Contenidos a Trabajar

1. Operadores
2. Condicionales
3. Ciclos y Bucles
4. Funciones
5. Callbacks
6. Arrays y sus métodos

Agencia de Aprendizaje a lo largo de la vida

Operadores

Al desarrollar programas en cualquier lenguaje se utilizan los operadores, que sirven para hacer los cálculos y operaciones necesarios para llevar a cabo tus objetivos. Hasta el menor de los programas imaginables necesita de los operadores para realizar cosas, ya que un programa que no realizará operaciones, sólo se limitaría a hacer siempre lo mismo.

Es el resultado de las operaciones lo que hace que un programa varíe su comportamiento según los datos que tenga para trabajar y nos ofrezca resultados que sean relevantes para el usuario que lo utilice. Existen operaciones más sencillas o complejas, que se pueden realizar con operandos de distintos tipos, como números o textos.

Operadores aritméticos

Son los utilizados para la realización de operaciones matemáticas simples como la suma, resta o multiplicación. En javascript son los siguientes:

```
+   //  Suma de dos valores
-   //  Resta de dos valores, también invierte el signo de un número.
*   //  Multiplicación de dos valores
/   //  División de dos valores
%   //  El resto de la división de dos números.
++  //  Incremento en una unidad, se utiliza con un solo operando
--  //  Decremento en una unidad, utilizado con un solo operando
```

Ejemplos

```
unidades = 100 + 12; // suma
precio = 250 - 39; // resta
factura = precio * unidades // multiplicación
descuento = precio / off // división
resto = factura % 3 // resto
precio++ // incrementa en una unidad el precio.
precio-- // decrementa en una unidad el precio.
```

Operadores de asignación

Sirven para asignar valores a las variables, ya hemos utilizado en ejemplos anteriores el operador de asignación (=), pero hay otros operadores de este tipo.

```
= /* Asignación.

Asigna la parte de la derecha del igual a la parte de la izquierda.

A la derecha se colocan los valores finales y a la izquierda
generalmente se coloca una variable donde queremos guardar el dato. */

+= /* Asignación con suma.

Realiza la suma de la parte de la derecha con la de la izquierda
y guarda el resultado en la parte de la izquierda. */

-= // Asignación con resta

*= // Asignación de la multiplicación

/= // Asignación de la división

%= // Se obtiene el resto y se asigna
```

Ejemplos

```
ahorros = 7000; // asigna un 7000 a la variable.
ahorros += 3500; // incrementa en 3500 la variable ahorros.
ahorros /= 2; // divide entre 2 mis ahorros.
```

Operadores lógicos

Estos operadores sirven para realizar operaciones lógicas, que son aquellas que dan como resultado un verdadero o un falso, y se utilizan para tomar decisiones en nuestros scripts. En vez de trabajar con números, para realizar este tipo de operaciones se utilizan operandos boleanos, que conocimos anteriormente, que son el verdadero (true) y el falso (false). Los operadores lógicos relacionan los operadores booleanos para dar como resultado otro operador booleano.

En el siguiente grupo de operadores veremos sobre los operadores condicionales, que se pueden utilizar junto con los operadores lógicos para realizar sentencias todo lo complejas que necesitemos.

Por ejemplo:

```
if (x == 2 && y != 3) {
    /* SI la variable x vale 2 y
    la variable y es distinta de tres */
}
```

En la expresión condicional anterior estamos evaluando dos comprobaciones que se relacionan con un operador lógico. Por una parte x==2 devolverá un true en caso que la variable x valga 2 y por otra, y!=3 devolverá un true cuando la variable y tenga un valor distinto de 3. Ambas comprobaciones devuelven un booleano cada una, que luego se le aplica al operador lógico && para comprobar si ambas comprobaciones se cumplieron al mismo tiempo. Para ver ejemplos de operadores condicionales, necesitamos aprender estructuras de control como if, a las que no hemos llegado todavía.

```
! Operador NO o negación. Si era true pasa a false y viceversa.
&& Operador Y, si son los dos verdaderos vale verdadero.
|| Operador O, vale verdadero si por lo menos uno de ellos es verdadero.
```

Ejemplo:

```
miBoleano = true; // true
miBoleano = !miBoleano; // false

tengoHambre = true;
tengoComida = true;
comoComida = tengoHambre && tengoComida; // true
```

Operadores condicionales

Sirven para realizar expresiones condicionales todo lo complejas que deseemos. Estas expresiones se utilizan para tomar decisiones en función de la comparación de varios elementos, por ejemplo si un número es mayor que otro o si son iguales. Los operadores condicionales se utilizan en las expresiones condicionales para la toma de decisiones.

Seguidamente podemos ver la tabla de operadores condicionales:

```
== // iguales (en valor)
=== // estrictamente iguales (en valor y tipo)
!= // distintos (en valor)
!== // estrictamente distintos (en valor y tipo)
>  // Mayor que
<  // Menor que
>= // Mayor igual que
<= // Menor igual que
```

Operador typeof - control de tipos

Hemos podido comprobar que, para determinados operadores, es importante el tipo de datos que están manejando, puesto que si los datos son de un tipo se realizarán operaciones distintas que si son de otro.

Por ejemplo, cuando se utiliza el operador +, si se trata de números los sumaba, pero si se trataba de cadenas de caracteres las concatenan, entonces, el tipo de los datos determina que nuestras operaciones se realicen tal como esperábamos. Para comprobar el tipo de un dato se puede utilizar el operador typeof, que devuelve una cadena de texto que describe el tipo del operador que estamos comprobando.

```
var boleano = true;
var numerico = 22;
var numerico_flotante = 13.56;
var texto = "mi texto";
var fecha = new Date();

console.log("El tipo de booleano es: " + typeof boleano);
// El tipo de booleano es: boolean
console.log("El tipo de numerico es: " + typeof numerico);
// El tipo de numerico es: number
console.log("El tipo de numerico_flotante es: " + typeof numerico_flotante);
// El tipo de numerico_flotante es: number
console.log("El tipo de texto es: " + typeof texto);
// El tipo de texto es: string
console.log("El tipo de fecha es: " + typeof fecha);
// El tipo de fecha es: object
```

Condicionales

IF es una estructura de control utilizada para tomar decisiones. Es un condicional que sirve para realizar unas u otras operaciones en función de una expresión y primero se evalúa una expresión, si da resultado positivo se realizan las acciones relacionadas con el caso positivo. La sintaxis de la estructura IF es la siguiente.

```
if (expresión) {
    //acciones a realizar en caso positivo
    //...
}
```

Opcionalmente se puede definir un caso alternativo en caso que la condición de la expresión no se cumpla y por ende se salte la primera parte del bloque.

```
if (expresión) {
    //acciones a realizar en caso positivo
    //...
} else {
    //acciones a realizar en caso negativo
    //...
}
```

Las llaves engloban las acciones que queremos realizar en caso de que se cumplan o no las expresiones. Estas llaves han de colocarse siempre, excepto en el caso de que sólo haya una instrucción como acciones a realizar, que son opcionales.

Por ejemplo:

```
<script>

    if (llueve)
        alert("Cae agua");

    //Sería exactamente igual que este código:
    if (llueve){
        alert("Cae agua");
    }

</script>
```

Generalmente, cuando utilizamos las llaves, el código queda bastante más claro, porque se puede apreciar en un rápido vistazo qué instrucciones están dependiendo del caso positivo del if.

Los saltos de línea tampoco son necesarios y se han colocado también para que se vea mejor la estructura. Perfectamente podríamos colocar toda la instrucción IF en la misma línea de código, pero eso no ayudará a que las cosas estén claras.

Ejemplo de condicionales IF.

```
if (dia == "lunes") {
    console.log("Que tengas un feliz comienzo de semana");
}
```

Si es lunes nos deseará una feliz semana. No hará nada en caso contrario. Como en este ejemplo sólo indicamos una instrucción para el caso positivo, no hará falta utilizar las llaves (aunque sí sería recomendable haberlas puesto).

Este el próximo ejemplo comprueba si tiene crédito para realizar una supuesta compra. Para ello analizamos si el crédito es mayor o igual que el precio del artículo, si es así informo de la compra, introduzco el artículo en el carrito y resto el precio al crédito acumulado. Si el precio del artículo es superior al dinero disponible informo de la situación y mando al navegador a la página donde se muestra su carrito de la compra.

```
if (credito >= precio) {
    console.log("has comprado el artículo " + nuevoArtículo);
    // Enseño compra
    carrito += nuevoArticulo;
    // Introduzco el artículo en el carrito de la compra
    credito -= precio;
    // Disminuyo el crédito según el precio del artículo
} else {
    console.log("se te ha acabado el crédito");
    // informo que te falta dinero
    window.location = "carritodelacompra.html";
    // voy a la página del carrito
}
```

Expresiones condicionales

La expresión a evaluar se coloca siempre entre paréntesis y está compuesta por variables que se combinan entre sí mediante operadores condicionales.

Recordamos que los operadores condicionales relacionan dos variables y devuelven siempre un resultado booleano. Por ejemplo un operador condicional es el operador "es igual" (==), que devuelve true en caso de que los dos operandos sean iguales o false en caso de que sean distintos.

```
if (edad > 18) {
    console.log("puedes ver esta página");
}
```

Para este ejemplo utilizamos el operador condicional "es mayor" (>). En este caso, devuelve true si la variable edad es mayor que 18, con lo que se ejecutaría la línea siguiente que nos informa de que se puede ver el contenido para adultos.

Las expresiones condicionales se pueden combinar con las expresiones lógicas para crear expresiones más complejas. Recordamos que las expresiones lógicas son las que tienen como operandos a los booleanos y que devuelve otro valor booleano.

Son los operadores negación lógica, AND lógico y OR lógico.

```
if (bateria < 0.5 && redElectrica == 0) {
    console.log("la notebook se va a apagar en segundos");
}
```

Lo que hacemos es comprobar si la batería de nuestra supuesta notebook es menor que 0.5 (está casi vacía) y también comprobamos si no tiene red eléctrica (desenchufada). Luego, el operador lógico los relaciona con un Y, de

modo que si está casi sin batería Y sin red eléctrica, muestro el mensaje que el equipo se va a apagar.

La estructura if es de las más utilizadas en lenguajes de programación, para tomar decisiones en función de la evaluación de una sentencia.

Sentencias IF anidadas

Para hacer estructuras condicionales más complejas podemos anidar sentencias IF, es decir, colocar estructuras IF dentro de otras estructuras IF. Con un solo IF podemos evaluar y realizar una acción u otra según dos posibilidades, pero si tenemos más posibilidades que evaluar podemos anidar IF's y crear el flujo de código necesario para decidir correctamente.

Por ejemplo, para comprobar si un número es mayor menor o igual que otro, tengo que evaluar tres posibilidades distintas. Primero puedo comprobar si los dos números son iguales, si lo son, ya he resuelto el problema, pero si no son iguales todavía tendré que ver cuál de los dos es mayor.

Veamos este ejemplo en código Javascript:

```
let numero1 = 23;
let numero2 = 63;
if (numero1 == numero2){
   console.log("Los dos números son iguales");
}else{
   if (numero1 > numero2) {
      console.log("El primer número es mayor que el segundo");
   }else{
      console.log("El primer número es menor que el segundo");
   }
}
```

El flujo del programa primero evalúa si los dos números son iguales, en caso positivo se muestra un mensaje informando de ello, en caso contrario ya sabemos que son distintos, pero aún debemos averiguar cuál de los dos es mayor. Para eso se hace otra comparación para saber si el primero es mayor que el segundo. Si esta comparación da resultado positivo mostramos un mensaje que el primero es mayor que el segundo, en caso contrario indicó que el primero es menor que el segundo.

Operador Ternario

Este operador es un claro ejemplo de ahorro de líneas y caracteres al escribir los scripts.

Un ejemplo de uso del operador IF se puede ver a continuación.

```
let variable = (condición) ? valor1 : valor2;
```

Este ejemplo no sólo realiza una comparación de valores, además asigna un valor a una variable. Lo que hace es evaluar la condición (colocada entre paréntesis) y si es positiva asigna el valor1 a la variable y en caso contrario le asigna el valor2. Veamos un ejemplo:

```
let momento = (hora_actual < 12)
    ? "Antes del mediodía"
    : "Después del mediodía";
```

Este ejemplo analiza si la hora actual es menor que 12. Si es así, es antes del mediodía, y asigna "Antes del mediodía" a la variable momento. Si la hora es mayor o igual a 12 es después del mediodía, con lo que se asigna el texto "Después del mediodía" a la variable momento.

Estructura SWITCH de Javascript

Las estructuras de control son la manera con la que se puede dominar el flujo de los programas, para hacer cosas distintas en función de los estados de las variables. Switch se utiliza para tomar decisiones en función de distintos

estados de las variables. Esta expresión se utiliza cuando tenemos múltiples posibilidades como resultado de la evaluación de una sentencia.

Su sintaxis es la siguiente.

```
switch (expresión) {
    case valor1:
        // Sentencias a ejecutar si la expresión tiene como valor a valor1
    break
    case valor2:
        // Sentencias a ejecutar si la expresión tiene como valor a valor2
    break
    case valor3:
        // Sentencias a ejecutar si la expresión tiene como valor a valor3
    break
    default:
        // Sentencias a ejecutar si el valor no es ninguno de los anteriores
}
```

La expresión se evalúa, si vale valor1 se ejecutan las sentencias relacionadas con ese caso. Si la expresión vale valor2 se ejecutan las instrucciones relacionadas con ese valor y así sucesivamente, por tantas opciones como deseemos. Finalmente, para todos los casos no contemplados anteriormente se ejecuta el caso por defecto.

La palabra break es opcional, pero si no la ponemos a partir de que se encuentre coincidencia con un valor se ejecutarán todas las sentencias relacionadas con este y todas las siguientes. Es decir, si en nuestro esquema anterior no hubiese ningún break y la expresión valiese valor1, se ejecutarán las

sentencias relacionadas con valor1 y también las relacionadas con valor2, valor3 y default.

También es opcional la opción default u opción por defecto.

Ejemplo:

Indicar qué día de la semana es. Si el día es 1 (lunes) sacar un mensaje indicando, si el día es 2 (martes) debemos sacar un mensaje distinto y así sucesivamente para cada día de la semana, menos en el 6 (sábado) y 7 (domingo) que queremos mostrar el mensaje "es fin de semana". Para días mayores que 7 indicaremos que ese día no existe.

```
switch (dia_de_la_semana) {
    case 1:
        console.log("Es Lunes");
    break
    case 2:
        console.log("Es Martes");
    break
    case 3:
        console.log("Es Miércoles");
    break
    case 4:
        console.log("Es Jueves");
    break
    case 5:
        console.log("Es viernes");
    break
    case 6:
    case 7:
        console.log("Es fin de semana");
    break
    default:
        console.log("Ese día no existe");
}
```

El ejemplo es relativamente sencillo, solamente puede tener una pequeña dificultad, consistente en interpretar lo que pasa en el caso 6 y 7, que habíamos

dicho que teníamos que mostrar el mismo mensaje. En el caso 6 en realidad no indicamos ninguna instrucción, pero como tampoco colocamos un break se ejecutará la sentencia o sentencias del caso siguiente, que corresponden con la sentencia indicada en el caso 7 que es el mensaje que informa que es fin de semana. Si el caso es 7 simplemente se indica que es fin de semana, tal como se pretendía.

Ciclos y Bucles

Bucle FOR

El bucle FOR se utiliza para repetir una o más instrucciones un determinado número de veces. De entre todos los bucles, el FOR se suele utilizar cuando sabemos seguro el número de veces que queremos que se ejecute. La sintaxis del bucle for se muestra a continuación.

```
for (inicialización; condición; actualización) {
    //sentencias a ejecutar en cada iteración
}
```

El bucle FOR tiene tres partes incluidas entre los paréntesis, que nos sirven para definir cómo deseamos que se realicen las repeticiones. La primera parte es la inicialización, que se ejecuta solamente al comenzar la primera iteración del bucle. En esta parte se suele colocar la variable que utilizaremos para llevar la cuenta de las veces que se ejecuta el bucle.

La segunda parte es la condición, que se evaluará cada vez que comience una iteración del bucle. Contiene una expresión para decidir cuándo se ha de detener el bucle, o mejor dicho, la condición que se debe cumplir para que continúe la ejecución del bucle. Por último tenemos la actualización, que sirve

para indicar los cambios que queramos ejecutar en las variables cada vez que termina la iteración del bucle, antes de comprobar si se debe seguir ejecutando.

Después del for se colocan las sentencias que queremos que se ejecuten en cada iteración, acotadas entre llaves. Un ejemplo de utilización de este bucle lo podemos ver a continuación, donde se imprimirán los números del 0 al 10.

```
for (let i=0; i<=10; i++) {
    console.log(i);
    console.log("---");
}
```

En este caso se inicializa la variable i a 0. Como condición para realizar una iteración, se tiene que cumplir que la variable i sea menor o igual que 10. Como actualización se incrementará en 1 la variable i.

Como se puede comprobar, en una sola línea podemos indicar muchas cosas distintas y muy variadas, lo que permite una rápida configuración del bucle y una versatilidad enorme. Por ejemplo si queremos escribir los números del 1 al 1.000 de dos en dos se escribirá el siguiente bucle.

```
for (let i = 1; i <= 1000; i += 2) {
    console.log(i);
}
```

Si nos fijamos, en cada iteración actualizamos el valor de i incrementándose en 2 unidades.

Si queremos contar descendentemente del 200 al 10 utilizamos este bucle.

```
for (let i = 200; i >= 10; i--) {
    console.log(i);
}
```

En este caso decrementamos en una unidad la variable i en cada iteración, comenzando en el valor 200 y siempre que la variable tenga un valor mayor o igual que 10.

Bucle WHILE

Estos bucles se utilizan cuando queremos repetir la ejecución de unas sentencias un número indefinido de veces, siempre que se cumpla una condición. Sólo se indica, como veremos a continuación, la condición que se tiene que cumplir para que se realice una iteración.

```
while (condición){
    //sentencias a ejecutar
}
```

Un ejemplo de código donde se utiliza este bucle se puede ver a continuación.

```
var color = "";

while (color != "rojo"){

    color = prompt("Ingresa un color o escribe rojo para salir");

}
```

Este es un ejemplo de lo más sencillo que se puede hacer con un bucle WHILE.

Lo que hace es pedir que el usuario introduzca un color y lo hace repetidas veces, mientras que el color introducido no sea rojo. Para ejecutar un bucle como este primero tenemos que inicializar la variable que vamos utilizar en la condición de iteración del bucle.

Con la variable inicializada podemos escribir el bucle, que comprobará para ejecutarse que la variable color sea distinto de "rojo". En cada iteración del bucle se pide un nuevo color al usuario para actualizar la variable color y se termina la iteración, con lo que retornamos al principio del bucle, donde

tenemos que volver a evaluar si lo que hay en la variable color es "rojo" y así sucesivamente mientras que no se haya introducido como color el texto "rojo".

Bucle DO WHILE

El bucle do...while es la última de las estructuras para implementar repeticiones de las que dispone en Javascript y es una variación del bucle WHILE visto anteriormente.
Se utiliza generalmente cuando no sabemos cuantas veces se habrá de ejecutar el bucle, igual que el bucle WHILE, con la diferencia de que sabemos seguro que el bucle por lo menos se ejecutará una vez.

La sintaxis es la siguiente:

```
do {
    //sentencias del bucle
} while (condición)
```

El bucle se ejecuta siempre una vez y al final se evalúa la condición para decir si se ejecuta otra vez el bucle o se termina su ejecución.

Ejemplo de un bucle DO WHILE:

```
var color;

do {
   color = prompt("Ingresa un color oescribe rojo para salir");
} while (color != "rojo");
```

Este ejemplo funciona exactamente igual que el anterior, excepto que no tuvimos que inicializar la variable color antes de introducirnos en el bucle. Pide un color mientras que el color introducido es distinto que "rojo".

Ejemplo de uso de los bucles while

En este ejemplo vamos a declarar una variable e inicializarla a 0. Luego iremos sumando a esa variable un número aleatorio del 1 al 100 hasta que sumemos 1.000 o más, imprimiendo el valor de la variable suma después de cada operación. Será necesario utilizar el bucle WHILE porque no sabemos exactamente el número de iteraciones que tendremos que realizar (dependerá de los valores aleatorios que se vayan obteniendo).

```
var suma = 0;

while (suma < 1000){
    suma += parseInt(Math.random() * 100);
    console.log(suma + "\n");
}
```

Break y Continue

Existen dos instrucciones que se pueden usar en de las distintas estructuras de control y principalmente en los bucles, que te servirán para controlar dos tipos de situaciones.

Son las instrucciones break y continue:

break: Significa detener la ejecución de un bucle y salirse de él.
continue: Sirve para detener la iteración actual y volver al principio del bucle para realizar otra iteración, si corresponde.

Break

Se detiene un bucle utilizando la palabra break. Detener un bucle significa salirse de él y dejarlo todo como está para continuar con el flujo del programa inmediatamente después del bucle.

```
for (let i = 0; i < 10; i++){
    console.log(i);

    escribe = prompt("dime si continuo preguntando...", "si");

    if (escribe == "no") {
        break;
    }
}
```

Este ejemplo escribe los números del 0 al 9 y en cada iteración del bucle pregunta al usuario si desea continuar. Si el usuario dice cualquier cosa continua, excepto cuando dice "no", situación en la cual se sale del bucle y deja la cuenta por donde se había quedado.

Continue

Sirve para volver al principio del bucle en cualquier momento, sin ejecutar las líneas que haya por debajo de la palabra continue.

```
var i = 0;

while (i < 7) {

    let incrementar = prompt("La cuenta está en " + i + ", dime si incremento", "si");

    if (incrementar == "no") {
        continue;
    }

    i++
}
```

Este ejemplo, en condiciones normales contaría hasta desde i=0 hasta i=7, pero cada vez que se ejecuta el bucle pregunta al usuario si desea incrementar la variable o no. Si introduce "no" se ejecuta la sentencia continue, con lo que se vuelve al principio del bucle sin llegar a incrementar en 1 la variable i, ya que se ignorarán las sentencia que haya por debajo del continue.

Ejemplo de la sentencia break

Un bucle FOR planeado para llegar hasta 1.000 pero que lo vamos a detener con break cuando lleguemos a 333.

```
for (i=0;i<=1000;i++){
    console.log(i + "\n");

    if (i==333) {
        break;
    }
}
```

Funciones

Cuando se desarrolla una aplicación, solemos repetir consecuentemente diversas instrucciones. Esto presenta una serie de problemas a futuro ya que:

- El código de la aplicación es mucho más largo y repetitivo.
- Si se quiere modificar alguna de las instrucciones repetidas, se deben hacer tantas modificaciones como veces se haya escrito esa instrucción, lo que se convierte en un trabajo muy pesado y muy propenso a cometer errores.

Las funciones son la solución a todos estos problemas, tanto en JavaScript como en el resto de lenguajes de programación. Una función es un conjunto de instrucciones que se agrupan para realizar una tarea concreta y que se pueden reutilizar fácilmente.

Estas nos permiten abstraer la solución a un problema dentro de una instrucción de código y adaptarla para que tenga diversos resultados dependiendo de su uso o aplicación.

Declaración de una función

En javascript tenemos varias formas para crear una función, actualmente veremos las funciones declaradas y las expresadas.

Declarada

```
function buscar() {
    // rutina o proceso a realizar
}
```

Se utiliza la palabra reservada ***function*** seguida del nombre de la función pegada a un juego de paréntesis y un bloque de llaves donde se ingresará el código que realizará la función.

Expresada

```
const buscar = function() {

    // rutina o proceso a realizar

}
```

Agencia de Aprendizaje a lo largo de la vida

En este caso la función es anónima (sin nombre) y se guarda dentro de una variable o constante tradicional.

La diferencia fundamental entre las funciones declaradas y las funciones expresadas es que estas últimas sólo están disponibles a partir de la inicialización de la variable. Si ejecutamos la variable antes de declararla, nos dará un error.

Invocar una función

Una vez hayamos declarado nuestra nueva función, para poder utilizarla debemos invocarla. Para ello escribiremos el nombre de la función seguida de un par de paréntesis.

```
function mostrarSuma() {

    console.log(20 + 10);

}

mostrarSuma(); // Imprime 30 por consola
```

En los casos que las funciones retornen un valor podemos guardar el resultado en una variable para ser utilizado posteriormente.

Palabra reservada RETURN

Existen casos donde nuestras funciones deben ejecutar cierta rutina o proceso sin necesidad de devolver un resultado, como por ejemplo en la WEB borrar un elemento de nuestro HTML que ya no necesitamos o imprimir un valor por consola, sin embargo en la mayoría de los casos nuestras funciones abstraen o guardan comportamientos que finalmente retornan un dato que tendrá que ser utilizado en otra parte de nuestro código, como por ejemplo el resultado de la suma de 2 valores.

Para capturar nuevamente ese valor que resulta de la ejecución de la función debemos utilizar la palabra reservada ***return*** antes del valor a devolver.

Ese valor se puede guardar en una variable para ser utilizado más adelante o implementarlo directamente desde la invocación de la función.

```javascript
function sumar() {
    let resultado = 20 + 10;

    return resultado;
}

// Guarda el valor 30 en la variable suma
let suma = sumar();

// Imprime en consola el valor
// devuelto por la función.
console.log(sumar());
```

Es importante mencionar que todo lo que se escriba luego de la sentencia ***return*** en el ámbito de nuestra función será ignorado, ya que esta es considerada como la línea de salida del bloque de la función.

Función de Flecha

Introducidas al lenguaje a partir de la especificación ES6 del estándar EcmaScript, son otra forma de declarar nuestras funciones y hoy en día juegan un papel fundamental en la forma en la que se escriben programas de Javascript moderno.

Una de las principales ventajas es que nos permiten declarar funciones que ocupen tan solo una línea de código.

Para declarar un arrow function usamos la siguiente sintaxis:

```
const buscar = () => { /* rutina o proceso */ };
```

Cómo podemos observar, es similar a la declaración de una función expresada solo que prescindimos de la palabra reservada function y luego de los paréntesis colocamos una “flecha”. A continuación de ella el uso de llaves es OPCIONAL y en caso de NO usarlas, la flecha será un RETURN implícito para lo que esté inmediatamente después en la misma línea.

```
const buscar = () => 'No entontramos lo que buscabas';
```

En este caso, no es necesaria la palabra return, ya que se encuentra implícito en la flecha. Sin embargo en los casos donde necesitamos declarar nuestra función en más de una línea, entonces si necesitaremos las llaves y la palabra return para devolver nuestros valores.

```
const sumar = () => {
    let resultado = 20 + 10;

    return resultado;
};
```

Parámetros y Argumentos

Hasta ahora solo vimos que cada declaración de una función, ya sea declarada, expresada o de flecha llevaba consigo un par de paréntesis vacíos. Estos no están allí de casualidad ya que su función principal es la de recibir parámetros.

Un parámetro es un nombre que reservamos como una suerte de comodín para indicarle a la función que al momento de ser invocada deberán pasar un valor en esa posición para luego ser utilizado dentro de la misma función. Ese valor pasado en la invocación y que toma el lugar del parámetro se lo conoce como argumento.

Para poner los términos claros:

- Un parámetro es una variable listada dentro de los paréntesis en la declaración de función (es un término reservado para el momento de la declaración).

- Un argumento es el valor que es pasado a la función cuando esta es llamada (es el término para el momento en que se llama).

Al momento de crear nuestra función, debemos tener claro cuántos parámetros vamos a esperar o necesitar y colocarlos dentro de los paréntesis separados por cada uno por una coma.

Estos nos van a permitir crear funciones “multiuso” es decir, que no operen bajo datos estáticos, sino que el resultado que devuelva la función dependa de los datos que recibe como input.

En el próximo ejemplo podemos ver una función usada más de una vez que retorna diferentes resultados dependiendo los argumentos pasados como parámetro.

```
const sumar = (a, b) => {
    let resultado = a + b;

    return resultado;
};

sumar(12, 5); // devuelve 17
sumar(20, 33); // devuleve 53
```

En este ejemplo, ***a*** y ***b*** son los parámetros mientras que 12,5, 20 y 33 son los argumentos pasados como parámetros de la función en cada invocación.

Gracias a ellos podemos reutilizar la lógica guardada dentro de la función sumar para obtener distintos resultados dependiendo de los valores que recibe.

Callback

Una función de callback es una función que se pasa a otra función como un argumento, que luego se invoca dentro de la función externa para completar algún tipo de rutina o acción. Es decir, pasamos una función B por parámetro a una función A, de modo que la función A puede ejecutar esa función B de forma genérica desde su código, y nosotros podemos definirlas desde fuera de dicha función.

```
const sumar = (a, b, accion) => {
    let resultado = a + b;

    return accion(resultado);
};

// devuelve 17 por la consola
sumar(12, 5, console.log);

// devuleve 53 con un alert
sumar(20, 33, alert);
```

En el ejemplo anterior, tenemos un tercer parámetro llamado acción que recibirá como argumento otra función, en este caso utilizamos console.log y alert que ya conocemos pero esta podría ser también cualquier otra función creada por nosotros para tal fin.

Esa acción que enviamos, es ejecutada o invocada en el return de la función principal.
Cabe destacar que los callbacks a menudo se utilizan para ejecutar una acción luego que otra acción haya finalizado, es decir, nos permiten comenzar una tarea, continuar con la ejecución de nuestro programa hasta que la tarea haya finalizado y luego recuperar el resultado en otra función que retorne el valor o realice otra acción.

Este es el principio de asincronismo conocido en javascript ya que al ser un lenguaje de un solo hilo de ejecución, solo podemos procesar una tarea a la vez y eso bloquea los tiempos de espera o procesamiento del resto del programa.

Gracias al asincronismo y como trabaja es que vamos a encontrar alternativas para “dejar corriendo” ciertas tareas mientras nuestro programa continúa y recuperar lo resultados de esas tareas cuando hayan finalizado, sin embargo este concepto lo trabajaremos más adelante.

Arrays

En ocasiones, a los arrays se les llama vectores, matrices e incluso arreglos. No obstante, el término array es el más utilizado y es una palabra comúnmente aceptada en el entorno de la programación.

Un array es una colección de variables o datos, que pueden ser todas del mismo tipo o cada una de un tipo diferente. Su utilidad se comprende mejor con un ejemplo sencillo: si una aplicación necesita manejar los días de la semana, se podrían crear siete variables de tipo texto lo cual representaría mayor dificultad a la hora de tratar esa información como un conjunto de datos relacionados.

Imaginemos que en lugar de los días de la semana queremos guardar todos los países del mundo, necesitaríamos declarar más de 200 variables diferentes.

Para declarar un array debemos utilizar la misma sintaxis que para la declaración de variables, salvo que luego del operador de asignación colocaremos un par de corchetes donde alojaremos los datos de nuestro array separados por comas.

```
const frutas = ['Manzana', 'Pera', 'Frutilla', 'Kiwi', 'Sandía'];
```

En este ejemplo declaramos un array de 5 elementos, donde si bien nosotros conocemos su extensión en este caso, muchas veces consultar este dato ya sea porque no lo sabemos o porque fue modificado a través de la ejecución del programa.

Para saber cuántos elementos posee un array debemos acceder a su propiedad length (al igual que sucede con los strings), de este modo obtendremos la cantidad de elementos que existen en nuestra colección.

```
const frutas = ['Manzana', 'Pera', 'Frutilla', 'Kiwi', 'Sandía'];

console.log(frutas.length); // 5
```

Otro punto importante es que cada elemento dentro de un array posee un índice único. Esté índice va en orden ascendente y comienza en 0 desde el primer al último elemento del array.

```
// indice:          0          1          2          3          4
const frutas = ['Manzana', 'Pera', 'Frutilla', 'Kiwi', 'Sandía'];
```

Sabiendo esto, ahora podemos consultar los elementos de un array en relación a la posición que ocupan dentro de él.

Para ellos usaremos la siguiente sintaxis:

```
// indice:          0         1         2         3         4
const frutas = ['Manzana', 'Pera', 'Frutilla', 'Kiwi', 'Sandía'];

console.log(frutas[2]); // Frutilla
```

De este modo, rescatamos el valor del elemento con índice 2 del nuestro arreglo, en este caso el elemento número 3 con valor “Frutilla”.

A partir de la especificación EcmaScript 2022, podemos acceder a un array con el método .at().

```
// indice:          0         1         2         3         4
const frutas = ['Manzana', 'Pera', 'Frutilla', 'Kiwi', 'Sandía'];

console.log(frutas.at(3)); // Kiwi
```

Métodos de Arrays

Son funciones que poseen los objetos array de forma nativa, con ellas seremos capaces de agregar, quitar, buscar, filtrar, modificar, cortar, y muchas cosas más sobre los elementos internos de nuestros arreglos.

En este apartado conoceremos algunos de ellos.

Añadir o eliminar elementos

Existen varias formas de añadir elementos a un array ya existente. Tengamos en cuenta que en todos estos casos estamos mutando (variando los elementos del array ya existente).

```
const frutas = ['Manzana', 'Pera', 'Frutilla', 'Kiwi', 'Sandía'];

frutas.push('Ananá'); // Agregamos un elemento al final del array
frutas.unshift('Melón'); // Agregamos un elemento al inicio del array

console.log(frutas);

// ['Melón', 'Manzana', 'Pera', 'Frutilla', 'Kiwi', 'Sandía', 'Ananá'];

frutas.pop(); // Elimina el último elemento del array
frutas.shift(); // Elimina el primer elemento del array

console.log(frutas);

// ['Manzana', 'Pera', 'Frutilla', 'Kiwi', 'Sandía'];
```

Unir elementos de un array en una cadena (string)

En este caso utilizamos el método .join() que recibe como parámetro el separador de nuestros elementos.

```
frutas.join('-')

console.log(frutas);

'Melón - Manzana - Pera - Frutilla - Kiwi - Sandía - Ananá'
```

Recorrer un los elementos de un array

Nos permite recorrer los elementos de un array y ejecutar una acción frente a cada iteración. El método forEach() no devuelve nada y espera que se le pase por parámetro una función de callback que se ejecutará por cada elemento del array.

```
const frutas = ['Manzana', 'Pera', 'Frutilla', 'Kiwi', 'Sandía'];

frutas.forEach((fruta) => console.log(fruta));

// Manzana
// Pera
// Frutilla
// Kiwi
// Sandía
```

Filtrar elementos en un array

Para ello usaremos el método .filter() a quien le debemos pasar una función de callback con la condición que deben cumplir los elementos para ser filtrados.

```
const precios = [100, 349, 3, 63, 524, 217, 14];

const mayores = precios.filter( (precio) => precio >= 100);

console.log(mayores);

// [100, 349, 524, 217];
```

Modificar o crear arrays a partir de otros

Hay situaciones en las que tenemos un array y queremos crear nuevos subarrays a partir del original, o simplemente deseamos modificarlo para hacer ciertos cambios, pero de una forma más general y no tener que hacerlo elemento a elemento.

Método	Descripción
.slice(start, end)	Devuelve los elementos desde la posición start hasta end (excluido).
.splice(start, size)	Altera el array, eliminando size elementos desde posición start. **Mutable**

.copyWithin(pos, start, end)	Altera el array, modificando en pos y copiando los ítems desde start a end. **Mutable**
.fill(element, start, end)	Altera los elementos del array desde start hasta end. **Mutable**

Crear nuevos arrays a partir de una condición

El método map() es un método muy potente y útil para trabajar con arrays, puesto que su objetivo es devolver un nuevo array donde cada uno de sus elementos será lo que devuelva la función callback por cada uno de los elementos del array original.

```
const names = ["Carla", "Pablo", "Lucía", "José", "Camila"];
const nameSizes = names.map((name) => name.length);

console.log(nameSizes); // Devuelve [3, 5, 5, 9, 9]
```

Acumular los valores de una array

.reduce(); se encarga de recorrer todos los elementos del array, e ir acumulando sus valores (o alguna operación diferente) y sumarlo todo, para devolver su resultado final.

```
const numbers = [95, 5, 25, 10, 25];
numbers.reduce((first, second) => {
  console.log(`F=${first} S=${second}`);
  return first + second;
});

// F=95  S=5    (1ª iteración: elemento 1: 95 + elemento 2: 5) = 100
// F=100 S=25   (2ª iteración: 100 + elemento 3: 25) = 125
// F=125 S=10   (3ª iteración: 125 + elemento 4: 10) = 135
// F=135 S=25   (4ª iteración: 135 + elemento 5: 25) = 160

// Finalmente, devuelve 160
```

En esta oportunidad solo conocimos algunos pero existen muchos más métodos que nos brindaran la posibilidad de trabajar sobre nuestros arrays.

Iteradores de Arrays

Si bien es posible recorrer arrays con los ciclos tradicionales como un for, existen una estructura para este fin que simplifica mucho el trabajo a realizar.

Esta es la estructura for of, que propone la siguiente sintaxis:

```
const frutas = ['Manzana', 'Pera', 'Frutilla', 'Kiwi', 'Sandía'];

for (let fruta of frutas) {

    console.log(fruta);
}

// Manzana
// Pera
// Frutilla
// Kiwi
// Sandía
```