:** `let actual = new Date();`
- **Desde milisegundos:** `let fecha = new Date(numMilisegundos);`
- **Con parámetros:** `let fecha = new Date(año, mes, [día, hora, min, seg, ms]);`
- **Desde cadena:** `let fecha = new Date(cadenaFecha);` (admite formatos como `YYYY-MM-DD`, `MM/DD/YYYY`, entre otros).
    - Formatos disponibles: `"Wed Mar 25 2015 09:23:45 GMT +0100"` (hora estándar de Europa central), `"October 12, 2016 10:30:00"`, `"January 25 2015"`, `"Jan 25 2015", "25 Jan 2015"`, `"2016-05-12T12:34:25Z"` (la T separa la fecha de la hora y la Z indica la zona horaria), `"2015-03-25T12:00:00-06:30"` (para modificar la zona horaria, lugar de Z pondremos +HH:MM o -HH:MM), `"2016-05-12"`, `"2016-05", "2016"`, `"05/12/2016"`.

```
let hoy = new Date(); // Crear un Date con la fecha y hora actuales
console.log(hoy);

let enero01_1970 = new Date(0); // Milisegundos desde el 01/01/1970
console.log(enero01_1970);

let sabado = new Date("2022-10-29"); // 29-10-2022 02:00
console.log(sabado);

let domingo = new Date("2022-10-30 00:00:00"); // 29-10-2022 00:00
console.log(domingo);

let fiesta = new Date(2022, 11, 1); // año: 4 dígitos,
// mes: de 0 a 11 (11: diciembre)
console.log(fiesta);
```

Métodos `get` de `Date`

Método	Descripción
`fecha.getDate();`	Día del mes (1-31)
`fecha.getMonth();`	Mes (0=enero, ..., 11=diciembre)
`fecha.getFullYear();`	Año de 4 dígitos
`fecha.getHours();`	Hora
`fecha.getMinutes();`	Minutos
`fecha.getSeconds();`	Segundos
`fecha.getMilliseconds();`	Milisegundos

Método	Descripción
`fecha.getDay();`	Día de la semana (0=domingo)
`fecha.getTime();`	Milisegundos desde el 01/01/1970

```
navidad = new Date("2022-12-25"); // 25-12-2022
console.log(navidad.getFullYear()); // 2022
console.log(navidad.getMonth()); // 11
console.log(navidad.getDate()); // 25
console.log(navidad.getHours()); // 0
console.log(navidad.getMinutes()); // 0
console.log(navidad.getSeconds()); // 0
console.log(navidad.getMilliseconds()); // 0
```

Métodos `set` de `Date`

Permiten modificar los valores de las propiedades de fecha.

Método	Descripción
`fecha.setDate(dia);`	Cambia el día del mes
`fecha.setMonth(mes);`	Cambia el mes (0=enero)
`fecha.setFullYear(año);`	Cambia el año
`fecha.setHours(hora);`	Cambia la hora (0-24)
`fecha.setMinutes(min);`	Cambia los minutos (0-59)
`fecha.setSeconds(seg);`	Cambia los segundos: (0-59)
`fecha.setMilliseconds(ms);`	Cambia los milisegundos
`fecha.setTime(ms);`	Milisegundos desde el 01/01/1970

```
navidad.setFullYear(2023);
console.log(navidad);
```

Conversión de `Date` a cadena

Método	Ejemplo de salida
`fecha.toString();`	"Sun Nov 18 2018 18:28:27 GMT+0100"
`fecha.toUTCString();`	"Sun, 18 Nov 2018 17:28:53 GMT"
`fecha.toDateString();`	"Sun Nov 18 2018"

Métodos estáticos

Método	Descripción
`Date.parse(cadenaFecha);`	Convierte una fecha en milisegundos desde 01/01/1970
`Date.now();`	Devuelve los milisegundos actuales desde 01/01/1970

Observaciones sobre fechas

- **Autocorrección:** Las fechas se corrigen si están fuera de rango.
- **Conversiones:** Se pueden convertir fechas a números para cálculos, como la diferencia en milisegundos entre dos fechas.

Asignación múltiple y operadores `rest` / `spread`

Asignación múltiple (destructuración)

La asignación múltiple o destructuración permite asignar valores de arrays o propiedades de objetos a variables en una sola instrucción, lo que facilita la creación de código más corto y legible. Este enfoque admite valores predeterminados.

- **En arrays**: Las variables se agrupan con corchetes `[]` y corresponden por posición a los valores del array.

```
// Ejemplo arrays

//Definición de variables
let [x, y, z] = [5, 1, 3, 4]; // x => 5, y => 1, z => 3

//Asignación de variables: intercambiar contenidos
let x = 5, y = 1;
[x, y] = [y, x]; // x => 1, y => 5

let [x, y, z = 1, t = 2, v] = [5, , , 10];
// x => 5, y => undefined,
// z => 1, t => 10, v => undefined;
```

- **En objetos**: Las variables se agrupan con llaves `{}` y se corresponden con el nombre de las propiedades. Es posible renombrar propiedades con `:`, y si se asignan dentro de una expresión deben ir entre paréntesis.

```
// Ejemplo objetos

//Definición de variables
let { a, c = 1, d, e } = { a: 5, e: 3, f: 4 }; // a => 5, c => 1,
// d => undefined, e => 3

//Asignación de variables ()
let a, c, d;
({ a, c = 1, d } = { a: 5, e: 3 }); // a => 5, c => 1, d => undefined
```

```
// Ejemplo de llamada a una función
let a = 5,
c = 3,
d = 4;
```

```
// ES5-agrupar variables en un objeto con propiedades de igual nombre // a la
let objES5 = { a: a, c: c, d: d }; // objES5 => {a:5, c:3, d:4}

// ES6-las mismas variables se agrupan así:
let objES6 = { a, c, d }; // objES6 => {a:5, c:3, d:4}
```

Operador `rest ...`

El operador rest permite agrupar el resto de los elementos no asignados en un array u objeto. Esto resulta útil en arrays y objetos para capturar múltiples valores de manera flexible.

```
// Ejemplo arrays

let [x, ...rest1] = [0, 2, 3]; // x => 0, rest1 => [2, 3]
[x, ...rest2] = [4, x, ...rest1]; // x => 4, rest2 => [0, 2, 3]
```

```
// Ejemplo objetos

let { a, ...x } = { a: 5, b: 1, c: 2 }; // a => 5, x => {b:1, c:2}
let b, y;
({ b, ...y } = { a: 1, b: 2 }); // b => 2, y => {a:1}
```

```
// Ejemplo de llamada a una función
// Nos permite asignar a un array los parámetros por defecto.

function suma(...valores) {
    let acu = 0;
    for (let valor of valores) {
        acu += valor;
    }
    return acu;
}

console.log(suma(1, 2, 3, 4, 5, 6));
```

Operador `spread ...`

El operador spread permite expandir elementos de un array u objeto en otros contextos, como al pasar argumentos en funciones o combinar objetos y arrays.

```
// Ejemplo arrays

let a = [2, 3];
let b = [0, 1, ...a]; // b => [0, 1, 2, 3]
```

```
// Ejemplo objetos

let x = {a:5, b:1};
let y = {...x, c:6, d:7}; // y => {a:5, b:1, c:6, d:7}
```

```
// Ejemplo de llamada a una función

let datos = [1, 2, 3, 4, 5, 6];
console.log(Math.min(...datos));
```

```
// Ejemplo de llamada a una función

let a = [2, 3];
f(0, ...a); // => f(0, 2, 3)
```

JSON en JavaScript

JSON (JavaScript Object Notation) es un formato de texto ligero, derivado del estándar ECMAScript, ampliamente usado para representar y compartir datos estructurados entre diferentes lenguajes de programación. JSON permite la representación de cuatro tipos de datos primitivos (cadena de texto, número, booleano y nulo) y dos tipos estructurados (objetos y arreglos). No soporta otros valores de JavaScript como funciones, `undefined` o ciertos objetos internos que contienen métodos.

El objeto JSON en JavaScript

JavaScript incluye el objeto `JSON`, que ofrece métodos para convertir valores a JSON y para reconstruir un objeto JavaScript desde una cadena JSON:

- `JSON.stringify(valor)`: Convierte un valor JavaScript en una cadena JSON. Algunos valores experimentan cambios durante la conversión:

  ```
  JSON.stringify({}); // '{}'
  JSON.stringify(true); // 'true'
  JSON.stringify(null) // 'null'
  JSON.stringify("hola"); // '"hola"'
  JSON.stringify(127) // '127'
  JSON.stringify([1, 2, 3]) // '[1, 2, 3]'
  JSON.stringify({a:27, b:"hola"}) // '{"a":27,"b":"hola"}'
  JSON.stringify([1, "false", false]); // '[1,"false",false]'
  ```

- Valores como `NaN`, `Infinity` y `-Infinity` se convierten en `null`.

  ```
  JSON.stringify([NaN, null, Infinity]); // '[null,null,null]'
  ```

- Objetos `Date` se serializan en formato de cadena ISO 8601.

```
JSON.stringify(new Date(2006, 0, 2, 15, 4, 5));
// '"2006-01-02T15:04:05.000Z"'
```

- `JSON.parse(cadena)`: Convierte una cadena JSON en su valor de JavaScript correspondiente.

```
JSON.parse('null') => null
```

Temporización en JavaScript: `setTimeout` y `setInterval`

JavaScript ofrece dos funciones de temporización: `setTimeout` y `setInterval`, útiles para programar la ejecución de código tras un intervalo específico.

Función	Descripción
`setInterval(función, milisegundos)`	Ejecuta la función de forma repetitiva cada cierto número de milisegundos. Devuelve un identificador único, que puede usarse para detener la ejecución con `clearInterval`.
`setTimeout(función, milisegundos, par1, par2…)`	Ejecuta la función una única vez después de que pase el tiempo indicado. Devuelve un identificador único que puede usarse para cancelar la ejecución mediante `clearTimeout`. Los parámetros `par1`, `par2`, etc., son argumentos opcionales que se pueden pasar a la función.

`clearInterval(identificador);`
 	
`clearTimeout(identificador);`
	Cancelan la ejecución de `setInterval` o `setTimeout` correspondientes al identificador dado.

```
// Ejecutar una función cada 3 segundos

function muestra() {
    console.log("Han pasado 3 segundos");
}
let v = setInterval(muestra, 3000);

let v = setInterval(function () {
    console.log ("Han pasado 3 segundos");
}, 3000);

let v = setInterval(() => {
    console.log ("Han pasado 3 segundos");
}, 3000);
```

```
// Detener la ejecución tras 10 segundos

setTimeout(function () {
    clearInterval(v);
```

```
    console.log("Cancelada la orden");
}, 10 * 1000);
```

Clases en JavaScript (ES6)

La versión ES6 de JavaScript introdujo una sintaxis simplificada para crear clases usando la palabra clave `class`, facilitando el desarrollo de código orientado a objetos. Esta sintaxis permite definir propiedades, métodos y estructuras de herencia en JavaScript de forma más clara y legible.

Declaración básica de clases

La estructura básica de una clase es la siguiente:

```
class MyClass {
    prop = valor; // Solo últimas versiones del lenguaje
    prop;

    constructor() { ... }; // Opcional
    metodo1() {....};
    metodo2() {....};

    get algo() {};
    set algo() {};
}
```

- Las clases en JavaScript funcionan en *modo estricto* por defecto.
- Se deben definir antes de usarlas y se instancian con `new`.
- Los métodos y propiedades se acceden mediante `this`.

```
// Ejemplo de clase

class Persona {
    constructor(nombre, apellido, nacimiento) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.nacimiento = nacimiento;
    }
    saluda() {
        console.log("Hola, soy " + this.nombre);
    }
}
```

Expresiones de clase

Además de las declaraciones, las clases pueden crearse como expresiones asignadas a una variable:

```
let Libro = class {
    constructor(titulo, autor) {
        this.titulo = titulo;
        this.autor = autor;
    }
    datos() {
        console.log("Titulo: " + this.titulo + "; autor: " + this.autor);
    }
};
let miLibro = new Libro("El Quijote", "Cervantes");
```

Creación de objetos de una clase

Los objetos en JavaScript se crean a partir de clases utilizando la palabra clave `new`. Al instanciar una clase, se llama automáticamente al método `constructor` de la clase para inicializar el objeto.

```
let persona = new Persona("Ada", "Lovelace", 1815);
persona.saluda();
```

Definición de métodos de clase

Los métodos de una clase se pueden definir de dos maneras. La forma más común es incluir el nombre del método seguido de su implementación dentro de la clase. También es posible asignar una función anónima a una propiedad.

```
// Ejemplo de sintaxis habitual

class Ejemplo {
  metodo() {
    console.log("Este es un método");
  }
}
```

```
// Ejemplo con función anónima

class Ejemplo {
  metodo = function() {
    console.log("Este es un método con función anónima");
  }
}
```

```
// Invocar un método

let objeto = new Ejemplo();
objeto.metodo();
```

`Getters` y `Setters`

Los getters y setters permiten acceder y modificar propiedades de un objeto de manera controlada, implementando el principio de encapsulación. En JavaScript, los `get` y `set` deben tener nombres distintos al de la propiedad subyacente para evitar un bucle de referencia; una práctica común es anteponer `_` al nombre de la propiedad interna.

```
// Ejemplo 1

class Telefono {
    constructor(marca) {
        this._marca = marca;
    }
    get marca() {
        return this._marca;
    }
    set marca(marca) {
        this._marca = marca;
    }
}

let miTelefono = new Telefono("Google");
//Para utilizar estos métodos, no es necesario utilizar los paréntesis ()
miTelefono.marca = "iPhone";

// Ejemplo 2

class Persona {
    set nombre(valor) {
        if (typeof valor != "string") {
            console.log("Esperaba una cadena de texto");
            return;
        }
        this._nombre = valor;
    }
    get nombre() {
        return this._nombre;
    }
}

let p = new Persona();
console.log("** PROBANDO LOS SETTERS Y LOS GETTERS");
console.log("Sentencia: p.nombre = 12");

p.nombre = 12; // Error: esperaba una cadena de texto
console.log("p.nombre: " + p.nombre);
console.log("p._nombre: " + p._nombre);
console.log("Sentencia: p.nombre = Lucas");
```

```
p.nombre = "Lucas";
console.log("p.nombre: " + p.nombre);
console.log("p._nombre: " + p._nombre);
```

Herencia de las clases

En JavaScript, todas las clases heredan de la clase base `Object` a menos que se indique lo contrario. La herencia se implementa con la palabra clave `extends` (`class ClaseHijo extends ClasePadre {...};`), que permite a una clase hija heredar propiedades y métodos de una clase padre. Para acceder a los métodos y propiedades de la clase padre desde la clase hija, se utiliza la palabra clave `super` (`super.metodoPadre(parámetros);`).

```
// Ejemplo de herencia

class Telefono {
    constructor(marca) {
        this.marca = marca;
    }
    anuncio() {
        return "Ha llegado el nuevo teléfono de " + this.marca;
    }
}
class Modelo extends Telefono {
    constructor(marca, modelo) {
        super(marca);
        this.modelo = modelo;
    }
    anuncio() {
        return super.anuncio() + ": el modelo " + this.modelo;
    }
}
miTelefono = new Modelo("iPhone", "21");
console.log(miTelefono.anuncio());
```

Métodos estáticos en clases

Los métodos estáticos, definidos con `static`, pertenecen a la clase y no a instancias individuales. Estos métodos se invocan directamente a través de la clase sin crear un objeto y son útiles para funciones que no dependen de los datos específicos de una instancia.

- Definir un método estático: `static nombreMetodo() {...}`
- Llamar al método: `NombreClase.nombreMetodo();`

```
// Definición y llamada a un método estático

class Utilidades {
    static saludo() {
        return "Hola desde un método estático";
    }
```

```
}

console.log(Utilidades.saludo()); // "Hola desde un método estático"
```