



Centro Integrado de Formación Profesional

AVILÉS

Principado de Asturias

**UNIDAD 2:
EL LENGUAJE JAVASCRIPT. SINTAXIS
BÁSICA**

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	16/09/2024	Aprobado	Primera versión
1.1	02/10/2024	Aprobado	Corrección de errores
1.2	08/10/2024	Aprobado	Modificación de ejemplos de código

ÍNDICE

ÍNDICE	1
UNIDAD 2: EL LENGUAJE JAVASCRIPT. SINTAXIS BÁSICA.....	3
2.1 Sentencias. Comentarios. Bloques de código	3
2.2 Etiquetas y ubicación del código.....	4
2.3 Variables. Ámbito. Tipos de datos.....	6
2.4 Conversión de tipos	9
2.5 Literales.....	11
2.6 Operadores. Expresiones. Asignación.....	12
2.6.1 Comparación de cadenas teniendo en cuenta diferencias locales.....	13
2.7. Estructuras de decisión.....	15
2.7.1 Estructuras condicionales simples	15
2.7.2 Estructuras condicionales compuestas.....	15
2.7.3 Estructuras condicionales anidadas.....	17
2.7.4 Switch	18
2.7.5 Valores pseudo-Booleanos: Truly y Falsy	20
2.8 Bucles. Anidado. Sentencias de ruptura y continuación.....	25
2.8.1 Bucle While.....	25
2.8.2 Bucle Do/While	26
2.8.3 Bucle For	27

2.8.4 Uso avanzado de bucles determinados	28
2.8.5 Bucle for...in.....	31
2.9 Prueba y documentación de código	32
2.9.1 Estructuras de control de errores	32
2.9.2 Documentación del código.....	34
BIBLIOGRAFÍA - WEBGRAFÍA	35

UNIDAD 2: EL LENGUAJE JAVASCRIPT. SINTAXIS BÁSICA

JavaScript es un lenguaje de scripts que sigue el estándar ECMAScript y que fue originalmente diseñado para ser utilizado con el navegador Netscape. Se puede decir que, virtualmente, todos los navegadores actuales soportan JavaScript. Es preciso tener en cuenta que, aunque Java aparece en su nombre, no hay relación entre ambos lenguajes más allá de tener una sintaxis parecida. Al ser un lenguaje interpretado, requiere de un intérprete que está integrado en el navegador y gracias a ello, este puede crear páginas web dinámicas.

En definitiva, JavaScript añade comportamiento dinámico a lo que sería el contenido estático de una página. Como se verá posteriormente, JavaScript es capaz de actuar sobre lo que se conoce como DOM (Document Object Model), una estructura que genera el propio navegador web.

JavaScript establece un conjunto de normas sintácticas básicas que deben respetarse cuando se implementa un script. En una página se pueden incluir tantos scripts como se considere necesario, tanto en la cabecera como en el cuerpo de esta. Hay que tener en cuenta, eso sí, que JavaScript es sensible a mayúsculas y minúsculas. Por lo tanto, para JavaScript, False es distinto de false.

2.1 SENTENCIAS. COMENTARIOS. BLOQUES DE CÓDIGO

Como ya es conocido, un programa se compone de una lista de instrucciones que serán ejecutadas, en este caso interpretadas, por el ordenador. Estas instrucciones es lo que se conocen como sentencias. También es preciso recordar que JavaScript tiene una sintaxis basada en C, con lo que cada sentencia de código debe terminar con un punto y coma (;)

Por otra parte, también al igual que su lenguaje ancestro, los bloques de código en JavaScript se delimitan por llaves, una de apertura y otra de cierre:

```
{  
    // Aquí se ubica el código referente a este bloque  
}
```

En cuanto a los comentarios y, precisamente por su lenguaje “abuelo” C, en JavaScript se pueden usar comentarios de línea (encabezados por //) o de bloque (comenzando por /* y terminando por */). A continuación, se pueden ver los dos tipos de comentario en código JavaScript:

```
// Esto es un comentario en una línea  
/* Esto es un comentario en un bloque.  
    Como puede verse, puede ocupar más de una línea  
    y solo se termina cuando se llegue al cierre */
```

2.2 ETIQUETAS Y UBICACIÓN DEL CÓDIGO

Un lenguaje de programación está compuesto por palabras básicas, denominadas tokens o etiquetas, que tienen un significado especial. Las etiquetas de JavaScript se clasifican en cuatro grupos: identificadores, palabras reservadas, literales y operadores:

- **Identificadores.** Un identificador es un nombre que representa una variable, un método o un objeto. Está compuesto por una combinación de caracteres (letras y/o guiones bajos) y dígitos con algunas restricciones: deben comenzar por una letra o un guion bajo y en su totalidad no podrán coincidir con ninguna de las palabras reservadas del lenguaje.
- **Palabras reservadas.** Todo lenguaje de programación se reserva determinados identificadores para usos específicos. Estos se conocen como palabras reservadas. JavaScript tiene diversas palabras reservadas que sirven, por ejemplo, para declarar variables o indicar que se repitan varias veces las instrucciones de un script. Cada palabra reservada tiene su propia sintaxis.
- **Literales.** Un literal se usa para representar valores fijos y está compuesto por una combinación de números o caracteres. Un literal representa un valor que no varía durante la ejecución de un script. En JavaScript existen cinco tipos de literales: enteros, reales, booleanos, cadenas de caracteres (Strings) y caracteres especiales.
 - Un literal entero representa un valor numérico entero expresado en base decimal, octal o hexadecimal. Los valores octales deben comenzar con el valor 0, mientras que los hexadecimales deben comenzar con 0x ó 0X.
 - Un literal real representa un valor numérico real. Pueden expresarse tanto en notación científica como estándar. Con la primera usando la letra e o E para el exponente y tanto la base como el exponente pueden ir precedidos del signo + o -.
 - Los literales booleanos son las palabras reservadas true y false (en minúsculas), los cuales tienen asociados los valores numéricos 1 y 0 respectivamente.
 - Las cadenas de caracteres pueden incluirse entre dobles o simples comillas. Por ejemplo, la cadena *Adios* puede especificarse como "Adios" o 'Adios'.
 - El último tipo de literales de JavaScript son los caracteres especiales. Por ejemplo, si se desean introducir unas dobles comillas en una cadena, se debe hacer de forma diferente ya que las comillas se utilizan como elemento contenedor del propio texto. Para ello se utiliza lo que se conoce como carácter de escape. En el caso de inclusión de dobles comillas dentro de una cadena, se debe escribir \", de forma que al encontrarse con el carácter de escape interprete que no debe procesar el siguiente carácter y lo muestre como tal.

A continuación, se enumeran una serie de caracteres especiales que se pueden incluir en el código JavaScript:

<code>\b</code>	borra el último carácter
<code>\f</code>	genera alimentación de línea
<code>\n</code>	nueva línea
<code>\r</code>	retorno de carro
<code>\t</code>	tabulador
<code>\\</code>	<code>\</code>
<code>\'</code>	<code>'</code>
<code>\"</code>	<code>"</code>

También se pueden incluir comillas simples y dobles dentro de una cadena siempre y cuando se defina el literal con el otro tipo de comilla: **"Esto es un literal string"** o **'Esto es un literal string'**

- **Operadores.** Son símbolos que expresan una operación a realizar. Los más habituales son los de operaciones aritméticas simples, `+`, `-`, `*`, `/`

2.3 VARIABLES. ÁMBITO. TIPOS DE DATOS

Las variables son posiciones de memoria de la máquina a las que se les asigna un nombre o identificador y a la que se puede acceder para obtener o modificar el valor almacenado, simplemente dando su nombre.

Las operaciones básicas que permite una variable son: definición, inicialización, consulta y modificación. Para definir una variable se puede usar la palabra reservada **var** seguida del identificador de la variable.

Ejemplos:

```
var numero1;  
var literal1, literal2;  
var numero3 = 8;  
var numero4 = 3, numero5 = 6;  
numero1 = 24;  
numero1;  
numero1 += 3;
```

Cuando JavaScript interpreta la expresión **var numero1**, reserva un espacio de memoria para una nueva variable y anota que el nombre de esa posición de memoria es **numero1**. Al definir una variable, antes de poder consultar su valor, se debe inicializar. Esto se hace con el operador de asignación. Por ejemplo, **numero1 = 24**.

Para consultar el valor de una variable solo hay que indicar su identificador. La expresión **numero1** devuelve el valor de esa variable. También se puede definir una variable sin necesidad de la palabra **var**.

Las variables de JavaScript pueden ser de varios tipos, entendiendo esto como el rango de valores que se pueden almacenar en ellas. Los tipos de JavaScript según ECMAScript son:

- **undefined**: variable a la que no se le ha asignado ningún valor o no declarada.
- **boolean** (booleano): cierto o falso. Ejemplo: **true**.
- **number**: números enteros y reales, incluyendo algunos valores especiales como **infinity**. Ejemplo: **3.1416**
- **string**: cadenas de texto. Ejemplo: **"abcd"**
- **bigint**: permite operar con números enteros de tamaño arbitrario, no está limitado el tamaño como ocurre en el tipo **number**. Hay que añadir **n** al final de un número para convertirlo en **BigInt**. Es preciso fijarse en que un número no debe ser obligatoriamente "grande" para declararlo como **BigInt**. Puede aplicarse a cualquier valor entero, por ejemplo: **let x = 42n**.
- **Symbol**: es un tipo especial que permite crear funcionalidades similares a las enumeraciones de otros lenguajes, puesto que cada **symbol** que se crea se garantiza que se trata de un valor único.

- **null**: El tipo nulo tiene exactamente un único valor: **null** se trata de un valor especial. El operador **typeof** lo considera un **object** y es el valor devuelto por algunas funciones cuando el resultado no es válido, a diferencia de **undefined**, que es el valor de una variable a la que no se le ha asignado ningún valor.
- **Object** (objeto): Ejemplo: **document**. Tipo asignado a casi cualquier elemento instanciado mediante la palabra clave **new** (**array**, **map**, **set**, **date**, etc.), así como objetos declarados como literales (por ejemplo, **{nombre: 'Juan', edad: '23'}**).

Se puede decir que tanto **number**, como **boolean** y **string** son tipos primitivos de JavaScript, con lo que no pueden tener ni propiedades ni métodos. Complementarios a ellos existen los que se conocen como **wrappers** o envoltorios, los cuales se identifican con el mismo nombre, pero la primera letra en mayúscula (**Number**, **Boolean** y **String**). Estos tipos especiales son muy importantes para la conversión entre objetos y literales ya que, a diferencia de los primitivos, en este caso sí disponen de métodos.

Un ejemplo muy ilustrativo se puede realizar con la función **typeof** que devuelve el tipo de una expresión, tal como puede verse en el siguiente código (en comentarios, la salida de la función):

```
typeof "abc"; // string (primitivo)
typeof String("abc"); // string
typeof new String("abc"); // object
typeof (new String("abc")).valueOf(); // string
```

En JavaScript al definir una variable no se indica el tipo al que pertenece, con lo cual se puede declarar con un valor numérico inicialmente y, a posteriori, hacer que pase a ser un **string** tal como puede verse en el código siguiente:

```
var longitud = 10;
longitud = "10";
```

El último concepto relacionado con las variables es el ámbito. Éste puede ser local o global. Cuando una variable tiene ámbito global, cualquier sentencia de cualquier script que contenga la página HTML puede consultar o modificar la variable, mientras que cuando es local, sólo las sentencias que forman parte de la visibilidad de la variable pueden acceder a ella.

Si se declara una variable dentro de una función con **var**, el ámbito de esa variable es local a esa función. Si la declaración de una variable se hace dentro de una función sin usar **var**, ésta tendrá un ámbito global.

Se debe tener cuidado cuando se usan variables no declaradas, ya que JavaScript las considera globales, es decir, cualquier script de la página HTML podría hacer referencia a esa variable y modificar su contenido.

En las últimas versiones de JavaScript aparece una nueva forma de declarar variables que se pretende acabe desplazando a la clásica con **var**. Se trata del identificador **let** el cual tiene un

comportamiento más esperado que el de **var**. Si se declara una variable con **let**, solo estará disponible en el ámbito del bloque en el que ha sido declarada, algo que no sucede con **var**. Se puede decir que el comportamiento con esta palabra clave es muy similar al que tiene una inicialización de variable en cualquier otro lenguaje, acabando con lo que sería la barra libre que permite **var**, que incluso lleva al principio del bloque su inicialización pudiéndose usar la variable antes de ser inicializada. Para ilustrar la comparativa entre ambos modos, se verá a continuación un ejemplo de código:

```
let extLet = 1;
var extVar = 1;
if (true) {
  let intLet = 1;
  var intVar = 1;

  console.log("Dentro del bloque");
  console.log("extLet:", extLet);
  console.log("extVar:", extVar);
  console.log("intLet:", intLet);
  console.log("intVar:", intVar);
}

console.log("Fuera del bloque");
console.log("extLet:", extLet);
console.log("extVar:", extVar);
console.log("intVar:", intVar);
console.log("intLet:", intLet);
```

Si se prueba este código, el sistema lanzará una excepción “intLet is not defined” ya que la variable se está intentando utilizar fuera de su ámbito. En cambio, con las declaradas como **var** no da ningún problema al poder accederse desde cualquier parte del código.

Por último, otra forma de declarar una variable es mediante la palabra reservada **const**. En este caso, el ámbito de la variable es el mismo que con **let**, pero prohíbe sobrescribir el valor y, por consiguiente, no se trataría de trabajar con una variable, sino con una constante.

2.4 CONVERSIÓN DE TIPOS

Para operar con distintas variables, JavaScript debe conocer el tipo de datos de las variables, y entonces puede aplicar sus reglas internas. Por ejemplo, en JavaScript es válido escribir:

```
let animal = "Águila"; // String
let numPatas = 2; // Number

console.log (animal + numPatas);
// muestra Águila2
```

Esto ocurre porque internamente numPatas se convierte a cadena, que se concatena con la variable animal. Además, en JavaScript las expresiones se evalúan de izquierda a derecha. Estas dos sentencias dan resultados distintos:

```
let animal = "Águila"; // String
let numPatas = 2; // Number

console.log(numPatas + numPatas + animal); // 4Águila
console.log(animal + numPatas + numPatas); // Águila22
```

En el primer caso se han sumado los valores de numPatas y, a continuación, se ha concatenado "Águila" (Number + Number = tipo Number y después Number + String = tipo String), mientras que en el segundo caso primero se produce la concatenación de "Águila" y 2, y en la cadena resultante se concatena el valor 2.

Por la misma razón, si el valor de una de las variables fuese un número entre comillas (por ejemplo: "2"), el resultado sería la concatenación de los dos números en lugar de la suma:

```
let numPatas = 2; // Number
let numColas = "1"; // String

console.log (numPatas + numColas); // "21"
```

Para asegurar que el tipo es el esperado, JavaScript ofrece la posibilidad de realizar las siguientes conversiones de tipos:

- Boolean(valor): convierte el valor booleano.
- String(valor): convierte el valor en una cadena de texto. Otra opción es realizar una concatenación del valor con una cadena vacía.
- Number(valor): convierte el valor en un número. Si el valor no es válido, el resultado será NaN (no es un número).
- parseInt(valor): convierte el valor en un número entero, aunque se encuentre un separador decimal.
- parseFloat(valor): convierte el valor en un número real.

El siguiente código muestra varios ejemplos de conversión:

```
let cadena = "3.1415";
let nombre = "42";
let nom = "Joan";
let aprobado = true;

// Conversiones a booleano
console.log(Boolean(cadena)); // true
console.log(Boolean(0)); // false
console.log(Boolean("")); // false
console.log(Boolean(null)); // false
console.log(Boolean(undefined)); // false

// Conversiones a cadena
console.log(String(nombre)); // "42"
console.log(String(nombre) + nombre); // "4242"
console.log(String(aprobado)); // "true"

// Conversiones a nombres
console.log(Number(nom)); // NaN, no es un nombre
console.log(Number(cadena) * 2); // 6.283
console.log(Number(aprobado)); // 1
console.log(parseInt(cadena)); // 3
console.log(parseFloat(cadena)); // 3.1415
```

Hay que tener en cuenta que JavaScript no da otra opción para realizar conversiones de tipos, pero esto no supone ningún inconveniente porque es innecesario, ya que no hay restricciones a los valores que se pueden pasar a las funciones y los tipos de las variables son dinámicos. Es decir, que la misma variable puede cambiar de tipos de datos sin problema:

```
let z = 34;
z = "Ahora soy una cadena";
z = true;
z = undefined;
```

2.5 LITERALES

Los literales son representaciones directas y explícitas de un valor fijo en el código. En el apartado de variables ya se vieron varios ejemplos. Un concepto moderno de **JavaScript** son los **String literals** o **templates**. Este tipo de literal añade a las clásicas formas de expresar las cadenas con comillas simples o dobles, la comilla invertida (```). Estos templates evaluarán el contenido entre estas comillas como si fuera código, permitiendo incrustar expresiones concretas como se puede ver en este ejemplo:

```
const libs = ["React", "Vue", "Angular"];  
console.log(`Este curso trata de ${libs[0]}`);
```

En este caso, `${libs[0]}` se evalúa dentro de la cadena, con lo que permite más dinamismo en el código. Otro ejemplo más complejo:

```
const l = libs.length;  
console.log(`  
  ${l > 1 ? `Estas ${l} bibliotecas` : "Esta biblioteca"}:  
  ${libs.join(", ")}  
`);
```

En la codificación anterior, se anidan dos **string literals** que además usan un operador ternario en su interior.

2.6 OPERADORES. EXPRESIONES. ASIGNACIÓN

En este apartado se verán los distintos tipos de operadores utilizados en JavaScript:

1. Operadores aritméticos. Prácticamente idénticos a los existentes en lenguajes basados en C se pueden consultar en la siguiente tabla.

Operador	Descripción
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo o resto
++	Incremento en 1
--	Decremento en 1

2. Operadores de asignación. Por ejemplo, para x e y, la sintaxis es del tipo x operador y (i.e. x += y)

Operador	Descripción
=	x = y
+=	x = x + y
-=	x = x - y
*=	x = x * y
/=	x = x / y
%=	x = x % y

3. Operadores de manejo de cadenas. Se pueden utilizar los operadores + y += para concatenar cadenas. Ejemplo: var saludo = "hola" + "," + "mundo"
4. Operadores lógicos y de comparación. Pueden consultarse en la siguiente tabla:

Operador	Descripción
==	Igual que
===	Igual a valor y tipo
!=	Distinto que
!==	Distinto valor o distinto tipo
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
?	Operador ternario

5. Operadores de tipo. Permiten conocer si una variable u objeto son de un tipo u otro e incluso conocer cuál es. Son los siguientes:
 - a. `typeof`. Devuelve el tipo de una variable
 - b. `instanceof`. Devuelve `true` si un objeto es una instancia de un tipo concreto

2.6.1 COMPARACIÓN DE CADENAS TENIENDO EN CUENTA DIFERENCIAS LOCALES

En JavaScript se pueden comparar las cadenas de texto con los operadores habituales (igualdad (`==`), igualdad estricta (`===`), mayor, menor, etc.) utilizando el código Unicode de cada letra para decidir el criterio de ordenación. Este método de comparación básico convencional tiene el defecto de que es estrictamente binario, es decir, si se comparan dos cadenas iguales, pero una está definida con letras mayúsculas y la otra con minúsculas, el resultado daría como respuesta `false` cuando en realidad se podrían considerar iguales.

Una forma habitual de salvar esta limitación es convertir ambas cadenas de manera que se encuentren todas en minúsculas o todas en mayúsculas. Las funciones que llevan a cabo estas tareas en JavaScript son `toLowerCase` y `toUpperCase` respectivamente, ambas miembros de la clase `String`.

Así, a la hora de comparar las cadenas `"JavaScript"` con `"javascript"` se escribirá:

```
if ("JavaScript".toLowerCase() == "javascript".toLowerCase())
```

que convierte las frases a minúsculas antes de compararlas. O bien mediante:

```
if ("JavaScript".toUpperCase() == "javascript".toUpperCase())..
```

pasando ambas cadenas a caracteres en mayúscula.

Este tipo de comparaciones son las que más a menudo se utilizan y es muy común encontrárselas porque generalmente la distinción entre mayúsculas y minúsculas no resultará de interés.

Sin embargo, conviene conocer que, a la hora de ordenar los textos, **algunos idiomas tienen sus pequeñas sutilezas**. Por ejemplo, las palabras:

- `cote`
- `coté`
- `côte`
- `côté`

se ordenarían así en inglés o español, pero si se es francés (como esas mismas palabras), se esperarían ver ordenadas de este modo:

- `cote`
- `côte`
- `coté`
- `côté`

Debido a ello existe **una manera más específica a la hora de comparar cadenas de texto** que es utilizar el método **localeCompare** de la clase **String**. Éste compara la cadena actual con otra cadena usando para ello los ajustes de comparación específicos del lenguaje indicado como preferido en el navegador, y teniendo en cuenta las preferencias de cada lenguaje a la hora de ordenar. Este método devuelve un número negativo si la cadena actual es menor que con la que se compara, un 0 si son iguales o un número positivo en caso de ser mayor:

```
alert("cote".localeCompare("côte")); //Devuelve -1
```

También existen unas variantes específicas para el lenguaje actual del navegador a la hora de poner en mayúsculas y minúsculas las cadenas: **toLocaleLowerCase** y **toLocaleUpperCase** que funcionan con las de antes, pero teniendo en cuenta las particularidades del idioma actual. Salvo que se sepa de antemano que pueda haber problemas con el lenguaje, generalmente no se suelen utilizar estas variantes localizadas, pero conviene conocerlas.

2.7. ESTRUCTURAS DE DECISIÓN

2.7.1 ESTRUCTURAS CONDICIONALES SIMPLES

No todos los problemas pueden resolverse empleando estructuras secuenciales. Por supuesto que en un problema se combinan estructuras secuenciales y condicionales. Cuando se presenta la elección existe la opción de realizar una actividad o no realizarla. En una estructura **CONDICIONAL SIMPLE** por el camino del “falso” no hay actividades y por el camino del “verdadero” si hay actividades (operaciones, entradas y salidas, inclusive puede haber otras estructuras condicionales).

Ejemplo: Realizar la carga de una nota de un alumno. Mostrar un mensaje de aprobado si tiene una nota mayor o igual a 5:

```
<script>
  let nombre;
  let nota;
  nombre=prompt('Escriba nombre:', '');
  nota=prompt('Escriba su nota:', '');
  if (nota >= 5){
    alert(nombre + ' está aprobado con un ' + nota);
  }
</script>
```

Aparece la instrucción **if** en el lenguaje JavaScript. La condición debe ir entre paréntesis. Si la condición se verifica verdadera se ejecuta todas las instrucciones que se encuentran encerradas entre las llaves de apertura y cerrado seguidas al **if**. Hay que tener en cuenta que en la condición del **if** deben intervenir una variable, un operador relacional y otra variable o valor fijo.

Para comprobar que una variable no es nula no es necesario siquiera escribir expresiones como:

```
if (persona != null){}
```

ya que JavaScript considera los nulos como falsos, así que si en este ejemplo se escribiera simplemente el nombre de la variable, se conseguirá exactamente el mismo efecto:

```
if (persona){}
```

o, por ejemplo:

```
if (persona && persona.Rol == "Administrador"){}
```

De hecho, es muy habitual verlo escrito de esta forma si se examina código ajeno, ya que se utiliza mucho, por ejemplo, para comprobar las características que soporta un determinado navegador web.

2.7.2 ESTRUCTURAS CONDICIONALES COMPUESTAS

Cuando se presenta la elección existe la opción de realizar una actividad u otra. Es decir, existen actividades cuando la condición es **true** y otras cuando devuelve **false**. Lo más

importante que hay que tener en cuenta es que se realizan las actividades de la rama del **verdadero (true)** o las del **falso (false)**, **NUNCA** se realizan las actividades de las dos ramas. En una estructura condicional compuesta hay entradas, salidas, operaciones, tanto por la rama del verdadero como por la rama del falso.

Ejemplo: Realizar un programa que lea dos números distintos y muestre el mayor de ellos:

```
<script>
  let num1,num2;
  num1=prompt('Introduce el primer número:', '');
  num2=prompt('Introduce el segundo número:', '');
  num1=parseInt(num1);
  num2=parseInt(num2);
  if (num1 > num2)
    alert('el mayor es '+num1);
  else
    alert('el mayor es '+num2);
</script>
```

La función **prompt** retorna un **string** por lo que hay que convertirlo a entero cuando se quiera saber cuál de los dos valores es mayor numéricamente. Esta porción de código muestra una **estructura condicional compuesta** ya que se tienen actividades por la rama del **verdadero** y del **falso**. La estructura condicional compuesta tiene la siguiente codificación:

```
if (<condición>) {
  <Instruccion(es)>
}
else {
  <Instruccion(es)>
}
```

Es igual que la estructura condicional simple salvo que aparece la palabra clave “**else**” y posteriormente un bloque **{ }**. Si la condición del **if** es verdadera se ejecuta el bloque que aparece después de la condición, en caso que la condición resulte falsa se ejecuta la instrucción o bloque de instrucciones que indicamos después del **else**.

Pero no solo esta estructura funciona de forma dicotómica; también se puede preguntar por varias condiciones antes de llegar al **else** final (que tampoco es estrictamente necesario). Seguiría este formato:

```
if (<condición>){
  <Instruccion(es)>
}
else if (<otracondición>){
  <Instruccion(es)>
}
else{
  <Instruccion(es)>
}
```

2.7.3 ESTRUCTURAS CONDICIONALES ANIDADAS

Se dice que una estructura condicional es anidada cuando por la rama del verdadero o el falso de una estructura condicional hay otra estructura condicional.

Ejemplo: Confeccionar un programa que pida por teclado tres notas de un alumno, calcule el promedio e imprima alguno de estos mensajes:

- Si el promedio es ≥ 9 mostrar "Sobresaliente".
- Si el promedio es ≥ 7 y < 9 mostrar "Notable".
- Si el promedio es ≥ 5 y < 7 mostrar "Aprobado".
- Si el promedio es < 5 mostrar "Suspenso".

```
<script>
  let nota1, nota2, nota3;
  nota1=prompt('Introduce 1ra. nota:', '');
  nota2=prompt('Introduce 2da. nota:', '');
  nota3=prompt('Introduce 3ra. nota:', '');
  //Se convierten los 3 string en enteros
  nota1=parseInt(nota1);
  nota2=parseInt(nota2);
  nota3=parseInt(nota3);
  let pro;
  pro = (nota1 + nota2 + nota3) / 3;
  if (pro >= 9){
    alert('Sobresaliente');
  }
  else{
    if (pro>=7)
      alert('Notable');
    else{
      if (pro>=5)
        alert('Aprobado');
      else
        alert('Suspenso');
    }
  }
}</script>
```

Se introducen tres strings por teclado que representan las notas de un alumno, se transforman a variables enteras y se obtiene el promedio sumando los tres valores y dividiendo por 3 dicho resultado. Se pregunta si el promedio es superior o igual a 9, en caso afirmativo por la rama del verdadero de la estructura condicional se muestra un mensaje que indique 'Sobresaliente'.

En caso que la condición dé falso, por la rama del falso aparece otra estructura condicional, porque todavía hay que averiguar si el promedio del alumno es superior/igual a 7 o inferior. El proceso se repite para las notas mayores de 5 finalizando con 'Suspenso' en las notas inferiores a 5.

En cualquier caso, la solución anidada no es la más óptima en algunos casos. Por ejemplo, el anterior código se puede optimizar utilizando una estructura if.. else if .. else.

El código modificado sería el siguiente:

```
if (pro >= 9)
    alert('Sobresaliente');
else if (pro>=7)
    alert('Notable');
else if (pro>=5)
    alert('Aprobado');
else
    alert('Suspenso');
```

2.7.4 SWITCH

La instrucción switch es una alternativa para remplazar los if-else if. Se puede aplicar en ciertas situaciones donde la condición se verifica si es igual a cierto valor.

Confeccionar un programa que solicite un valor entre 1 y 5. Luego mostrar con texto, el valor del número.

```
<script>
    let valor;
    valor = prompt('Ingrese un valor comprendido entre 1 y 5:', '');

    //Se convierte a entero
    valor = parseInt(valor);
    switch (valor){
        case 1:
            alert('uno');
            break;
        case 2:
            alert('dos');
            break;
        case 3:
            alert('tres');
            break;
        case 4:
            alert('cuatro');
            break;
        case 5:
            alert('cinco');
            break;
        default:
            alert('debe ingresar un valor comprendido entre 1 y 5.');
```

La variable que se analiza debe ir después de la instrucción switch entre paréntesis. Cada valor analizado debe ir después de la palabra clave 'case', seguida de ":", y las instrucciones a ejecutar en el caso de que se cumpla la condición.

Es importante disponer la palabra clave 'break' al finalizar cada caso. Las instrucciones que hay después de la palabra clave 'default' se ejecutan en caso que la variable no se verifique en algún case. De todos modos, el default es opcional en esta instrucción.

Al igual que en otros muchos lenguajes (incluyendo C#, C++ o Java) **esta estructura es bastante limitada**, en el sentido de que no deja comprobar condiciones más elaboradas.

Por ejemplo, no es posible:

- Comprobar varios casos separándolos por comas: "case 0, 1, 2, 3:" Habría que usar varios case separados en varias líneas, sin usar el break al final.
- Comprobar condiciones complejas basadas en rangos y operadores. Por ejemplo, esto no es válido: "case x>0 && x<10:" para que se ejecute un código con cualquier número entre 0 y 10.

Ante estas limitaciones, sobre todo la segunda, a veces la instrucción se convierte en algo bastante limitado y se debe rechazar su uso. Sin embargo, **existe un truco poco conocido** que es realmente útil y permite crear instrucciones switch mucho más complejas. Se trata de **comprobar un valor booleano en lugar de constatar directamente el valor de la variable** o expresión que se va a verificar. No es muy intuitivo, pero funciona bien.

Por ejemplo, supóngase que se necesitan ejecutar acciones diferentes en función de que una variable "x" esté entre 0 y 10, entre 0 y 20, sea menor que 0 o mayor que 100, y el resto de los posibles casos. Con una instrucción **switch** normal no es posible, pero si se sustituye por esto:

```
switch (true) {  
  case x > 0 && x < 10:  
    alert("x > 0 && x < 10");  
    break;  
  case x > 0 && x < 20:  
    alert("x > 0 && x < 20");  
    break;  
  case x < 0 || x > 100:  
    alert("x < 0 || x > 100");  
    break;  
  default:  
    alert("Ninguno");  
}
```

¡Es posible conseguir exactamente lo que se necesita!

¿Qué se ha hecho? En lugar de comprobar el valor de "x", se comprueba true. Esto hará que salte la coincidencia cuando alguna de las condiciones del case sea verdadera, saltándose las demás. En la práctica se está consiguiendo lo buscado, algo que a priori era imposible.

Hay que tener en cuenta que:

1. Las condiciones pueden ser lo complejas que se quieran mientras devuelvan un booleano o un valor verdadero/falso (se verá a continuación).
2. Hay que tener cuidado de poner, o bien muchas condiciones que sean mutuamente excluyentes, o bien colocarlas en el orden más apropiado en el que se quieran detectar. Si no se hace así pueden saltar casos de manera inesperada.
3. Es importante poner siempre el **break** al final de cada caso o saltarán todos los siguientes al primero que se detecte. Esto no es algo que se quiera normalmente, aunque quizá haya ocasiones en las que pueda ser útil.

2.7.5 VALORES PSEUDO-BOOLEANOS: TRULY Y FALSY

En JavaScript existen los valores booleanos (`true` y `false`), pero **existen también otros valores que usados en el contexto de un operador booleano se comportan también como si fueran booleanos**.

A los tipos de datos no booleanos que se interpretan como verdadero cuando se evalúan se les denomina valores **"truly"** (o como se les podría llamar en español: valores "verdaderos"). Cuando se evalúan como falsos se les denomina valores **"falsy"** (o valores "falsos").

Por ejemplo, considérese este código:

```
let v = "cualquier cosa";
if (v)
    alert("verdadero");
else
    alert("falso");
```

¿Cuál será el resultado que muestra el mensaje?

En este caso el resultado será ver por pantalla "verdadero". El motivo es que dentro de la expresión a evaluar en el condicional se encuentra una cadena de texto y ésta se convierte automáticamente a un booleano para poder devolver un resultado y ejecutar la rama pertinente del condicional. Como es un valor "verdadero", el resultado de convertirlo es `true`. El [apartado 14.6 de la especificación ECMAScript](#) indica que se debe llamar al método interno `ToBoolean()` sobre el resultado de evaluar la expresión que hay dentro de los paréntesis del `if`. Por ello **se convierte automáticamente la cadena en un booleano**.

Las reglas de conversión se indican en la [especificación ECMAScript](#), dependen del tipo de dato, y son las que nos proporcionan los valores "verdaderos" y "falsos":

- **undefined**: se interpreta como falso. Un miembro de un objeto que no existe, por ejemplo.
- **Null**: sería falso también.
- **NaN**: falso
- **Un número**: si es 0 se interpreta como falso, siendo verdadero para cualquier otro valor.

- **Cadenas de texto:** si la cadena está vacía (""), entonces es falso. Cualquier otra cadena se interpreta como un verdadero.
- **Un objeto** cualquiera siempre se interpreta como true.

Es decir, son valores:

- **falsosos o falsy:** el número 0, las cadenas vacías, los resultados de operaciones no válidas (NaN) los valores nulos y los valores no definidos.
- **verdaderos o truly:** cualquier otra cosa (los números distintos de cero, las cadenas no vacías, cualquier objeto...)

Es gracias a esto que, por ejemplo, se puede verificar la existencia de ciertas características en un navegador escribiendo cosas como:

```
if (window.Worker)
    alert("Este navegador soporta Web Workers");
else
    alert("¡Navegador viejo!! Actualízate!!");
```

Si la ventana tiene el objeto [Worker](#) lo devuelve y al convertirlo en booleano, según las reglas anteriores, el resultado del condicional será true. Si no existe, devolverá **undefined**, y por lo tanto se interpretará como un false.

Ahora cabe preguntarse lo siguiente: ¿Cuándo se interpreta un valor como booleano? Bueno, obviamente cuando ya es booleano, claro, pero **la conversión implícita** que se ha visto de valores "verdaderos" o "falsosos" en verdaderos booleanos **sólo se da cuando se evalúa el valor dentro de cierto contexto**. En concreto cuando se evalúa como resultado de una expresión en un condicional (se ha visto más arriba), pero también cuando se utiliza con un operador booleano ([apartado 13.13 de la especificación ECMAScript](#)). Y este detalle es muy importante porque si no se tiene en cuenta, se pueden acabar cometiendo errores difíciles de detectar.

Por ejemplo, si se varía ligeramente el código del condicional anterior y se escribe una comparación en la expresión dentro del if:

```
let v = "cualquier cosa";
if (v == true)
    alert("verdadero");
else
    alert("falso");
```

Aparentemente no ha cambiado demasiado. Cabría pensar que como v contiene una cadena no vacía (un valor *verdadero*) el condicional sigue siendo cierto. Sin embargo, lo que se verá por pantalla será "falso". El motivo es que ahora lo que se está haciendo es comparar el valor de la variable v, que es una cadena, con un booleano para ver si son iguales. Y no lo son. Por lo tanto, el resultado de la expresión es false. En este caso no se hace una conversión a booleano de la cadena, aunque esté dentro de un condicional. Lo que se convierte siempre a booleano es el resultado de la expresión dentro del if, pero no cada elemento de esta. Importante diferencia.

Sin embargo, se va a cambiarla nuevamente de manera sutil. ¿Qué pasaría si se utilizara un operador booleano (AND, OR...) dentro de la expresión?:

```
let v = "cualquier cosa";  
if (v && true)  
    alert("verdadero");  
else  
    alert("falso");
```

Lo que se ha hecho es comparar el valor v con true usando un operador AND (&&). Según la lógica es equivalente a comprobar la igualdad, ya que el AND solo devuelve true si ambas partes son true. En este caso, sin embargo, se mostraría por pantalla la palabra "verdadero" y no "falso" como en el anterior. ¿Por qué? El motivo es que ahora el valor de la variable v se convierte a booleano antes de hacer la operación, ya que está en el contexto de un operador lógico booleano.

Ahora se va a dar una vuelta de tuerca más al asunto, con algo que es bastante sorprendente. ¿Qué devolverá por pantalla esta expresión?:

```
alert("cualquier cosa" && true);
```

Efectivamente, aparece "true" por pantalla pues la expresión se evalúa como verdadera al ser ambos valores "verdaderos". Si se varía ligeramente la expresión:

```
alert(true && "cualquier cosa");
```

Es exactamente la misma comparación, así que debería verse lo mismo ¿no? Pues no, en este caso se verá por pantalla la frase "cualquier cosa" y no un "true" aunque el resultado de la operación es un verdadero al ser ambos valores "verdaderos". ¿Qué ha pasado aquí? Para comprenderlo hay que ver bien lo que pone la especificación acerca de cómo [evaluar operadores lógicos booleanos](#).

En el caso del operador AND (&&), la especificación dice que se evalúa de la siguiente manera (resumiendo los pasos en una sola frase):

- Si el valor convertido a booleano del primer operando es false, entonces devolver el primer operando, sino devolver el segundo operando.

En el caso del operador lógico OR (||) la regla es parecida:

- Si el valor convertido a booleano del primer operando es true, entonces devolver el primer operando, sino devolver el segundo operando.

Así de sencillo, pero con muchas implicaciones:

1. En primer lugar, **los operadores lógicos AND y OR no devuelven siempre un booleano**, como casi todo el mundo piensa. Devuelven el valor del primer o del segundo operando dependiendo de si son "verdaderos" o "falsos". Solo retornan un booleano si el operando que devuelven lo es.
2. **Se hace cortocircuito de expresiones**, es decir, que llega con evaluar el primero si con eso ya sabemos cuál va a ser el resultado. Así, en un AND, si el primero es false ya no hace falta seguir evaluando pues tendrían que ser los dos true y ya es imposible. En el

caso de OR si el primero es true ya no hace falta seguir evaluando porque si uno de ellos es true el resultado será true.

Al fijarse en las expresiones anteriores a la luz de estas reglas, entonces se ve que tienen toda la lógica del mundo. En el primer caso ("cualquier cosa" && true) el primer operador es verdadero (verdadero), así que se devuelve directamente el segundo. Al darles la vuelta (true && "cualquier cosa"), pasa lo mismo y se devuelve directamente el segundo argumento, que en este caso es una cadena.

Es fácil ver que toda esta casuística es muy peligrosa cuando se trabaja con valores que no son verdaderos booleanos, es decir, que se pueden interpretar como tales o no dependiendo del caso y de dónde se usen. Para solucionarlo se puede usar una técnica muy sencilla que consiste en asegurarse de devolver siempre todos los valores de las funciones que deban ser booleanos convertidos precisamente en booleanos. ¿Y cómo convertirlos? El operador **ToBoolean** es interno y no es posible llamarlo, así que se puede hacer de dos maneras: usando un objeto de tipo Boolean o **utilizar el operador de negación dos veces**.

La primera técnica, que es farragosa e ineficiente, implica crear un objeto a partir de un tipo primitivo:

```
let b = new Boolean("cualquier cosa");  
alert(b);
```

La segunda es mucho más elegante y concisa; se basa en que el uso del operador negación (!) convierte cualquier valor en booleano y lo niega, devolviendo un verdadero booleano. Si se aplica dos veces, se vuelve a negar obteniendo el valor original convertido en booleano:

```
let b = !!"cualquier cosa";  
alert(b);
```

Así, si hay que asegurarse de que las expresiones anteriores devuelven un booleano siempre, se pueden escribir así. Por ejemplo, el comparador entre una cadena y un booleano dentro de un if, se puede transformar así:

```
let v = "cualquier cosa";  
if (!!v == true)  
    alert("verdadero");  
else  
    alert("falso");
```

y funcionará como un booleano de verdad, devolviendo un "verdadero" como se esperaba. Sin embargo, no es necesario aplicarlo a los operadores de la condición un if cuando se usan con operadores lógicos booleanos:

```
let v = "cualquier cosa";  
if (!!v && true) //Esto no sirve para nada  
    alert("verdadero");  
else  
    alert("falso");
```


Ya que como se ha visto, el operador booleano ya fuerza la conversión/tratamiento como booleano de `v`. Esto se considera una mala práctica porque no aporta absolutamente nada y aumenta la ineficiencia del código.

Una buena práctica, sin embargo, consiste en asegurarse de que lo que devuelve una función es un booleano cuando es el tipo que se espera obtener como resultado.

Para ello se usa el operador negación dos veces (`!!`):

```
return !!res;
```

De este modo, aunque los datos originales no sean booleanos, la función va a devolver siempre uno. Esto es especialmente importante cuando se trabaja con elementos del DOM, que en un momento dado pueden no existir o carecer de alguna propiedad que se está utilizando en el código. De hecho, bibliotecas como jQuery hacen esto constantemente para devolver valores en sus funciones.

Un último ejemplo. Dado el código siguiente:

```
console.log([1,2] + ![]);
```

¿qué va a mostrar por consola?

El resultado sería `1,2false`. ¿Por qué? En primer lugar, cuando se muestra un array (se verán posteriormente) por consola o pantalla, se visualizan sus elementos separados por comas. A esta cadena se le une la negación de un array vacío. Aunque pudiera parecer que un array vacío devolvería `false`, la propia existencia del array hace que sea un `truly`, es decir, se traduciría a `true`, que una vez negado, quedaría en `false`. Por lo tanto, el resultado será: `1,2false`

2.8 BUCLES. ANIDADO. SENTENCIAS DE RUPTURA Y CONTINUACIÓN

2.8.1 BUCLE WHILE

Una estructura repetitiva o bucle permite ejecutar una instrucción o un conjunto de instrucciones varias veces. Una ejecución repetitiva de sentencias se caracteriza por:

- La o las sentencias que se repiten.
- El test o prueba de condición antes de cada repetición, que motivará que se repitan o no las sentencias.

Funcionamiento del **while**: En primer lugar, se verifica la condición, si la misma resulta verdadera se ejecutan las operaciones que se indican entre las llaves que le siguen al **while**. En caso que la condición sea **Falsa** continúa con la instrucción siguiente al bloque de llaves. El bloque se repite **MIENTRAS** la condición sea **Verdadera**.

Si la condición siempre retorna verdadero se está en un ciclo repetitivo infinito. Dicha situación es un error de programación, nunca finalizará el programa.

Ejemplo: Realizar un programa que imprima en pantalla los números del 1 al 100.

Se puede resolver el problema empleando una estructura secuencial. Se inicializa una variable con el valor 1, luego se imprime la variable, se incrementa nuevamente la variable y así sucesivamente.

```
<script>
  let x;
  x = 1;
  while (x <= 100) {
    document.write(x);
    x++;
  }
</script>
```

La primera operación inicializa la variable *x* en 1, seguidamente comienza la estructura repetitiva *while* y se dispone la siguiente condición (*x <= 100*), que implica que se lee **MIENTRAS** la variable *x* sea menor o igual a 100.

Al ejecutarse la condición, retorna **VERDADERO**, porque el contenido de *x* (1) es menor o igual a 100. Al ser la condición verdadera se ejecuta el bloque de instrucciones que contiene la estructura *while*. El bloque de instrucciones contiene dos salidas al documento y una operación. Se imprime el contenido de *x* y seguidamente se incrementa la variable *x* en uno.

Se incrementa *x* en 1 y al finalizar el bloque de instrucciones que contiene la estructura repetitiva, se verifica nuevamente la condición de la estructura repetitiva y se repite el proceso explicado anteriormente.

Mientras la condición retorne verdadero, se ejecuta el bloque de instrucciones; al retornar falso la verificación de la condición, se sale de la estructura repetitiva y continúa el algoritmo, en este caso, finaliza el programa.

Lo más difícil es la definición de la condición de la estructura `while` y qué bloque de instrucciones se va a repetir. Observar que si, por ejemplo, se dispone la condición `x >= 100` (si `x` es mayor o igual a 100) no provoca ningún error sintáctico, pero aparece un error lógico porque al evaluarse por primera vez la condición retorna falso y no se ejecuta el bloque de instrucciones que se quería repetir 100 veces.

Una vez planteado el programa hay que verificar si el mismo es una solución válida al problema (en este caso se deben imprimir los números del 1 al 100 en la página), para ello se puede hacer un seguimiento del flujo del diagrama y los valores que toman las variables a lo largo de la ejecución:

x
1
2
3
4
.
.
100

101 Cuando `x` vale 101 la condición de la estructura repetitiva retorna falso, en este caso finaliza el diagrama.

La variable `x` recibe el nombre de **CONTADOR**. Un contador es un tipo especial de variable que se incrementa o decrementa con valores constantes durante la ejecución del programa. El contador `x` indica en cada momento la cantidad de valores impresos en la página.

La variable `x` debe estar inicializada con algún valor antes que se ejecute la operación `x++`.

2.8.2 BUCLE DO/WHILE

La sentencia **do/while** es otra estructura repetitiva, la cual ejecuta al menos una vez su bloque repetitivo, a diferencia del **while** que puede no ejecutar el bloque. Esta estructura repetitiva se utiliza cuando se conoce de antemano que por lo menos una vez se ejecutará el bloque repetitivo.

La condición de la estructura está abajo del bloque a repetir, a diferencia del **while** que está en la parte superior.

Finaliza la ejecución del bloque repetitivo cuando la condición retorna falso, es decir igual que el **while**.

Problema: Escribir un programa que solicite la carga de un número entre 0 y 999, y muestre un mensaje de cuántos dígitos tiene el mismo. Finalizar el programa cuando se cargue el valor 0 o un valor no válido.

```
<script>
  let valor;
  do {
    valor = prompt('Ingrese un valor entre 0 y 999:', '');
    valor = parseInt(valor);
    if (valor < 0 || valor > 999 || isNaN(valor))
    {
      document.write('El número ', valor, ' no está en el rango elegido (0,999).  
El programa terminará.');
```

En este problema al menos se carga un valor. Si se carga un valor menor a 10 se trata de un número de una cifra, si es mayor a 10 pero menor a 100 se trata de un valor de dos dígitos, en caso contrario se trata de un valor de tres dígitos. Este bloque se repite mientras se ingresa en la variable 'valor' un número mayor que cero y menor o igual que 999. Además, controla que no se hayan introducido números negativos o nulos.

2.8.3 BUCLE FOR

Cualquier problema que requiera una estructura repetitiva se puede resolver empleando la estructura **while**. Pero hay otra estructura repetitiva cuyo planteo es más sencillo en ciertas situaciones.

Esta estructura se emplea en aquellas situaciones en las cuales **SE CONOCE** la cantidad de veces que se quiere que se ejecute el bloque de instrucciones. Ejemplo: cargar 10 números, ingresar 5 notas de alumnos, etc. Se conoce de antemano la cantidad de veces que se quiere que el bloque se repita.

Por último, hay que decir que la ejecución de la sentencia **break** dentro de cualquier parte del bucle provoca la salida inmediata del mismo. Sintaxis básica:

```
for (<Inicialización> ; <Condición> ; <Incremento o Decremento>) {  
    <Instrucciones>  
}
```

Esta estructura repetitiva tiene tres argumentos: variable de inicialización, condición y variable de incremento o decremento. Se utiliza generalmente cuando se sabe la cantidad de veces que se desea que se repita el bloque.

Ejemplo: Mostrar por pantalla los números del 1 al 10.

```
<script>  
    let f;  
    for (f = 1; f <= 10; f++) {  
        document.write(f + " ");  
    }  
</script>
```

Inicialmente f se la inicializa con 1. Como la condición se verifica como verdadera se ejecuta el bloque del for (en este caso se muestra el contenido de la variable f y un espacio en blanco). Luego de ejecutar el bloque pasa al tercer argumento del for (en este caso con el operador ++ se incrementa en uno el contenido de la variable f, existe otro operador -- que disminuye en uno una variable), hubiera sido lo mismo poner f+=1 pero este otro operador matemático ya simplifica las cosas.

2.8.4 USO AVANZADO DE BUCLES DETERMINADOS

Una cuestión poco conocida es que los bucles determinados en JavaScript pueden ofrecer una sintaxis más compleja que la convencional, ya que las tres partes de la estructura (contador, condición de parada e incremento) pueden usarse de varias maneras diferentes. A continuación, se verá cada una de las tres partes:

Contador

Normalmente en el contador se inicializa la variable de conteo del bucle. En realidad, esta parte es opcional y es posible omitirla. Por ejemplo, este bucle es perfectamente válido:

```
let i = 0;  
let res = "";  
for (; i < 10; i++){  
    res += i  
}
```

Se ha omitido la declaración de la variable usando una que ya está definida fuera del bucle. También se puede hacer todo lo contrario: inicializar varias variables en lugar de una sola, separándolas con comas. Por ejemplo:

```
let res = "";
for (let i = 0, j = 10; i < j; i++){
  res += i
}
```

Hay que fijarse en cómo se declaran dos variables, *i* y *j*, y luego se usan para hacer la comparación de la parte de la condición de parada del bucle. Se pueden inicializar ahí tantas variables como se quieran, cada una de un tipo diferente incluso, del mismo modo que si se hiciera fuera del bucle.

Condición

La segunda parte es la condición de parada y puede ser cualquier cosa que devuelva `true` o `false`. Es decir, se pueden crear condiciones todo lo complejas que se quieran, llamar a otras funciones, etc.

Un hecho menos conocido es que se puede directamente omitir esta condición. Si se hace así, el bucle se ejecutará indefinidamente (es decir, será un bucle infinito) por lo que hay que asegurarse de llamar a `break` dentro de éste en algún momento para poder salir. Por ejemplo:

```
for (let i = 0; ; i++){
  res += i
  if (i > 9) break;
}
```

Obsérvese que no se indica nada entre la variable de contador y la parte de incremento del final.

Al igual que se puede llamar a `break` dentro de un bucle `for`, también se puede utilizar `continue` si se desea saltar a la siguiente iteración del bucle.

```
for (let i = 0; ; i++){
  res += i
  if (i % 3 == 0)
    continue;
  console.log(i);
  if (i > 9) break;
}
```

En este ejemplo, solo se mostrarán los números que no sean divisibles por 3, ya que cuando se llegue al `continue` pasará a la siguiente iteración sin evaluar el resto del bloque.

Incremento

La tercera parte, el incremento, también es muy flexible. Por supuesto, es posible variar de la forma que se desee la variable de conteo, usando expresiones todo lo complejas posibles, llamando a otras funciones, etc.

Al igual que las anteriores, esta parte se puede omitir por completo, siempre y cuando se modifique el contador dentro del bucle o se asegure que éste converge.

Por ejemplo:

```
let res = "";
for (let i = 0; i<10;){
  res += i
  i++;
}
```

En este caso se aumenta el contador dentro del bucle, aunque podría hacerse cualquier otra cosa que fuerce la convergencia además de variar el contador: salir con un break o cambiar alguno de los parámetros de la condición de la segunda parte.

También es posible ejecutar varias instrucciones al mismo tiempo en esta tercera parte simplemente separándolas con comas. Por ejemplo, este bucle que lleva de todo:

```
for (let i = 0, j = 0; i<10 && j < 5; i++, j++){
  res += i
}
```

En este caso se declaran dos variables en el primer fragmento del bloque, se introduce una condición compleja que involucra a ambas en el segundo, y se varían los dos contadores al mismo tiempo en la tercera parte. Este bucle se ejecutará 5 veces.

Como puede verse, JavaScript es (casi) infinitamente flexible y "traga" con todo lo que le se le eche. Un último ejemplo extremo sería este:

```
var res = "";
var i = 0;
for (;;) {
  res += i
  i++;
  if (i>9) break;
}
```

Se trata de un bucle for que no especifica ninguna de sus tres partes constituyentes, pero que funciona sin problema y sería equivalente a un bucle básico que se ejecuta 10 veces.

2.8.5 BUCLE FOR...IN

El bucle **for...in** es una modificación del bucle **for** que permite explorar todas las propiedades de un objeto o los componentes de una colección. Al igual que en **for**, se usa una variable para iterar, solo que, si se trabaja con un objeto, la variable índice del bucle **for** estándar se sustituye por una variable de tipo cadena que contendrá el nombre de cada una de las propiedades del objeto manejado en el bucle **for...in**. Si se trabaja con matrices, la variable será numérica y contendrá la situación del componente respecto de la matriz. No se necesita la condición que se ha de evaluar para saber si se debe ejecutar el cuerpo puesto que lo hará siempre que existan propiedades distintas dentro del objeto o componentes aún no explorados en la matriz.

Sintaxis:

```
for (variable in [objeto|matriz])  
{  
    // Sentencias  
}
```

Ejemplo:

```
// Evaluar un objeto JavaScript  
let NombreObjeto, objeto;  
NombreObjeto = prompt("Introduzca el nombre de un objeto JavaScript","navigator");  
objeto = eval(NombreObjeto);  
for (let propiedad in objeto)  
{  
    document.write(NombreObjeto+"." + propiedad + ": " + objeto[propiedad] + "<br>");  
}
```


2.9 PRUEBA Y DOCUMENTACIÓN DE CÓDIGO

Las pruebas de software son investigaciones empíricas y técnicas cuyo objetivo es proporcionar información objetiva e independiente sobre la calidad del producto desarrollado al futuro usuario y/o comprador. Son una actividad más en el proceso de calidad. Básicamente, se trata de un conjunto de actividades como una etapa más dentro del desarrollo de software.

La prueba exhaustiva de software es impracticable ya que no se pueden probar todas las posibilidades de su funcionamiento ni siquiera en programas sencillos. Por ejemplo, en una aplicación simple de suma de dos números es imposible probar todos los números existentes a sumar. El objetivo de las pruebas no es asegurar la ausencia de defectos en un software, sino demostrar que existen.

Dado que el objetivo de las pruebas es detectar defectos en el software, el hallazgo de aquellos no implica que se sea mal profesional ya que es inherente al ser humano cometer errores. Es más, el descubrimiento de un defecto es un éxito en la mejora de calidad del producto.

A continuación, se verán una serie de conceptos importantes en el campo de las pruebas de software:

- Prueba (test). Actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas observando sus resultados y procediendo a su registro. Este proceso de evaluación abarca uno o varios aspectos concretos del software en desarrollo.
- Casos de prueba (test case). Conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular.
- Defecto (fault, bug). Defecto en el software, por ejemplo, definición de datos o paso de procesamiento incorrectos en un programa. En el ejemplo de la suma, el software es incapaz de cargar los sumandos.
- Fallo (failure). Incapacidad de un sistema o de alguno de sus componentes para realizar las funciones requeridas dentro de los requisitos de rendimiento especificados. Por ejemplo, cargar los datos, pero no sumarlos.
- Error (error). Diferencia entre un valor calculado, observado o medio y valor verdadero, especificado o teóricamente correcto. Por ejemplo, el resultado de la suma es incorrecto.

2.9.1 ESTRUCTURAS DE CONTROL DE ERRORES

Una estructura disponible en JavaScript (así como en otros lenguajes de programación) es try ... catch, resultando de gran utilidad para detectar errores y procesarlos de la forma deseada.

En caso de que ocurra algún error dentro del try se procesará la parte del catch().

En el siguiente ejemplo se generaría un error al procesar la suma (puesto que la variable 'y' no está definida) y se mostraría el mensaje especificado dentro del catch():

```
<script>
  let x = 33;
  try {
    x += y;
  }
  catch (e) {
    alert ("error: " + e.message);
  }
</script>
```

Observar que catch() recibe un objeto con información sobre el error ocurrido, pero dependiendo del navegador web usado contendrá unas propiedades u otras. Para mostrar un mensaje de error se usará la propiedad 'message', ya que es reconocida en todos los navegadores web.

Si se especifica una sección finally, su contenido se procesará en todo caso (ocurra un error o no):

```
<script>
  let x = 33;
  try {
    x += y;
  }
  catch (e) {
    alert ( "error: " + e.message );
  }
  finally {
    alert ("En finally");
  }
</script>
```

También, dentro de un bloque **try**, se puede lanzar una excepción, sin que tenga por qué haberse producido un error. Esto se hace con la sentencia **throw**.

```
<script>
  try {
    throw "miexcepcion"; //lanza una excepción
  }
  catch (e) {
    //tratamos la excepción alert(e);
  }
</script>
```

Este código lanza una excepción en el bloque **try**. Luego, en el bloque **catch**, se muestra la excepción que se ha detectado.

2.9.2 DOCUMENTACIÓN DEL CÓDIGO

La documentación del código constituye el conjunto de información que se basa en qué hace y cómo lo hace. Principalmente consiste en un material que explica las características de dicho código.

Esta documentación es un aspecto sumamente importante, tanto en lo que se refiere al desarrollo como al mantenimiento de una aplicación. A veces en desarrollo no se le da mucha importancia lo que hace que se pierda la posibilidad de reutilizar parte del programa en otras aplicaciones sin necesidad de conocer el código punto por punto.

La importancia de la documentación podría compararse con la importancia de tener una póliza de seguro; cuando todo va bien no se tiene la precaución de confirmar si está vigente o no. A la hora de desarrollar un código, es posible ayudarse de comentarios que se detallan sobre este mismo código para ofrecer documentación sobre los temas tratados. Se conoce como metainformación y su formato depende del lenguaje de programación o Framework utilizado.

Por ejemplo, en JavaScript se puede añadir metainformación al código utilizando comentarios especiales como los de [JSDoc](#). Se trata de un estándar ampliamente utilizado para documentar código JavaScript, el cual permite añadir descripciones detalladas de funciones, parámetros, tipos de datos, retornos, y más. En el siguiente ejemplo se puede ver cómo documentar una función que suma dos números:

```
/**
 * @summary Calcula la suma de dos números.
 * @description Esta función toma dos números como entrada y devuelve su suma
 * @param {number} x - Primer sumando
 * @param {number} y - Segundo sumando
 * @returns {number} Devuelve la suma de los dos números.
 */
function suma(a, b) {
  return a + b;
}
```

Si se introduce esta metainformación, con la herramienta JSDoc se puede generar documentación HTML basada en ella. Su uso es muy sencillo como puede verse en los siguientes pasos:

1. Se debe tener instalado Node.js
2. Instalar JSDoc como paquete Node:
npm install -g jsdoc
3. Ejecutar JSDoc dando un nombre de fichero y un directorio de destino:
jsdoc suma.js -d docs

Al término de la ejecución, se tendrán una serie de archivos HTML en el directorio docs. Abriendo el index se puede consultar toda la documentación generada.

BIBLIOGRAFÍA – WEBGRAFÍA

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