



**Centro Integrado de Formación Profesional**

**AVILÉS**

Principado de Asturias

**UNIDAD 3:**  
**EL LENGUAJE JAVASCRIPT: ARRAYS,**  
**FUNCIONES Y OBJETOS**

**DESARROLLO WEB EN ENTORNO CLIENTE**

**2º CURSO**

**C.F.G.S. DISEÑO DE APLICACIONES WEB**

## REGISTRO DE CAMBIOS

Versión	Fecha	Estado	Resumen de cambios
1.0	16/10/2024	Aprobado	Primera versión
1.1	17/10/2024	Aprobado	Añadido patrón factoría

## ÍNDICE

ÍNDICE .....	1
UNIDAD 3: EL LENGUAJE JAVASCRIPT. ARRAYS, FUNCIONES Y OBJETOS .....	3
3.1 Funciones.....	3
3.1.1 Funciones predefinidas del lenguaje .....	3
3.1.2 Funciones del usuario. Valores de retorno .....	5
3.1.3 Definición de funciones. Parámetros. El objeto Function.....	5
3.1.4 Valores por defecto para parámetros opcionales.....	6
3.1.5 Llamadas a funciones. Funciones llamantes.....	7
3.1.6 Ejecutar una función en la carga de una página .....	8
3.2 Matrices (Arrays).....	9
3.2.1 Creación de arrays.....	9
3.2.2 Tipos de datos en arrays.....	9
3.2.3 Manipulación de elementos .....	10
3.2.4 Ordenación en matrices .....	10
3.2.5 Arrays multidimensionales.....	11
3.2.6 Pilas y colas.....	11
3.2.7 Operaciones agregadas: recolección, filtrado y reducción .....	12
3.2.8 Usar arguments como una matriz.....	13
3.2.9 Arrays asociativos (HashTables) .....	15

3.2.10 Otros tipos de datos .....	16
3.3 Programación orientada a objetos .....	19
3.3.1 Conceptos básicos de Programación Orientada a Objetos .....	19
3.3.2 Inicialización de objetos en JavaScript.....	20
3.3.3 Clases JavaScript.....	21
3.3.4 Herencia .....	25
3.3.5 La propiedad prototype.....	26
3.3.6 Diseño modular .....	26
3.3.7 Desestructuración.....	29
3.4 Patrones de diseño .....	31
3.4.1 Singleton .....	31
3.4.2 Factoría.....	32
3.4.3 Strategy (estrategia).....	32
ÍNDICE DE FIGURAS.....	35
BIBLIOGRAFÍA – WEBGRAFÍA .....	35

## UNIDAD 3: EL LENGUAJE JAVASCRIPT. ARRAYS, FUNCIONES Y OBJETOS

### 3.1 FUNCIONES

El concepto de función en JavaScript es el mismo que en cualquier lenguaje de programación: un bloque de código identificado por un nombre que puede recibir parámetros y que también puede devolver algo.

#### 3.1.1 FUNCIONES PREDEFINIDAS DEL LENGUAJE

A continuación, se muestra una lista de funciones predefinidas, que se pueden utilizar a nivel global en cualquier parte de código de JavaScript. Estas funciones no están asociadas a ningún objeto en particular. Típicamente, permiten convertir datos de un tipo a otro tipo:

**decodeURI()** Decodifica los caracteres especiales de una URL excepto: , / ? : @ & = + \$ #

**decodeURIComponent()** Decodifica todos los caracteres especiales de una URL.

**encodeURI()** Codifica los caracteres especiales de una URL excepto: , / ? : @ & = + \$ #

**encodeURIComponent()** Codifica todos los caracteres especiales de una URL.

**eval()** Evalúa una cadena y la ejecuta si contiene código u operaciones.

**isFinite()** Determina si un valor es un número finito válido.

**isNaN()** Determina cuando un valor no es un número.

**Number()** Convierte el valor de un objeto a un número.

**parseFloat()** Convierte una cadena a un número real.

**parseInt()** Convierte una cadena a un entero.

**RegExp()** Crea un objeto 'expresión regular' para encontrar texto de acuerdo a un patrón.

Un ejemplo de uso de eval podría ser el siguiente:

```
<script>
    eval("x=50;y=30;document.write(x*y)"); // Muestra 1500
    document.write("<br>", eval("8+6")); // Muestra 14
    document.write("<br>", eval(x+30)); // Muestra 80
</script>
```

En el siguiente enlace se pueden consultar este tipo de funciones:

[https://www.w3schools.com/jsref/jsref\\_obj\\_global.asp](https://www.w3schools.com/jsref/jsref_obj_global.asp)

##### 3.1.1.1 Formateando logs

Una de las funciones predefinidas que se utiliza a menudo y se vio anteriormente es console.log para enviar mensajes a la consola que muestren lo que está sucediendo en una aplicación. Lo que pasa es que frecuentemente, la información mostrada está tan en crudo que es difícil de entender.

Por ejemplo, si se está depurando e interesa ver el valor de una variable, se suele hacer lo siguiente:

```
let x = 2;  
console.log(x);
```

lo que muestra un triste 2 sin saber ni a qué variable se refiere.

Para ayudar a visualizar el número junto con su variable, se puede aplicar un truco consistente en rodear entre llaves a la variable, lo que hará que se muestre no solo la variable sino su valor:

```
console.log({x}) // mostrará {x: 2}
```

lo que aportará más información interesante para conocer el comportamiento de la aplicación.

También puede ser muy útil utilizar especificadores en el mensaje que comiencen por un signo de porcentaje. Estos especificadores permiten registrar ciertos valores en distintos formatos:

- %s – cadena
- %i o %d – entero
- %f – valor en punto flotante
- %o – elemento DOM ampliable
- %O – objeto JavaScript ampliable

Además, se pueden mezclar varios de estos especificadores. El orden de reemplazo es el correspondiente al de la aparición de los parámetros. Por ejemplo:

```
console.log('%ix desarrollador en %s', 10, 'consola');  
// mostrará '10x desarrollador en consola'
```

Incluso, si el formato no es amigable como en un número largo en punto flotante, se puede realizar una conversión a entero, pero únicamente para la visualización:

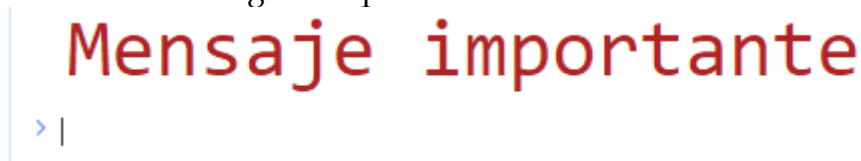
```
console.log('%i', 10.5478678234);
```

En realidad, estos especificadores no son nada nuevo en desarrollo ya que elementos similares se han utilizado históricamente en lenguajes como C o Pascal.

Pero aún hay más, se puede hasta visualizar el texto con un estilo concreto CSS, tal como se ve en el siguiente ejemplo:

```
console.log('%cMensaje importante', 'color:firebrick; font-size:40px');
```

El código anterior mostrará lo siguiente por consola:



#### 1. Mensaje en consola con estilo CSS

### 3.1.2 FUNCIONES DEL USUARIO. VALORES DE RETORNO

El modelo clásico de definición de una función es como sigue:

```
function doblar(a){  
    return a * 2;  
}
```

Como puede verse, un modelo convencional encabezado por la palabra clave function. Pero gracias a la programación funcional, existe un artefacto conocido como expresiones Lambda que introducen una buena parte de los lenguajes avanzados, aunque no sean puramente funcionales. En JavaScript, del uso de las expresiones lambda se llega a lo que se conoce como funciones Arrow (Arrow functions). De esta forma, se expresarían con este mecanismo una serie de funciones:

```
const sumar = (a, b) => a + b;  
const factorial = a => {  
    if (a <= 1){  
        return 1;  
    }  
    return a * factorial(a -1);  
}
```

En este caso se expresan dos funciones, sumar y factorial, mediante estas expresiones lambda. La llamada a las funciones se hace del mismo modo en ambos casos:

```
console.log("doblar", doblar(2));  
console.log("sumar", sumar(1,2));  
console.log("factorial", factorial(2));
```

### 3.1.3 DEFINICIÓN DE FUNCIONES. PARÁMETROS. EL OBJETO FUNCTION

Representa las funciones definidas con la palabra reservada correspondiente. A continuación, se describen algunas de sus propiedades.

El atributo **arguments** es un array de elementos que almacena los parámetros de la función, que no tienen por qué ser necesariamente aquellos que se han escrito explícitamente en la definición de esta.

Otra propiedad interesante se denomina **caller** y contiene una referencia a la función desde la cual se está ejecutando esta o **null** en caso de que la función se ejecute desde el nivel superior. En lo que respecta a los métodos, destacar dos: call, que sirve para ejecutar esta función como método de dicho objeto. Ejemplo:

```
alert("Hola, mundo");
```

cambiada por:

```
alert.call(window, "Hola, mundo");
```

Ello se debe a que **alert** es un método del objeto **window**. Otro ejemplo similar con la salvedad de que se emplea un objeto específicamente desarrollado:

```
let foo = {  
  alert: function(msg){  
    alert(msg + " (1)");  
  }  
};  
foo.alert.call(foo, "Hola, mundo");
```

Estos ejemplos a priori no son útiles, pero sí muy gráficos académicamente hablando.

Por último, el método **apply** con funcionalidad similar a la de **call**, tiene la salvedad de que su segundo argumento es un array. Un ejemplo:

```
alert.apply(window, ["Hola, mundo"]);
```

### 3.1.4 VALORES POR DEFECTO PARA PARÁMETROS OPCIONALES

Una cuestión interesante de las funciones con sobrecargas y parámetros opcionales es poder especificar valores por defecto para los parámetros opcionales (así sería posible omitir todos, aunque estén declarados).

Se puede sacar partido de los valores **truly** y **falsy** generados por las conversiones implícitas a booleanos para así poder especificar valores por defecto para parámetros en caso de que falten. Anteriormente, se expuso que el operador lógico OR (| |) se evalúa con cortocircuito de expresiones lógicas de la siguiente manera:

- Si el primer operando de la comparación es true devuelve ese mismo primer operando.
- Si el primer operando es false entonces se devuelve automáticamente el segundo operando

Sacando partido a esto es muy fácil definir valores por defecto para los parámetros sin tener que escribir condicionales ni código largo que "embarre" la definición de una función. Por ejemplo, si se tiene una función test que toma dos parámetros, a y b, y hay que asegurarse de que ambos tienen sendos valores por defecto, aunque no se hayan especificado, es posible definirla así:

```
function test(a, b) {  
  a = a || 0; //valor por defecto: 0  
  b = b || true; //valor por defecto: true  
  // ----  
}
```

Con esto lo que se consigue es que si el parámetro "a" no se le pasa a la función (es decir, **undefined**) se le asignará un 0 como valor por defecto. Y lo mismo con "b" que tendrá un true como valor por defecto.

Al evaluarse el operador OR (| |) el primer parámetro se convierte en un booleano. Si no está definido esto es equivalente a un **falsy** (o sea, se convierte implícitamente en un false) y por lo tanto se asigna en la propia variable el segundo operando del OR (el valor por defecto).

Si se le pasa una cadena o un objeto de cualquier tipo se evaluará como true y por lo tanto se devolverá el propio objeto (se reasignará a sí mismo).

**OJO:** Esta técnica no sirve si lo que se espera como valor para el parámetro es un booleano y se quiere que el valor por defecto sea true. En este caso si se le pasase un false como valor para el parámetro el efecto que se obtendría es que cambiaría su valor a true, y en la práctica no se conseguiría pasar un false como valor efectivo; así que hay que tener cuidado con esto. Para cualquier otro tipo de parámetro no hay problema.

### 3.1.5 LLAMADAS A FUNCIONES. FUNCIONES LLAMANTES

Una interesante cuestión relativa a la ejecución de funciones es que también **es posible averiguar desde dónde se está invocando una determinada función**. Esta facilidad la proporciona la propiedad **caller** de las funciones. Se accede a ella de la misma manera que a **arguments**, y contiene una referencia a la función que ha llamado a la actual.

Es decir, que podría averiguarse su nombre y hasta su código completo si fuera necesario. Por ejemplo, si se quiere definir, por cualquier motivo, una función que sólo pudiese ser llamada desde el nivel superior del script (es decir, desde otra función, solo desde el objeto global: la ventana en el caso del navegador), se escribiría:

```
function PruebaCaller() {  
    if (PruebaCaller.caller == null) {  
        alert("Me estás llamando desde el sitio correcto");  
    }  
    else {  
        alert("Desde ahí no me puedes llamar");  
    }  
}  
  
PruebaCaller();  
  
function fPrueba() {  
    PruebaCaller();  
}  
  
fPrueba();
```

De este modo, como la primera llamada a **PruebaCaller** se hace desde el nivel superior del script (desde el objeto Global) se obtendrá el mensaje "Me estás llamando desde el sitio correcto". La segunda invocación se efectúa desde el interior de la función **fPrueba**, y por lo tanto el resultado es un mensaje diciendo que no está permitida ya que la propiedad **caller** en ese caso no es nula.

Se podría recorrer el árbol de llamadas (conocido más frecuentemente como **call stack**) de la función actual comprobando la propiedad **caller** de cada uno de los **caller** anteriores hasta



que se llegue a uno nulo y así se conocería toda la cadena de llamadas que ha llevado hasta la actual.

También existe una propiedad de las funciones a la que se puede acceder desde su colección **arguments** que permite obtener una referencia a la función actual. Se trata de la propiedad **callee**. Su uso es poco frecuente porque lo que permite es obtener una referencia a la propia función que se está ejecutando, para poder llamarla u obtener acceso a su código. Esto puede ser útil en circunstancias muy particulares, como por ejemplo cuando se crea una **función anónima** que se va a usar con recursión (es decir, que se debe llamar a sí misma). Siendo anónima la única forma que existe de llamarla en la recursión es mediante **callee**.

### 3.1.6 EJECUTAR UNA FUNCIÓN EN LA CARGA DE UNA PÁGINA

Si se desea que se ejecute una función en JavaScript al cargarse la página Web, se escribirá en el evento onload de la etiqueta <body>. A continuación, se puede ver un ejemplo:

```
<html>
  <head>
    <title>Curso de JavaScript</title>
  </head>

  <body onload="alert('Hola');">
    <p>Tutorial de JavaScript</p>
  </body>
</html>
```

Aquí se puede ver otro ejemplo, en el que se pide la contraseña al usuario y se muestra si es válida (en caso de ser 33PPXX) o no:

```
<html>
  <head>
    <script>
      function prueba1() {
        let valor = prompt("Introduce la contraseña", "");
        if (valor == "HOLA123") {
          alert("La contraseña es correcta");
        } else {
          alert("Contraseña no válida: [" + valor + "]");
        }
      }
    </script>
  </head>
  <body onload="prueba1();">
    <p>Tutorial de JavaScript</p>
  </body>
</html>
```

En onload se pueden incluir varias sentencias JavaScript, separándolas con punto y coma.

## 3.2 MATRICES (ARRAYS)

Los arrays o matrices son objetos especializados de tipo colección que ayudan en el almacenamiento y obtención de datos gracias a claves indexadas. Su esquema de almacenamiento parte del índice 0 tal como en otros lenguajes de programación. Además, pueden crecer o comprimirse dinámicamente a medida que se añadan o eliminen elementos. Precisamente, por este carácter dinámico, el tamaño del array se determina por el índice más grande encontrado en aquel.

### 3.2.1 CREACIÓN DE ARRAYS

Los arrays se pueden declarar tanto usando un literal como con un constructor. Por ejemplo, si se hace mediante un constructor, se usa la palabra clave `Array` especificando los elementos como parámetros:

```
numArray = new Array(0, 1, 2, 3, 4, 5);  
numArray.length // devuelve 6
```

También se puede inicializar un Array vacío e ir llenándolo a posteriori:

```
var coche = new Array();  
coches[0]="Porsche";  
coches[1]="Renault";  
coches[2]="Seat";
```

Cuando se hace con literales, simplemente se crea el array y se inicializan sus elementos:

```
colores = ["rojo", "verde", "amarillo"];
```

La propiedad `length` de una matriz también es de escritura. Es posible, por tanto, redimensionarla poniendo una longitud diferente. El problema en este caso es que si se baja su tamaño, se perderán los elementos que había antes con el tamaño más grande, y si se aumenta, habrá elementos no definidos ya que nunca antes lo estuvieron.

### 3.2.2 TIPOS DE DATOS EN ARRAYS

Los elementos pueden ser de cualquier tipo; incluso se pueden tener elementos de tipos distintos en un mismo array. Si no está definido un elemento su valor será *undefined*. Ejemplos:

```
let a = ['Lunes', 'Martes', 2, 4, 6]  
console.log(a[0]) // imprime 'Lunes'  
console.log(a[4]) // imprime 6  
a[7] = 'Juan'      // ahora a = ['Lunes', 'Martes', 2, 4, 6, , , 'Juan']  
console.log(a[7])  // imprime 'Juan'  
console.log(a[6])  // imprime undefined  
console.log(a[10]) // imprime undefined
```

### 3.2.3 MANIPULACIÓN DE ELEMENTOS

Algunos ejemplos de trabajo con matrices:

```
var ejemplo=[0,1,2,3];  
alert(ejemplo.join("- - ")); // Construye una cadena con cada elemento separado por  
esos caracteres.
```

Concat permite concatenar otras matrices o más elementos. Un ejemplo:

```
var ejemplo=[0,1,2,3];  
var otraMatriz = ejemplo;  
otraMatriz[1]=4;  
alert(ejemplo[1]);
```

Es decir, cuando se asignan ambas matrices, lo que se modifica en una se traduce en la otra, ya que están asignadas por referencia. Si lo que se quiere es manipular una matriz copia de otra, se debe usar **slice**:

```
var otraMatriz = ejemplo.slice();
```

Otro ejemplo. Splice corta una matriz y asigna sus elementos cortados a otra diferente.

```
var ejemplo=[0,1,2,3,4,5,6,7];  
var res = ejemplo.splice(3,2); // Elemento inicial de índice 3 y número de elementos  
a quitar.
```

Por tanto, res = [3,4] y ejemplo [0,1,2,3,6,7]

### 3.2.4 ORDENACIÓN EN MATRICES

Se puede ordenar de forma inversa con **.reverse**. Lo que hace es ordenar los elementos de forma inversa. Además, es un método que modifica la matriz.

**sort()** ordena la matriz de menor a mayor. Ojo con esto, ya que ordena por cadenas, lo que es un problema en ordenaciones numéricas.

Javascript tiene comparadores para permitir cambiar nuestros criterios de ordenación:

```
var frutas= ["Naranja", "Pera", "Melocotón", "Albaricoque", "Uva"]  
function comparerCadenaLongitud(){  
    if (f1.length > f2.length)  
        return 1;  
    else if (f1.length < f2.length)  
        return -1;  
    else  
        return 0;  
}  
frutas.sort(comparerCadenaLongitud);
```

### 3.2.5 ARRAYS MULTIDIMENSIONALES

Hasta ahora se han visto las matrices unidimensionales, las cuales pueden asemejarse a lo que es un vector. No obstante, en términos matemáticos, habitualmente se utilizan matrices de dos dimensiones, lo que no quita que se puedan extender a más dimensiones. En programación se pueden considerar arrays multidimensionales a aquellos que tienen al menos dos dimensiones. Existen distintas maneras de inicializar una matriz dimensional. En este ejemplo se puede ver una forma muy curiosa de inicializado de este tipo de arrays:

```
var dimension1=["00", "01", "02"];  
var dimension2=["10", "11", "12"];  
var matriz2d = [dimension1, dimension2];  
alert(matriz2d[1][2]); // Mostraría 12
```

Un ejemplo de matriz multidimensional sería la escalonada:

```
var dimension1=["00", "01", "02"];  
var dimension2=["10", "11"];  
var matriz2d = [dimension1, dimension2];  
alert(matriz2d[1][1]); // Mostraría 11
```

En JavaScript no existen matrices asociativas, como en PHP, pero se puede simular su comportamiento.

```
var coches =[];  
coches["Alemanes"] = ["Audi", "Volkswagen", "Porsche"];  
coches["Franceses"] = ["Renault", "Citroen"];  
coches["Italianos"] = ["Fiat", "Alfa Romeo", "Ferrari"];  
alert(coches.Alemanes);
```

Aquí mostrará la colección de coches alemanes separadas por comas (también se puede usar `alert(coches[Alemanes])`). Esto no es una matriz asociativa porque su `length` de hecho vale 0, pero sí aprovecha la asignación de una propiedad a la variable `coches` (en el ejemplo, `Alemanes`).

### 3.2.6 PILAS Y COLAS

En JavaScript existen funciones **push** y **pop** para trabajar con pilas:

```
var ejemplo=[0,1,2,3,4,5,6,7];  
ejemplo.push(8);
```

Y se puede quitar con **pop()**, con lo que quedaría con un elemento menos y además lo devolvería la función.

También se pueden utilizar colas.

```
var ejemplo=[0,1,2,3,4,5,6,7];  
ejemplo.unshift(-2,-1); // Encola al principio (por este orden) -2, -1  
var x = ejemplo.shift(); // Obtengo el -2 y se lo quito además a la cola
```

### 3.2.7 OPERACIONES AGREGADAS: RECOLECCIÓN, FILTRADO Y REDUCCIÓN

Una búsqueda sencilla de un elemento se puede realizar con **indexOf** pasándole el parámetro de búsqueda. Devolverá el índice del primer elemento encontrado. Si no lo encuentra, devuelve -1. Para buscarlo desde el final: **lastIndexOf(elementoabuscar)**.

OJO: Porque en este caso da el índice empezando desde el final.

Además del argumento por defecto, a **indexOf** se le puede indicar el índice a partir del cual debe buscar.

```
function encontrarTodos(elto, matriz){ // Encuentra todos los elementos
    var encontrados =[];
    var pos = matriz.indexOf(elto);
    while (pos!=-1)
    {
        encontrados.push(pos);
        pos = matriz.indexOf(elto, ++pos);
    }
    return encontrados;
}
```

También se puede filtrar en la matriz según un criterio. Para eso se usa el método **filter**.

```
function esPar(n) { return n%2 ==0; }
var numerosPares = matriz.filter(esPar);
```

**Filter** devuelve un nuevo array con los elementos que cumplen determinada condición del array al que se aplica. Su parámetro es una función, habitualmente anónima, que va interactuando con los elementos del array. Esta función recibe como primer parámetro el elemento actual del array (sobre el que debe actuar). Opcionalmente puede tener como segundo parámetro su índice y como tercer parámetro el array completo. La función debe devolver **true** para los elementos que se incluirán en el array a devolver como resultado y **false** para el resto.

Con **.some** y el mismo argumento, devuelve **true** si al menos uno cumple con la condición. **.every** indica si todos y cada uno de los elementos cumplen con la función indicada.

**Map** mapea valores de una matriz con lo que hace una función:

```
function dobla (n) { return n*2; }
var res = matriz.map(dobla);
```

**Map** tampoco toca la matriz original.

La reducción es un proceso por el cual se puede convertir una lista de elementos X en un resultado Y. Se realiza mediante el método de programación funcional **.reduce** el cual devuelve un valor calculado a partir de los elementos del array. En este caso la función recibe como primer parámetro el valor calculado hasta ahora y el método tiene como primer parámetro la

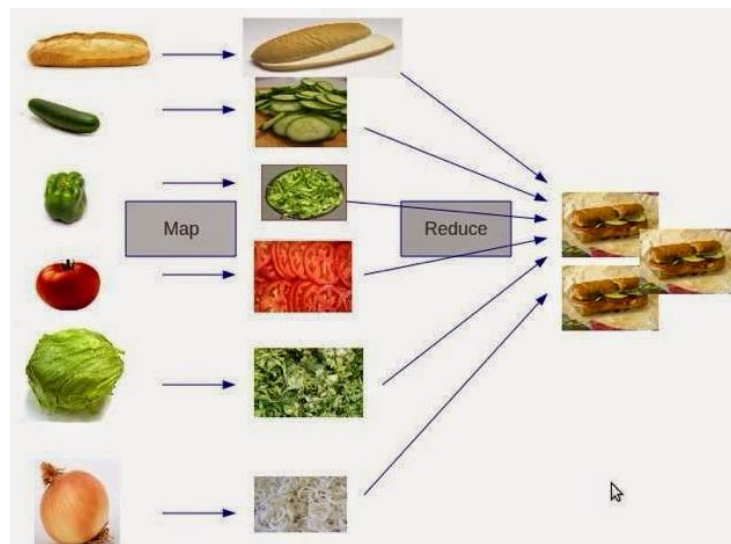
función y como segundo parámetro al valor calculado inicial (si no se indica será el primer elemento del array). Ejemplo: obtener la suma de unas calificaciones:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let sumaNotas = arrayNotas.reduce((total,nota)=> total + = nota, 0) // total = 35.35
// Se podría haber omitido el valor inicial 0 para total
let sumaNotas = arrayNotas.reduce((total,nota)=> total + = nota) // total = 35.35
```

Ejemplo: obtener la nota más alta:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let maxNota = arrayNotas.reduce((max,nota)=> nota > max ? nota : max) // max = 9.75
```

En el siguiente ejemplo gráfico se tiene un array de verduras al que se le aplica una función map para que las corte y al resultado se le aplica un reduce para que obtenga un valor (el sandwich) con todas ellas:



2. Ejemplo gráfico de Map y Reduce

### 3.2.8 USAR ARGUMENTS COMO UNA MATRIZ

La colección de argumentos de una función no es una matriz, sino que se comporta como si lo fuera. En realidad lo único que tiene en común con una matriz es la propiedad `length` para saber cuántos elementos tiene, y la posibilidad de acceder a la misma de manera indexada, es decir, indicando la posición de sus elementos: `arguments[0]` y similares. Además, es interesante que sus elementos, aparte de leerse se pueden también escribir, cambiando el valor de los argumentos que se le están pasando a la función.

La propiedad `callee` devuelve una referencia a la función en la que se está en ese momento. Si se quiere saber cuántos argumentos esperaba formalmente la función, es decir, cuántos parámetros están escritos en su definición y que no tienen que coincidir con los que realmente se le pasan y se obtienen con `arguments.length`, es posible hacerlo escribiendo este fragmento:

```
var numParamsOriginales = arguments.callee.length;
```

ya que `length` en una función representa el número de argumentos definidos para la misma.

Aunque muchas veces se ve **arguments** utilizada con el nombre de la función delante:

```
function sumaNumeros(){  
    var primero = sumaNumeros.arguments[0];  
}
```

en realidad, no es necesario ya que llega con escribir solamente **arguments** dentro de cualquier función para poder acceder a esta colección de parámetros:

```
function sumaNúmeros(){  
    var primero = arguments[0];  
}
```

Esto es especialmente útil y necesario en funciones anónimas de las cuales no se sabe su nombre, por ejemplo:

```
boton.addEventListener('click', function(){  
    var ev = arguments[0];  
}, true);
```

Además de que facilita el que se cambie el nombre de la función sin tener que acordarse de hacerlo también dentro, delante de **arguments**. Así que, **en general es mejor usarla de manera aislada**, poniendo simplemente **arguments** para acceder a los parámetros de una función.

Dado que no es una verdadera matriz sino lo que los anglosajones llaman un "array-like object", no se puede usar sobre ella los métodos de las matrices, como **pop** o **slice**, por ejemplo. No obstante, es muy fácil transformarla en una matriz auténtica si se quiere procesar, con un código como este:

```
var argumentos = Array.prototype.slice.call(arguments);
```

que introduce dentro de la variable argumentos una copia exacta de la colección arguments, sobre la que ya es posible operar como se haría con cualquier matriz normal. En cualquier caso, si el rendimiento es muy importante para la aplicación se debería evitar lo anterior ya que impide optimizaciones en el caso del intérprete de JavaScript de los navegadores. En ese caso quizá sea más eficiente, aunque no lo parezca, copiar la matriz recorriéndola en un bucle para copiar elemento a elemento. No obstante, en condiciones normales esto no debería preocupar en absoluto.

Otra cosa a tener en cuenta es que el objeto **arguments** contiene un "mapeado" entre un índice numérico (0, 1, 2, etc.) y cada uno de los argumentos de la función. Cuando, por ejemplo, se cambia el valor de un argumento usando por ejemplo **arguments[0] = "Otro valor"**, se cambia también el valor de dicho argumento, y si se accede a este por nombre se verá el cambio. Esto es así **menos en el modo estricto**. Si se indica **use strict** al principio el mapeo se mantiene, pero no afecta a los argumentos originales. Es una importante distinción. Se puede, por tanto, considerar mala práctica cambiar los valores directamente a través de este objeto.



### 3.2.9 ARRAYS ASOCIATIVOS (HASHTABLES)

Las HashTables son similares a los Arrays, pero con una gran diferencia. Véase el siguiente código:

```
<script>
  let myData = [];
  myData.push(["The Beatles", "248.3"]);
  myData.push(["Elvis Presley", "201.2"]);
  myData.push(["Michael Jackson", "155.8"]);

  let artist = "Elvis Presley"; //Recorremos el Array
  for (var i = 0; i < myData.length; i++) {
    if (myData[i][0] == artist) {
      alert("Ventas de " + artist + " es " + myData[i][1]);
    }
  }
</script>
```

MyData[0][0]	The Beatles	248.3	MyData[0][1]
MyData[1][0]	Elvis Presley	201.2	MyData[1][1]
MyData[2][0]	Michael Jackson	155.8	MyData[2][1]

3. Contenido de array por índice

El código anterior representa un array clásico referenciado mediante índices. Obsérvese el siguiente:

```
<script>
  let myData = new Object();
  myData["The Beatles"] = "248.3";
  myData["Elvis Presley"] = "201.2";
  myData["Michael Jackson"] = "155.8";

  for (x in myData) {
    //Recorremos el hashtable
    alert("Ventas de " + x + " es " + myData[x]);
  }
</script>
```

myData["The Beatles"]	248.3
myData["Elvis Presley"]	201.2
myData["Michael Jackson"]	155.8

4. Contenido de array asociativo

En resumen, mientras los Arrays se referencian mediante un índice, los HashTables se referencian mediante una clave.



### 3.2.10 OTROS TIPOS DE DATOS

En la anterior unidad se mencionaron algunos tipos que no fueron desarrollados en ella. A continuación, se verán en este apartado al tener una naturaleza de “colección” al igual que los arrays.

#### 3.2.10.1 Symbol

Aparece en ECMAScript 6. Equivalen a identificadores del estilo de un ID en HTML o que es una Primary Key en una base de datos relacional ya que son **únicos** y además también son **inmutables** así que no se puede modificar su contenido tras declararlo.

```
let a = Symbol();  
let b = Symbol();  
alert(a === b); // Devuelve false
```

Con la triple igualdad se comprueba que el tipo de dato y su contenido sean iguales. Ambos son de tipo Symbol, pero obviamente, cada vez que se declare uno nuevo será único y por tanto distinto. Incluso si se declarara como:

```
let a = Symbol("Oscar");  
let b = Symbol("Oscar");  
alert(a === b); // Devuelve false
```

También existen símbolos globales, pero hay que declararlos usando el método Symbol.for

```
let a = Symbol.for("Oscar");  
let b = Symbol.for("Oscar");  
alert(a === b); // Devuelve true
```

En la anterior unidad, se indicó que podrían servir para ser utilizados de forma similar a las enumeraciones en otros lenguajes. Como se ha podido ver, no es exactamente así ya que no tiene el mismo objetivo ni sintaxis. En cualquier caso, se puede simular el comportamiento de esas enumeraciones precisamente utilizando symbol. A continuación, un ejemplo:

```
const Directions = {  
  UP: Symbol("UP"),  
  DOWN: Symbol("DOWN"),  
  LEFT: Symbol("LEFT"),  
  RIGHT: Symbol("RIGHT"),  
};  
const currentDirection = Directions.UP;
```

En este código se usa Directions con propiedades que contienen Symbol para simular una enumeración. Además, cada una de ellas tiene un valor único gracias a Symbol.

Ahora podría usarse sin problema en un switch-case:

```
switch (currentDirection) {  
  case Directions.UP:  
    console.log("Going up!");  
    break;  
  case Directions.DOWN:  
    console.log("Going down!");  
    break;  
  case Directions.LEFT:  
    console.log("Going left!");  
    break;  
  case Directions.RIGHT:  
    console.log("Going right!");  
    break;  
  default:  
    console.log("Unknown direction");  
}
```

### 3.2.10.2 Map

En el apartado anterior se vieron los arrays asociativos. Map tiene una funcionalidad similar, ya que se pueden crear mapas clave/valor. A continuación, se muestran algunas de las operaciones que se pueden hacer con este nuevo tipo de objeto.

```
let coche = new Map();  
coche.set("marca", "chevrolet");  
coche.set("modelo", "aveo");  
coche.set("color", "blanco");  
alert(coche); //Map { marca: "chevrolet", modelo: "aveo", color: "blanco" }  
alert(coche.get("marca")); //chevrolet  
alert(coche.has("tv")); //false  
alert(coche.delete("color")); //true  
alert(coche.has("color")); //false  
coche.forEach(function(coche){ //Itera el Map  
  alert(coche); //chevrolet, aveo  
});  
for (let [clave, valor] of coche) { //Itera el Map  
  alert(clave + " - " + valor); //marca - chevrolet, modelo - aveo  
}  
for (let clave of coche.keys()) { //Itera las claves  
  alert(clave); //marca, modelo  
}  
for (var valor of coche.values()) { //Itera los valores  
  alert(valor); //chevrolet, aveo  
}  
coche.clear();  
alert(coche); //Map { }
```

### 3.2.10.3 Set

Set representa un conjunto y permite almacenar listas de valores únicos.

```
let dias = new Set();

dias.add("Lunes");
dias.add("Martes");
dias.add("Miércoles");
dias.add("Jueves");
dias.add("Viernes");
dias.add("Sábado");
dias.add("Domingo");
dias.add("Viernes");    //Ya existe. No produce ningún efecto

alert(dias); //Set [ "Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado",
"Domingo" ]
alert(dias.has("Sabado")); // true
alert(dias.size); //7
alert(dias.delete("Lunes")); //true
alert(dias.size); //6

dias.forEach(function(dia) {
    alert(dia); //Martes, Miércoles, Jueves, Viernes, Sábado, Domingo
});

dias.clear();
alert(dias.size); //0
```

Los métodos que incluye, son:

Método	Significado
<b>add</b>	Añade nuevos elementos al <b>Set</b>
<b>clear</b>	Elimina todos los elementos de un conjunto
<b>delete</b>	Elimina un elemento, devolviendo un booleano con el resultado de la operación
<b>entries</b>	Devuelve un objeto de tipo iterable que contiene a su vez un Array con los valores de cada elemento del Set
<b>forEach</b>	Recorre el Set
<b>has</b>	Comprueba que un valor, o el resultado de una expresión, se encuentra en el <b>Set</b> . Devuelve un valor <b>true</b> o <b>false</b>
<b>keys</b>	Idéntico al método <b>values</b>
<b>size</b>	Nos da el número de elementos
<b>values</b>	Devuelve un objeto de tipo iterable con el valor de cada uno de los elementos del <b>Set</b>

### **3.3 PROGRAMACIÓN ORIENTADA A OBJETOS**

A la hora de programar sistemas complejos una de las abstracciones más útiles que se han inventado es la de la programación orientada a objetos. Ésta ha dominado el mundo de la programación desde hace décadas y raro es el lenguaje que no soporta las partes más importantes de este paradigma.

La programación orientada a objetos trata de establecer una comparación entre el mundo físico (con objetos) y el del software (inmaterial), trabajando con entidades de código que representan objetos “reales”, y que nos permiten agrupar datos y funcionalidad para facilitarnos la escritura de código. Los objetos nos proporcionan muchas cosas, pero sobre todo orden y un modelo claro de trabajar con la información.

#### **3.3.1 CONCEPTOS BÁSICOS DE PROGRAMACIÓN ORIENTADA A OBJETOS**

En la realidad pocas cosas están construidas desde cero, sin tener partes constituyentes. Generalmente todos los objetos que se utilizan o tocan a diario están constituidos a su vez por otros objetos más pequeños. Estos tienen una funcionalidad determinada e independiente del conjunto al que pertenecen, aunque luego se vea potenciada por la interacción sobre los otros elementos que forman parte de aquel.

Por ejemplo, un ordenador está constituido por multitud de piezas u objetos que están especializados en unas funciones muy determinadas. Cada pieza por separado no es muy útil, pero todas juntas hacen que funcione el sistema. Cada objeto tiene unas características muy determinadas y unas funcionalidades concretas que lo identifican. La tarjeta de vídeo tiene una cierta cantidad de memoria, una velocidad de proceso, etc. y dispone de funciones para transformar vectores, o para calcular superficies 3D complejas. Estas piezas se pueden reutilizar, no sólo porque una tarjeta gráfica de un ordenador se puede colocar en otro, sino porque el propio concepto de tarjeta es lo que se reutiliza al fabricar dos tarjetas iguales. Lo primero que es preciso conocer sobre programación orientada a objetos es la diferencia que existe entre un objeto y una clase.

Una clase define aquellas propiedades y métodos que conceptualmente determinan a un ente o cosa. Se trata de algo genérico, no particular. Es decir, en un arcade de marcianos, no se define un marcianito en particular, sino que se define una clase “Marcianito” que determina cómo debe ser cualquier marcianito, qué características tienen todos ellos (color, posición, disparos disponibles...) y qué cosas son capaces de hacer (moverse, disparar, pintarse en pantalla, desaparecer...).

Cuando una clase adquiere entidad física con valores concretos para sus propiedades se dice que se está ante un objeto o ante un ejemplar de la clase. Así, un marcianito concreto que se haya creado, con un determinado color, una posición, etc. es un objeto o una instancia de la clase “Marcianito”. Es muy importante tener clara esta distinción.

Los tres pilares básicos de la Programación Orientada a Objetos son:

- **Encapsulación:** es el empaquetamiento de datos y funciones dentro de un único elemento (una clase), accesibles siempre a través de éste. Por ejemplo, para acceder a las propiedades de un objeto solamente es posible hacerlo a través de dicho objeto. De esta forma el objeto y sus datos y acciones forman un todo. Está también relacionado con la ocultación de datos que se refiere a que los datos internos de un objeto no son accesibles desde el exterior y sólo pueden tocarse a través de una interfaz determinada que se haya definido. Por ejemplo, se puede almacenar internamente el valor de la estatura de una clase Persona en centímetros, pero exponerla al exterior a través de una propiedad que devuelve metros, y a la hora de escribirlo que solamente se pueda hacer a través de una determinada función que valida múltiples condiciones y la convierte a centímetros antes de almacenarla de nuevo.
- **Herencia:** cuando existen diferentes variantes de una clase, en lugar de tener que definir una clase nueva que replique los datos y la funcionalidad de la primera, es más fácil crear una nueva clase que "herede" de la clase pre-existente todas sus características. Por ejemplo, si se tiene un objeto Persona que tiene información como nombre, apellidos, edad, dirección, etc. y ahora hay que manejar otro tipo de personas como Clientes, Proveedores, Familiares, etc. los cuales tienen los mismos datos pero que a mayores añaden otros y redefinen la manera de usar algunos otros, se pueden crear clases derivadas de la clase Persona que heredarán toda su funcionalidad y permitirán también particularizarla. Este mecanismo es muy potente y nos permite escribir código más conciso, fácil de gestionar y menos propenso a errores.
- **Polimorfismo:** cuando varios objetos heredan de la misma clase base, se podrán utilizar de la misma manera considerando que son de dicha clase base común. Por ejemplo, si se tienen las clases especializadas ya mencionadas: Cliente, Proveedor, Familiar... y todas heredan de Persona, es posible hacer uso de ellas en una misma función, sin importar su tipo concreto, ya que todas ellas son Personas también. En lenguajes más avanzados que JavaScript que definen el concepto de Interfaz (un contrato que define cómo interactuar con una clase), el polimorfismo se refiere también al uso de interfaces. En JavaScript no existen, aunque se puede simular su funcionalidad.

### 3.3.2 INICIALIZACIÓN DE OBJETOS EN JAVASCRIPT

A continuación, se irá viendo cómo JavaScript soporta las principales características del paradigma de programación orientada a objetos, aunque tenga sus propias particularidades al respecto.

En la concepción clásica de JavaScript, un objeto se define como:

```
let miObjeto = new Object();
```

Las propiedades en un objeto JavaScript se pueden definir dinámicamente. Por ejemplo, si se hace `miObjeto.name="nombre"`; se ha creado su propiedad y puede usarse cuando se quiera.

Para acceder a una propiedad, se tiene el sistema convencional con la sintaxis `objeto.propiedad`, pero además, en JavaScript también se pueden acceder como si fueran elementos de un array, es decir, `objeto[propiedad]`, de ahí que con objetos también se pueda usar el `for...in` que se vio anteriormente.

También se pueden crear métodos mediante `function`:

```
miObjeto.disparar = function() {  
    this.disparos--;  
    alert(this.name + "ha disparado.");  
}
```

Obviamente, este sistema no es muy útil. No deja de ser instanciar un tipo de objeto genérico `Object` y añadirle dinámicamente nuevos métodos y variables miembro. Hay muchas otras formas de hacerlo y, además, en ese sentido el lenguaje ha mejorado con las últimas versiones.

Una opción mejor es crear los objetos usando una función constructora.

```
let Persona = function (nombre, edad) {  
    //Constructor de La Clase  
    this.nombre = nombre; //Propiedad  
    this.edad = edad; //Propiedad  
};
```

Otra forma mediante constructor:

```
function Persona(nombre, edad) {  
    //Constructor de La Clase  
    this.nombre = nombre; //Propiedad  
    this.edad = edad; //Propiedad  
}
```

También se pueden definir objetos utilizando la sintaxis JSON.

```
var miobjeto = {  
    name : "Invasor del espacio #1",  
    color : "Azul",  
    x : 100,  
    y : 20,  
    disparos : 30,  
    disparar : function(){  
        this.disparos--;  
        alert(this.name + "ha disparado");  
    }  
};
```

### 3.3.3 CLASES JAVASCRIPT

Las clases de JavaScript, introducidas en ECMAScript 2015, son una mejora sintáctica sobre la herencia basada en prototipos de JavaScript. Proveen una sintaxis mucho más clara y simple para crear objetos y lidiar con la herencia. Como las expresiones de funciones y declaraciones

de funciones, la sintaxis de una clase tiene dos componentes: expresiones de clases y declaraciones de clases.

Una manera de definir una clase es mediante una declaración de clase. Para declarar una clase, se utiliza la palabra reservada **class** y un nombre para la clase "Marcianito".

```
class Marcianito {  
  constructor(nombre, color) {  
    this.nombre = nombre;  
    this.color = color;  
  }  
}
```

En el siguiente código se puede ver una clase más completa compuesta con distintos elementos posibles dentro de su sintaxis:

```
class Saludador {  
  nombre = "Sin nombre";  
  static palabraSaludo = "Hola";  
  
  constructor(nombre){  
    this.nombre = nombre;  
  }  
  
  saludo(){  
    return `${Saludador.palabraSaludo}, soy ${this.nombre}`;  
  }  
}
```

Como puede verse en el código, se tiene una propiedad nombre, una variable estática (dependiente únicamente de la clase y no de sus objetos) llamada palabraSaludo, un constructor y un método saludo. Todos los elementos son públicos en este caso, con lo que pueden accederse por cualquier objeto de esta clase.

Una particularidad de JavaScript es que pueden añadirse nuevas propiedades a los objetos, aunque estas no hayan sido declaradas en la definición de la clase. Por ejemplo:

```
miSaludador = new Saludador("Pedro");  
miSaludador.nuevaPropiedad = "Tengo una nueva propiedad; parezco un superalimento";
```

Si se quiere eliminar, basta con llamar a delete:

```
delete(miSaludador.nuevaPropiedad);
```

### 3.3.3.1 Constructores de clases

Anteriormente se ha visto que para crear un nuevo objeto de una clase propia de JavaScript – como una fecha – se empleaba la palabra clave new seguida del nombre del tipo de objeto, por ejemplo:

```
var Fecha = new Date(2024, 6, 27);
```

Así es como se deberían poder crear los objetos, especificando el tipo correcto para crear un verdadero objeto de esa clase. La cláusula `new` en JavaScript se utiliza para crear nuevas instancias de clases. Lo que hace `new` es llamar a una función especial que poseen todas las clases, denominada constructor, y que como en todos los lenguajes orientados a objetos, tiene el mismo nombre que la clase que se quiere instanciar.

Como se ve en la línea anterior, un constructor toma opcionalmente una serie de parámetros para inicializar el objeto que se está creando. En el ejemplo de la línea anterior se ha creado un objeto de la clase `Date`, especializada en albergar y trabajar con fechas, que tiene como valores de creación un determinado día, mes y año. Este objeto, gracias a lo que determina su clase, maneja una determinada información (en este caso la fecha y la hora) y tiene métodos que le permiten hacer cosas con ella, como descomponerla en sus partes, transformarla en texto y todo lo que hemos visto que puede hacer esta clase.

Además, es posible determinar si la variable asignada pertenece a la clase `Date` fácilmente, por ejemplo, si se comprueba su tipo con `typeof`.

En el código anterior con el que se definía la clase `Marcianito`, el constructor se incluía en la palabra clave con el mismo nombre (resaltado en negrita):

```
class Marcianito {  
  constructor(nombre, color) {  
    this.nombre = nombre;  
    this.color = color;  
  }  
}
```

Con esto ya se puede hacer lo siguiente:

```
let marcianito1 = new Marcianito("ALF", "Marrón");
```

Aunque se haya creado un objeto a partir de `new`, cuando se haga un `typeof` sobre el objeto, va a seguir dando como tipo `"object"`.

Lo que sí se crea al hacer un `new` es una referencia llamada `"constructor"` en forma de propiedad que devuelve una referencia a la propia función constructora. De esta forma se puede saber tranquilamente si un objeto es de un tipo o no:

```
if (marcianito1.constructor == Marcianito) // devolverá true si el objeto es del tipo  
Marcianito
```

Pero en realidad la mejor opción es utilizar `instanceof` de esta forma:

```
if (marcianito1 instanceof Marcianito) //devuelve true si mi objeto es del tipo  
Marcianito
```

Además, debido a que todos los objetos heredan de **object**, también dará `true` un `instanceof` sobre **object**.



### 3.3.3.2 Métodos estáticos

JavaScript también dispone de métodos estáticos (static). Estos son llamados sin instanciar su clase y no pueden ser llamados mediante un objeto. Por lo general, este tipo de métodos se reservan a clases que coleccionan utilidades y que no se espera de ellas que sean instanciadas.

```
class Tools {  
  static strToURL(str) {  
    return encodeURIComponent(str).replace(/%20/g, "+");  
  }  
}  
Tools.strToURL("La donna e mobile"); // "La+donna+e+mobile"  
let toolkit = new Tools();  
toolkit.strToURL("La donna e mobile"); //ERROR. EL Método no existe
```

### 3.3.3.3 Métodos públicos y privados

Cualquier otro método que complementa una, será un método público o privado, ejecutable únicamente desde sus instancias (directas o heredadas). Ejemplo de método público:

```
class Rectangle {  
  constructor(height = 0, width = 0) {  
    this._height = height;  
    this._width = width;  
  }  
  area() {  
    //Método Público (FUERA)  
    return this._height * this._width;  
  }  
}  
let rectangleOne = new Rectangle(10, 20);  
rectangleOne.area();
```

Ejemplo de método privado:

```
class Rectangle {  
  constructor(height = 0, width = 0) {  
    this._height = height;  
    this._width = width;  
    let devuelveAlto = function () { //Método privado (DENTRO)  
      return height;  
    };  
    this.queAlto = function () { //Método privilegiado y publico  
      return devuelveAlto();  
    };  
  }  
}  
let rectangleOne = new Rectangle(10, 20);  
rectangleOne.queAlto();
```

### 3.3.3.4 Variables públicas y privadas

En algún momento, se puede necesitar crear variables para el trabajo interno con las propiedades y métodos de la clase. Si las variables son públicas, podrán ser accedidas desde el objeto, no así las privadas.

```
class Rectangle {
  constructor(height = 0, width = 0) {
    this._height = height;
    this._width = width;

    this.var2 = 12; //Variable pública (DENTRO)

    let var1; //Variable privada (DENTRO)
    this._var1 = var1;
  }
  get var1() {
    //getter
    return this._var1;
  }
  set var1(valor) {
    //setter
    this._var1 = valor;
  }
}
let rectangleOne = new Rectangle(10, 20);
alert(rectangleOne.var2);
alert(miobjeto.var1); //undefined
miobjeto.var1 = 22;
```

### 3.3.4 HERENCIA

En las clases JavaScript también se disponen de mecanismos de herencia. Para ello, se usa la palabra clave `extends`. Véase el siguiente código:

```
class Usuario extends Saludador{
  apellido = "";

  constructor(nombre, apellido){
    super(nombre);
    this.apellido = apellido;
  }

  saludo(){
    return `${super.saludo()} ${this.apellido}`;
  }
}
```

En este ejemplo, se crea un nuevo constructor y se llama a métodos de la clase base mediante la palabra clave **super**. En el siguiente código puede verse cómo se crea un nuevo objeto de esta clase hija y además se modifica la variable estática de la clase base mostrando cómo se propaga el cambio entre sus descendientes.

```
const usuario = new Usuario("Juan Diego", "Bueno");  
console.log(usuario.saludo());  
Saludador.palabraSaludo = "Saludos";  
console.log(usuario.saludo());
```

### 3.3.5 LA PROPIEDAD PROTOTYPE

Los objetos tienen una propiedad **prototype** que almacena una referencia al prototipo que define un objeto. Modificando **prototype**, se cambia la definición misma de un objeto.

Este ejemplo muestra cómo modificar la definición del objeto **Array** de forma que, a partir de este momento, todos los objetos de este tipo tengan un método **average** para obtener la media de los elementos:

```
Array.prototype.average = function(){  
    var iTotal=0;  
    for (var i=0; i<this.length; i++){  
        iTotal+=this[i];  
    }  
    return iTotal / this.length;  
}
```

Una vez extendido el objeto **Array**, se puede usar el nuevo método de forma simple. Si este script se ejecuta varias veces, en todas ellas se modifica **prototype**, redefiniendo **average** (aunque no cambie). Por esa razón, siempre es conveniente verificar si ya existe:

```
if (Array.prototype.average == undefined) {...}
```

### 3.3.6 DISEÑO MODULAR

En las nuevas versiones de JavaScript se permite un diseño de aplicaciones mucho más modular, no solo basándose en la forma clásica de integración de un archivo externo.

Para trabajar con módulos se tienen a disposición las siguientes palabras clave:

<b>export</b>	Exporta uno o varios elementos (variables, funciones, clases...) del fichero actual
<b>import</b>	Importa uno o varios elementos (variables, funciones, clases...) desde otro fichero <b>.js</b>

Mediante la palabra clave `export` se crea un objeto (módulo de exportación) que contendrá una o varias propiedades. En estas propiedades se pueden guardar variables, funciones o clases (a partir de ahora, elementos). Si dicho módulo ya existe, es posible ir añadiendo más propiedades. Por otro lado, con la palabra clave `import` se pueden leer dichos módulos de otros ficheros y utilizar sus propiedades en el código. Existen varias formas de exportar código mediante la palabra clave `export`:

<code>export { name };</code>	Añade el elemento <b>name</b> al módulo de exportación.
<code>export { n1, n2, n3... };</code>	Añade los elementos indicados ( <b>n1, n2, n3, ...</b> ) al módulo de exportación.
<code>export * from './file.js';</code>	Añade todos los elementos del módulo <b>file.js</b> al módulo de exportación.
<code>export declaration;</code>	Declara una variable, función o clase y la añade al módulo de exportación.
<code>export default declaration;</code>	Declara una función o clase y la añade al módulo de exportación.

Si por un lado está `export`, la palabra clave `import` es su complementaria. Con ella es posible cargar un módulo de exportación de otro fichero JavaScript, con todos los elementos exportados que contiene. Existen varias formas de importar código utilizando `import`:

<code>import nombre from './file.js';</code>	Importa sólo el elemento por defecto de <b>file.js</b> en <b>nombre</b> .
<code>import { nombre } from './file.js';</code>	Importa sólo el elemento <b>nombre</b> de <b>file.js</b> .
<code>import { n1, n2.. } from './file.js';</code>	Importa los elementos indicados desde <b>file.js</b> .
<code>import * as obj from './file.js';</code>	Importa todos los elementos de <b>file.js</b> en el objeto <b>obj</b> .
<code>import './file.js';</code>	No importa elementos, pero ejecuta el código de <b>file.js</b> .

Supóngase un archivo **modulo-export.js** con el siguiente código:

```
export const hola = () => "Hola";
export const adios = () => "Adios";

export default class MiClase{
  adios() {
    return "Adios";
  }
}
```

En este módulo se exportan dos módulos muy sencillos, hola y adios, y una clase por defecto. En cada módulo solo se puede exportar un elemento por defecto. ¿Qué significa esto? En el siguiente código del archivo modulo.js se pueden ver ejemplos de uso:

```
import MiClaseImportada, { hola } from "./modulo-export";
import * as todo from "./modulo-export";

console.log(hola());
const miObjeto = new MiClaseImportada();
console.log(miObjeto.adios());

console.log(todo.hola());
const miObjeto2 = new todo.default();
console.log(miObjeto2.adios());
```

Es interesante ver, primeramente, cómo se importan los distintos elementos. **MiClaseImportada** representa a **MiClase** ya que es el elemento por defecto. Por otra parte, en la misma línea de código, se importa el método **hola**. En la siguiente, se realiza una importación de todos los elementos de modulo-export.js con el alias **todo**. Posteriormente, se ven distintos usos de los elementos: se llama al método **hola()** de forma explícita, se crea un objeto a partir de ese alias **MiClaseImportada** y, por último, se usa la palabra clave default, la cual representa ese elemento por defecto.

Para poder importar el módulo modulo.js para su uso en una página HTML, se necesita incluir un atributo especial en la línea de importación. El código quedaría así:

```
<script type="module" src="modulo.js"></script>
```

Nota: Este tipo de importación no funcionará si se está probando con archivos locales debido a las restricciones de seguridad de los navegadores con respecto al uso compartido de recursos entre orígenes (CORS - Cross-Origin Resource Sharing). Si se quiere ensayar este mecanismo de este modo, se debe montar una estructura sobre un servidor web ya existente o crear uno ad-hoc (usando Python HTTP Server).

Por ejemplo, el archivo **modulos.js** podría quedar como sigue:

```
import {PI} from './constantes.js' //Importa una constante definida en constantes.js
console.log(PI);
```

Si se utiliza el siguiente código:

```
import {PI,usuario} from './constantes.js';
console.log(PI);
console.log(usuario);
```

teniendo en cuenta que constantes.js contiene el siguiente:

```
export const PI = Math.PI();
let usuario = 'Carlos';
```

dado que usuario no aparece como exportada, el sistema dará un error al mostrar el usuario por consola.

### 3.3.7 DESESTRUCTURACIÓN

La desestructuración en JavaScript es un proceso que permite extraer valores de matrices o propiedades de objetos sobre una variable de ese tipo. A continuación, se muestra un ejemplo de uso. Supóngase la siguiente clase usuario definida en formato JSON:

```
const usuario = {  
  nombre: "Juan Diego",  
  apellido: "Bueno",  
  cursos: ["Desarrollo Web en entorno cliente", "Desarrollo de interfaces", "Puesta  
en producción segura"]  
}
```

Ahora se puede definir una función **getNombreCompleto** con la siguiente sintaxis:

```
const getNombreCompleto = ({nombre, apellido}) => `${nombre} ${apellido}`;  
console.log("getNombreCompleto: ", getNombreCompleto(usuario));
```

Este ejemplo ilustra, no solo la creación de la función, sino también el uso de string literals y de funciones arrow o lambda. Lo que se hace en esta función es recoger el parámetro pasado y extraer los atributos llamados nombre y apellido (siempre, claro está, que sea un objeto que tenga estas propiedades). Esta función devuelve la concatenación de ambos campos para obtener un nombre completo. Pero no es estrictamente necesario definir una función para obtener los valores de esas propiedades, también podría hacerse lo siguiente:

```
const {nombre, apellido} = usuario;  
console.log(nombre);
```

Con esto, del tirón se consigue asignar dos nombres de variables a las correspondientes con las propiedades del usuario, lo cual puede resultar útil en algún momento. Pero también se podría dar otro nombre a cualquiera de esas variables extraídas si así se deseara:

```
const {nombre: nuevoNombre} = usuario;  
console.log(nuevoNombre);
```

En el siguiente ejemplo, se va a ver cómo desestructurar un objeto en otro nuevo:

```
const nuevoUsuario = {  
  ...usuario,  
  nombre: "Lorena"  
};
```

Los tres puntos antes de usuario implican que lo que debe coger la expresión es todas las propiedades del objeto, es decir, como si se hubieran cogido todas las propiedades separadas por comas tal como se definió usuario en la primera porción de código. Posteriormente, se modifica la propiedad nombre dejando un objeto nuevo con el cambio dado.

De un modo similar, se pueden desestructurar los datos de un array. Véase el siguiente ejemplo:

```
const [primerCurso, ...otrosCursos] = nuevoUsuario.cursos;  
console.log("Primer curso: ", primerCurso);
```

```
console.log("Otros cursos: ", otrosCursos);
```

Con esta porción de código separamos el primer curso (primerCurso) del resto (...otrosCursos) apoyándose de nuevo en los tres puntos.

Otra de las utilidades de la desestructuración es la asignación de propiedades de objetos de forma compacta tal como puede verse a continuación:

```
const nuevoObjeto = {nuevoNombre, primerCurso};  
console.log("nuevoObjeto: ", nuevoObjeto);
```

En este caso se crea un nuevo objeto con los valores de las variables definidas anteriormente.

Por último, otra variante interesante de la desestructuración es la que permite hacerlo con los parámetros de una función:

```
const suma = (...nums) =>  
  nums.reduce((total, num) => total + num, 0);  
console.log("suma:", suma(1, 2, 3, 4, 5, 6, 7, 8, 9 ))
```

La función suma recoge como argumento un array con todos sus parámetros. En la expresión lambda se recorre dicho array para ir sumando cada uno de sus elementos.

### 3.4 PATRONES DE DISEÑO

Un patrón de diseño es una técnica de diseño que puede ayudar a resolver un problema concreto. Principalmente persigue que para un supuesto concreto se le dé una solución que ya ha sido probada anteriormente. Por ejemplo, se pueden usar para construir nuevos objetos, regular su comportamiento o la relación entre distintos objetos. A grandes rasgos, los patrones de diseño pueden ser:

- Creacionales. Patrones de creación de nuevos objetos.
- Estructurales. Patrones de relación entre clases y objetos y su estructura.
- Comportamiento. Patrones que se encargan del comportamiento de un objeto y su posible cambio.

En este apartado no se va a profundizar en todos los patrones de diseño conocidos ya que se podría extender a un curso completo. Únicamente, se verán algunos de los que pueden ser más interesantes aplicados a JavaScript.

#### 3.4.1 SINGLETON

El propósito de singleton es que solo haya únicamente un objeto de una clase concreta. En el siguiente código se ilustra una posible implementación de Singleton:

```
class Singleton{
  getInstance() {
    return Singleton.instance;
  }
  constructor(){
    if (Singleton.instance)
      return Singleton.instance;
    else
      Singleton.instance = this;
  }
}
```

¿En qué consiste este código? Bien, el nombre de la clase en este caso es **Singleton** pero únicamente es un nombre de ejemplo; en una aplicación práctica sería una clase de la que se eligiera tener exclusivamente un solo objeto. El constructor de esta clase primeramente evalúa si ya hay un objeto de ese tipo (**instance**); si no lo hay, lo crea y si lo hay, simplemente lo devuelve. Para obtener ese objeto único, existe un método llamado **getInstance()** que lo devuelve y que sería el que se utilizaría en el resto de código.

Supóngase que se necesita un objeto para guardar una conexión a una base de datos y esta debe ser única para todo un sistema. Mediante este patrón se garantiza que no habrá más de un objeto encargado de dicha conexión. Otra posible aplicación es un logger que registre los eventos más relevantes sucedidos en el sistema. También se podría utilizar en una clase que registre propiedades de configuración de un sistema.



### 3.4.2 FACTORÍA

Factory o factoría es un patrón creacional que se puede usar para construir objetos. El siguiente ejemplo de código ilustra cómo se puede crear un objeto mediante una función **crearObjeto** que devuelve ese ejemplar creado.

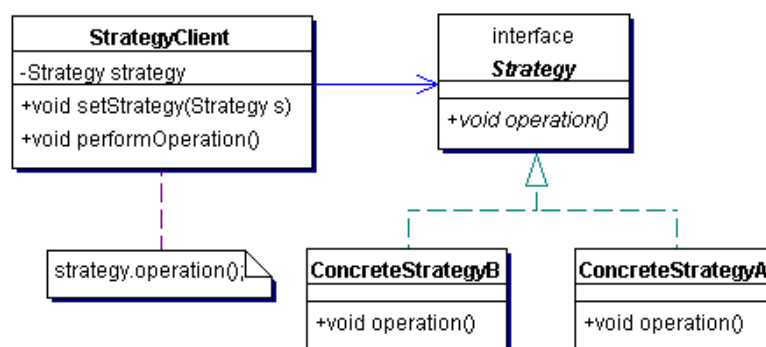
```
function crearObjeto(nom, col, posX, posY, disparosIniciales) {  
    let miobjeto = {  
        nombre: nom,  
        color: col,  
        x: posX,  
        y: posY,  
        disparos: disparosIniciales,  
        disparar: function () {  
            this.disparos--;  
            alert(this.nombre + " ha disparado");  
        },  
    };  
    return miobjeto;  
}  
  
let miObjetoNuevo1 = crearObjeto("invasor 1", "Azul", 0,0,30);  
let miObjetoNuevo2 = crearObjeto("invasor 2", "Verde", 100,30,50);
```

### 3.4.3 STRATEGY (ESTRATEGIA)

**Strategy** es un patrón de comportamiento que permite tener comportamientos distintos en un objeto y añadirle nuevos sin modificar el contexto inicial. Este patrón se puede usar cuando:

- Se tienen distintas formas de ejecutar una acción
- Es posible que no se sepa qué enfoque utilizar hasta el momento de la ejecución
- Se quiera añadir de forma fácil otras nuevas formas de ejecutar una acción
- Se quiera seguir teniendo el código mantenible después de añadir comportamientos

En las implementaciones de lenguajes como Java o C#, más orientados a objetos, se utilizan interfaces para algunos de los artefactos del sistema. Dado que en JavaScript no existen como tal, no se utilizará en el siguiente ejemplo. Una posible implementación que se podría utilizar en Java se puede ver en el siguiente diagrama de clases:



5. Diagrama de clases de patrón Strategy en Java

Prácticamente se centra en tener una clase contexto con una estrategia y que esta contenga una operación. Se trata de crear una nueva clase que tenga un comportamiento, se ajusta al contexto y este ejecutará la acción del contexto que tiene la estrategia.

En el ejemplo, se va a realizar un cliente para una venta de forma que, dependiendo del tipo de esta, realice una estrategia de cálculo distinta.

```
class VentaClient{
  constructor(strategy){
    this.strategy = strategy;
  }
  SetStrategy(strategy){
    this.strategy = strategy;
  }
  Calcular(cantidad){
    return this.strategy.Calcular(importe);
  }
}
```

Como puede verse en el código, la clase **VentaCliente (StrategyClient)** recibe una estrategia en el constructor además de poder pasársela como propiedad mediante **SetStrategy**. El equivalente a **performOperation** del diagrama de clases es, en este caso, el método **Calcular**, que no hace sino aplicar el método de cálculo según la estrategia proporcionada dado un importe. A partir de aquí, no hay más que definir qué estrategia o estrategias se desean implementar. La más sencilla es una venta regular, en la que simplemente se da el importe y el impuesto a aplicar devolviendo el importe total al llamar a **Calcular**.

```
class VentaRegularStrategy{
  constructor(impuesto){
    this.impuesto = impuesto;
  }
  Calcular(importe){
    return importe + (importe * this.impuesto);
  }
}
```

Aquí ya se puede ver una de las ventajas de este patrón: se separa la implementación del método en otra clase, lo que contribuye al desacoplamiento. Pero la mayor de las ventajas es que se pueden implementar distintas estrategias de cálculo sin prácticamente tocar la clase, de forma que se pueden “enchufar”; basta con que implementen un método **Calcular**. En lenguajes puramente orientados a objetos como Java, se garantiza que se cumpla este último requisito haciendo que las estrategias implementen un interfaz, que es lo que al fin y al cabo se pasa al objeto cliente. Dado que JavaScript no implementa interfaces, simplemente hay que cuidarse de que se implemente el método dejando esta operación al buen hacer de quien diseña el código.

Para probar este patrón, lo primero que se hace es crear una estrategia y luego “inyectársela” al cliente. Si se llama al método calcular, aplicará la estrategia de la venta regular. Además, con SetStrategy se puede cambiar en cualquier momento si se desea.

```
const ventaRegularStrategy = new VentaRegularStrategy(0.21);  
const venta = new VentaClient(ventaRegularStrategy);  
console.log(venta.Calcular(10));
```

Otro ejemplo de estrategia podría ser aplicar un descuento sobre el importe después de impuestos. Este ejemplo mostrará 12.1:

```
class VentaDescuentoStrategy{  
  constructor(impuesto, descuento){  
    this.impuesto = impuesto;  
    this.descuento = descuento;  
  }  
  Calcular(importe){  
    return importe + (importe * this.impuesto) - this.descuento;  
  }  
}
```

Aquí puede verse un ejemplo de implementación. Una vez aplicado el descuento, en este caso se muestra 10.21.

```
class VentaDescuentoStrategy{  
  constructor(impuesto, descuento){  
    this.impuesto = impuesto;  
    this.descuento = descuento;  
  }  
  Calcular(importe){  
    return importe + (importe * this.impuesto) - this.descuento;  
  }  
}  
  
const ventaDescuentoStrategy = new VentaDescuentoStrategy(0.21, 2);  
const ventaConDescuento = new VentaClient(ventaDescuentoStrategy);  
console.log(ventaConDescuento.Calcular(10));
```

Por último, se puede aplicar una estrategia de cambio de divisas para una exportación. En el ejemplo, la venta se va a realizar a Paraguay y es una muestra de código en la que el constructor de la estrategia no recoge ningún parámetro, algo que se ha hecho hasta ahora y no es para nada obligatorio ya que en la propia estrategia se recoge la tasa de cambio.

El siguiente ejemplo devolverá 85.000 guaraníes, el equivalente a 10 euros.

```
class VentaParaguayStrategy{
  Calcular(importe){
    return importe * this.getCambioGuaranies();
  }
  getCambioGuaranies(){
    return 8500;
  }
}

const ventaParaguayStrategy = new VentaParaguayStrategy();
const ventaParaguay = new VentaClient(ventaParaguayStrategy);
console.log(ventaParaguay.Calcular(10));
```

Este patrón y el anterior no son más que una pequeña muestra de la potencia y utilidad de estos patrones. La primera recopilación de este tipo de patrones se realizó en 1994 por el conocido como Gang of Four (GoF) aprovechando la emergencia de lenguajes orientados a objetos como Java y C++ por esos años.

## ÍNDICE DE FIGURAS

1. Mensaje en consola con estilo CSS .....	4
2. Ejemplo gráfico de Map y Reduce .....	13
3. Contenido de array por índice.....	15
4. Contenido de array asociativo .....	15
5. Diagrama de clases de patrón Strategy en Java .....	32

## BIBLIOGRAFÍA – WEBGRAFÍA

Moreno Pérez, J.C. (2020) *Desarrollo web en entorno cliente*. Editorial Síntesis. 1ª Edición

Heilmann, C. (2021) *Beyond Console.log() – Level up Your Debugging Skills*  
<https://www.sitepoint.com/beyond-console-log-level-up-your-debugging-skills/>

De León, H. (2022) *Patrones de Diseño en JavaScript y TypeScript*  
<https://www.udemy.com/course/patrones-de-diseno-en-javascript-y-typescript/>