1. ESTRUCTURAS DE DECISIÓN

Inicialmente, el código JavaScript se ejecuta de izquierda a derecha y de arriba abajo según ha sido escrito, de forma secuencial.

Sin embargo, JavaScript incorpora estructuras de control que pueden modificar el flujo secuencial del programa. Entre estas estructuras de control podemos encontrar dos categorías: **estructuras de decisión** y **estructuras de repetición**. En el ejemplo siguiente utilizamos el cuadro de diálogo **prompt** para obtener el nombre del usuario.

```
<!DOCTYPE HTML>
<html>
<head>
<title>Cuadros de diálogo</title>
</head>
<body>
<script type="text/javascript">
<!--
prompt("Introduzca su nombre:", "anónimo");
//-->
</script>
</body>
</html>
```

Ahora vamos a introducir una estructura de decisión de forma que realizaremos una acción u otra en función de dicho nombre. La estructura de decisión clásica es:

```
if (condición)
   acciones1
else
   acciones2
```

Donde la cláusula **else** es opcional. Debes entender esta estructura del siguiente modo: se evalúa la **condición** y, si da como resultado el valor verdadero (*true*), entonces se ejecuta el conjunto de instrucciones **acciones1**; en caso contrario, es decir, que la condición se evalúe a falso (*false*), se lleva a cabo el conjunto de instrucciones **acciones2**.

```
<script type="text/javascript">
<!--
var respuesta;
respuesta = prompt("Introduzca su nombre:", "anónimo");
if (respuesta == "Juan")
   document.write("Bienvenido, " + respuesta);
else
      document.write("Acceso prohibido");
//-->
</script>
```

Veamos. Lo que estamos haciendo es muy sencillo: simplemente declaramos una variable, de nombre **respuesta**, en la que recogemos lo que el usuario ha introducido en el cuadro de diálogo **prompt**.

Fíjate que en la declaración de la variable no le hemos dado ningún valor inicial, ya que inmediatamente después la utilizamos para guardar el nombre introducido en el cuadro de diálogo **prompt**. Si no fuese así, deberías inicializarla para no correr el riesgo de utilizarla sin ningún valor.

Una vez tenemos el nombre del usuario en la variable **respuesta**, escribimos una estructura de decisión para realizar una u otra acción. Así pues, la condición `(respuesta == "Juan")` solo puede evaluarse a verdadero (true) o falso (false). En función de ello, se ejecutará la instrucción correspondiente.

Por otra parte, observa cómo se utiliza el operador `+` para concatenar dos cadenas de texto (la variable **respuesta** es una cadena de texto ya que este es el tipo de datos que devuelve el cuadro de diálogo **prompt**).

Es posible introducir una instrucción **if - else** en el interior de otra. Por ejemplo, en el código que hemos escrito sabemos perfectamente cuándo la condición se va a evaluar a verdadero, ya que solo puede haber un caso.

Sin embargo, puede evaluarse a falso en muchos otros casos: que introduzca un nombre distinto a *Juan*, que simplemente pulse en el botón *Cancelar*, que acepte el nombre dado por defecto (*anónimo*), etc.

Es posible que desees realizar una acción distinta en cada caso. Para ello, simplemente tienes que introducir otra estructura de decisión tras la palabra **else**:

```
if (respuesta == "Juan")
   document.write("Bienvenido, " + respuesta);
else
   if (respuesta == "anónimo")
     document.write("Por favor, introduzca su nombre.");
   else
     document.write("Acceso prohibido");
```

A la hora de crear estructuras anidadas **if - else**, debes recordar que la cláusula **else** siempre se relaciona con la cláusula **if** inmediatamente anterior y que aún no ha sido emparejada. De esta forma, no puede haber ninguna ambigüedad.

Esto es importante, ya que la indentación o inserción de tabulaciones que se utiliza para mejorar la lectura del código no afecta de ninguna manera.

Aunque anidar varias instrucciones **if - else** permite reducir el número de comprobaciones que se deben realizar, existe una estructura de decisión más apropiada cuando se deben tener en cuenta muchas condiciones, sobre todo para añadir claridad al código. Esta estructura es:

```
switch(ExpresiónDeComparación)
{
   case valor1:
     instrucciones1;
     [break];
   case valor2:
     instrucciones2;
     [break];
   ....

   case valorN;
     instruccionesN;
     [break];
   [default:
     instrucciones N+1;]
}
```

Donde los corchetes [ ] representan partes opcionales de la sintaxis.

Con esta estructura de decisión se evalúa una sola vez la expresión de compración y se ejecuta el bloque de instrucciones de la primera cláusula **case** cuyo valor cumpla con ella.

Es importante no olvidar introducir la instrucción **break** al final del bloque de instrucciones de cada **case**. Esto es así porque se ejecutará el grupo de instrucciones que aparece a partir de la primera cláusula **case** que cumpla la condición inicial hasta encontrar un **break**.

La última cláusula **default** se utiliza únicamente cuando ninguna de las cláusulas **case** anteriores ha tenido éxito.

En el ejemplo que estamos siguiendo tendríamos esta estructura **switch** equivalente:

```
switch(respuesta)
{
   case "Juan":
   case "José":
     document.write("Bienvenido, " + respuesta);
     break;
   case "anónimo":
     document.write("Por favor, introduzca su nombre.");
     break;
   default:
     document.write("Acceso prohibido.");
}
```

Fíjate que como el conjunto de instrucciones para el nombre "*Juan*" y para el nombre "*José*" es el mismo, no se ha incluido nada para el caso "*Juan*", por lo que se ejecutará el conjunto de instrucciones de "*José*".

2. EXPRESIONES LÓGICAS

Las condiciones también suelen llamarse **expresiones lógicas** porque solo pueden dar como resultado los valores verdadero o falso.

Puedes utilizar los operadores de comparación siguientes para crear expresiones de este tipo:

== igual a	**!= distinto a**
< menor que	**> mayor que**
<= menor o igual que	**>= mayor o igual que**
=== estrictamente igual a	**!== estrictamente distinto a**

La diferencia entre el operador **==** (igual a) y **===** (estrictamente igual a) radica en que, en este último caso, no solo se tiene en cuenta el valor de los operandos, sino que también tiene que coincidir el tipo de datos para que sean estrictamente iguales.

En el caso del operador **!=** (distinto a) y **!==** (estrictamente distinto a) es parecido, ya que para que la operación devuelva el valor verdadero, los operandos tienen que tener valores distintos y/o ser de distinto tipo.

Para entender la necesidad de estos últimos operandos, debes recordar que JavaScript realiza conversiones implícitas de tipo cuando así lo requiera la operación.

Por eso, es posible que evalúe como *"iguales"* dos valores de distintos tipo de datos. Si utilizamos los operadores "*estrictos*" esto no ocurrirá.

En lugar de utilizar un operador de comparación para construir una condición, es posible especificar simplemente el nombre de una variable u operando.

Por ejemplo:

```
if (nombreVariable)
{
   acciones1;
}
else
{
   acciones2;
}
```

En este caso, si la variable **nombreVariable** existe en el contexto del *script* y tiene un valor válido, la condición se evaluará a ***true***. En cambio, si no existe; o no se ha establecido un valor para **nombreVariable**; o tiene el valor **null** o la cadena vacía, entonces se evaluará a ***false***.

Este tipo de expresiones se utiliza mucho para detectar si el navegador admite una determinada funcionalidad.

Además de estos operadores, también puedes utilizar los operadores lógicos **AND**, **OR** y **NOT**.

En JavaScript el operador **AND** se representa con los símbolos **&&**, **OR** con || y **NOT** con **!**

- Así, una expresión del tipo **exp1 AND exp2** se evalúa a verdadero solo en el caso de que tanto **exp1** como **exp2** se evalúen a verdadero. En cualquier otro caso, la expresión se evaluará a falso.

- Sin embargo, **exp1 OR exp2** se evalúa a verdadero en el caso de que bien **exp1** o bien **exp2** se evalúen a verdadero (observa que es suficiente con que lo haga una de las dos o las dos al mismo tiempo).

  En el caso de que ninguna se evalúe a verdadero, la expresión total se evaluará a falso.

- Finalmente, **NOT exp1** se evalúa a verdadero si **exp1** se evalúa a falso; y se evalúa a falso si **exp1** se evalúa a verdadero. Es decir, realiza la negación de **exp1**.

Es importante que todos los paréntesis de apertura tengan su correspon–diente de cierre. Además, si utilizas los operadores lógicos, deberás introducir cada expresión entre paréntesis, como se ha hecho en el ejemplo.

Así pues, la expresión:

```
if ((respuesta == "Juan") || (respuesta == "José"))
```

se evaluará a verdadero si la variable **respuesta** tiene almacenado el valor "Juan" o "José". En cualquiera de los dos casos.

JavaScript es eficiente a la hora de evaluar expresiones con el operador **AND**, ya que la expresión total se evalúa a falso inmediatamente después de evaluar una condición a falso, sin necesidad de seguir evaluando el resto. Esto hará que su código sea más rápido.

¿Cómo se comparan las cadenas en JavaScript?

Cuando se comparan cadenas de caracteres, el primer carácter de la primera cadena es comparado con el primer carácter de la segunda cadena. Los segundos caracteres solo se evalúan si la comparación de los primeros se ha realizado con éxito. El proceso continúa hasta que todos los caracteres se comparan con éxito o hasta que uno falle. Esto permite utilizar los operadores `==`, `>`, `<`, etc. para establecer relaciones de comparación entre cadenas de texto.

Ten en cuenta que para estas comparaciones se tienen en cuenta si las letras aparecen en mayúsculas o minúsculas. Esto es significativo.

3. ESTRUCTURAS DE REPETICIÓN

Otro tipo de estructuras que puede modificar el flujo de ejecución son las estructuras de repetición o también conocidas como *bucles*. Estas estructuras sirven para repetir un conjunto de instrucciones.

En este sentido, puedes repetir un número determinado de veces el conjunto de instrucciones o, sin conocer dicho número, desear repetirlas mientras se cumpla cierta condición.

Esto quiere decir que existen dos tipos de estructuras de repetición: aquellas en que el número de repeticiones es conocido y aquellas en las que no lo es.

Utilizando la estructura **for**, especificas el número de veces que quieres que se repita un conjunto de instrucciones. Por lo tanto, ese número debe ser conocido.

La sintaxis de esta estructura de repetición es:

```
for (expresión; condición; operación)
{
   acciones
}
```

La expresión suele consistir en dar un valor inicial a una variable. Este valor se va comparando en cada una de las repeticiones según la condición establecida.

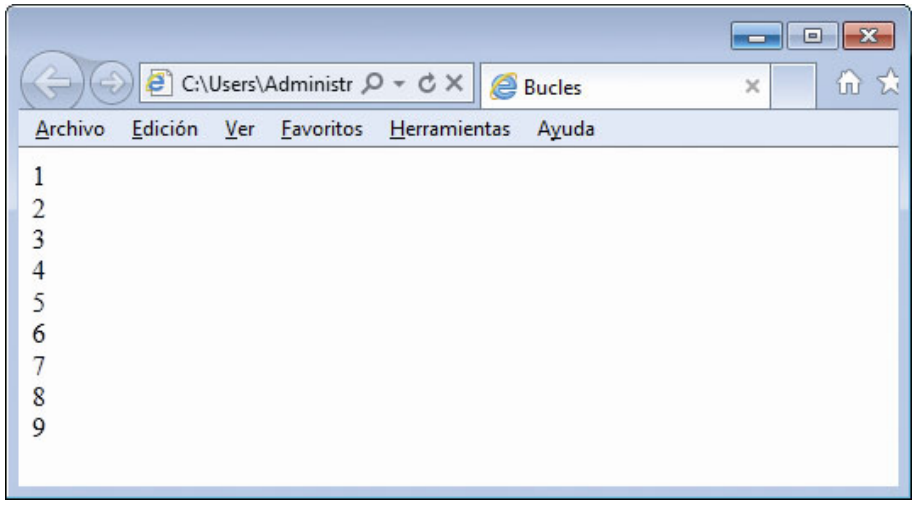
Por su parte, la operación permite acercar el valor inicial al valor final, es decir, en el cual ya no se realizarán más repeticiones.

La expresión i++ es una forma rápida de representar i = i + 1.

El conjunto de instrucciones o acciones que aparecen entre llaves **{ }** se llama cuerpo del bucle y son las que se ejecutan las veces que haga falta.

Fíjate en el código siguiente y su resultado en Internet Explorer:

```
<script type="text/javascript">
<!--
for (var i = 1; i < 10; i++)
{
   document.write(i);
   document.write("<br>");
}
//-->
</script>
```



Debes entender este código como sigue:

"Repetir las instrucciones que aparecen entre llaves las veces necesarias para que la variable i con valor inicial 1 llegue a valer 9, en saltos de uno en uno.".

Así, la variable **i** actúa como contador de las veces que se tiene que repetir el conjunto de instrucciones o cuerpo del bucle.

Finalmente, se utilizan las llaves **{** y **}** para crear un bloque de instrucciones, es decir, un conjunto de instrucciones que actúa como si fuera una sola. Es buena costumbre usar las llaves siempre, incluso cuando solo tengas una instrucción en el cuerpo del bucle.

Así, si no hubieses incluido las llaves en el ejemplo anterior, solo se hubiese repetido la primera instrucción, mientras que la segunda se hubiese ejecutado al finalizar el bucle.

Los bloques de instrucciones se pueden utilizar en muchos casos, no solo en estructuras de repetición.

El bucle finaliza cuando la variable **i** llega al valor **10**. En ese momento se sigue con la instrucción situada inmediatamente después del bucle.

¿Qué sucede, sin embargo, cuando no conocemos el número de veces que debe ejecutarse un bucle?

Por ejemplo, piensa en este escenario: pides al usuario que introduzca un número y seguidamente imprimes los números que van desde el 1 hasta dicho número. Observa cómo, de antemano, no sabes cuántas veces se va a repetir el bucle porque no conoces el número que introducirá el usuario.

Utilizando la estructura **while**, podrás repetir un conjunto de instrucciones *mientras* se cumpla cierta condición.

Esta es su sintaxis:

```
while (condición)
{
   acciones
}
```

Por ejemplo:

```
<script type="text/javascript">
<!--
var respuesta = prompt("Por favor, introduzca un
número:");
var i = 1;
while (i < respuesta)
{
   document.write(i);
   document.write("<br>");
   i++;
}
//-->
</script>
```

En este caso, lo que se hace es repetir las líneas que aparecen entre llaves *mientras* se cumpla la condición de que la variable **i** es menor que el valor dado por el usuario.

Fíjate que es necesaria la línea **i++** en el interior del bucle para que la variable **i** se acerque al valor en el cual la condición ya no se cumple. Si no fuese así, se podría crear un bucle infinito.

Por otra parte, como la condición se evalúa al principio del bucle, si esta no se cumple inicialmente, las instrucciones del cuerpo del bucle no llegarán a ejecutarse.

Además, debes iniciar la variable contador (**i** en este caso) antes del bucle. Es decir, que ya debe tener un valor para poder evaluar la condición.

En ocasiones, desearás salir de un bucle aunque la condición de prueba no se haya cumplido. En estos casos puedes utilizar la sentencia **break** si desea salir completamente del bucle o **continue** si lo que quiere es no ejecutar el resto de sentencias del cuerpo del bucle, sino pasar a la siguiente repetición.

Al igual que ocurría con las estructuras de decisión, puede anidar bucles en el interior de otros bucles. En estos casos, para cada repetición del bucle exterior, se ejecuta completamente el bucle interior.

Debes tener en cuenta que la incorporación de bucles en el interior de otros bucles puede hacer que la ejecución del código consuma mucho tiempo.

4. DEFINIR FUNCIONES

Cuando la tarea que debe realizar un *script* es compleja, es mejor dividir el problema en otros más pequeños, que sean más fáciles de manejar.

JavaScript permite descomponer un programa en tareas más pequeñas utilizando el concepto de **función**.

Una función no es más que un conjunto de instrucciones que realiza una tarea bien definida.

La misma idea la puedes encontrar en muchas situaciones cotidianas. Por ejemplo, cuando vas a cocinar una tortilla de patatas, seguro que divides el trabajo en tareas más concretas y sencillas: pelar las patatas, freír las patatas, batir los huevos, etc.

Cada una de estas tareas se correspondería con una función, mientras que cocinar la tortilla de patatas sería el *script* en sí.

Fíjate que es importante cada una de las tareas así como el orden en que se realizan.

Pero lo mejor de todo es que, si después deseas cocinar una tortilla de champiñones, seguro que alguna de las tareas que has realizado con la tortilla de patatas te sirve. Por ejemplo, batir los huevos.

Esto quiere decir que, al descomponer el programa en funciones, podrás utilizar esas funciones en más de un sitio y, por lo tanto, no tener que volver a escribir dicho código.

Para poder utilizar una función, es necesario definirla. Esto se hace con la sintaxis:

```
function Nombre(lista de parámetros)
{
     acciones
}
```

Es decir, la palabra clave **function** a la que le sigue un nombre válido de función y, entre paréntesis, una lista de parámetros. Después, entre llaves, aparece el cuerpo de la función; esto es, las instrucciones que la constituyen. Veamos un ejemplo:

```
function Validar(valor)
{
 if ((valor >= 1) && (valor <= 20))
      return true;
 else
      return false;
}
```

Volviendo al símil de la tortilla de patatas, es como si delegases la tarea de pelar patatas en uno de tus ayudantes (la función). Este ayudante necesitaría conocer cuántas patatas tiene que pelar (el parámetro).

La lista de parámetros sirve para comunicar la función con el resto del código del *script*. Por ejemplo, en este caso, la función necesita conocer el valor introducido por el usuario. Este valor se le da a partir del parámetro **value**.

El hecho de devolver un valor sería como si tu ayudante te avisara de que ya están peladas las patatas o que ha habido algún problema.

A su vez, es posible que la función devuelva un valor. En este caso, devolverá el valor booleano **true** (verdadero) si el valor es "correcto" (está en el rango permitido) o **false** (falso) si no lo es. Para devolver un valor, la función utiliza la instrucción **return**.

A tener en cuenta respecto de la definición de funciones:

- Para poder utilizar una función, es necesario definirla previamente.
- Una función puede devolver un valor al código que la utiliza a través de la instrucción **return**.
- Una función puede comunicarse con el código que la utiliza a través de los parámetros.
- Una función puede utilizar otra función para conseguir realizar su objetivo.

5. LLAMAR FUNCIONES

Al definir la función, lo único que estás indicando es lo que hace dicha función, es decir, su nombre, los parámetros que necesita y las instrucciones que llevará a cabo.

Sin embargo, la definición en sí no produce ningún resultado en la página web. Para poder utilizar la función, será necesario llamarla.

La forma de llamar una función es simplemente utilizando el nombre con el que se ha definido y, si es el caso, incluir los argumentos de la misma entre paréntesis, como en **`Validar(5)`**.

Si la función devuelve algún valor, podrás asignar dicho valor a alguna variable o utilizar la llamada a la función en alguna expresión en la que el valor devuelto sea correcto.

El ejemplo anterior es válido para comprobar esta situación. Deseas saber si el valor pasado a la función **Validar** es un número entre el 1 y el 20, por lo que la función devolverá *true* o *false* indicando esta circunstancia.

Lógicamente, si no guardas ese resultado en algún sitio, de poco te servirá que se haya ejecutado la función.

Obsérvalo en el siguiente código:

```
if (Validar(numero))
      document.write("Cuadrado: " + numero * numero);
else
      document.write("Valor fuera de rango.");
```

Recuerda que en la definición se habla de parámetros, es decir, aquellos datos que la función necesita para llevar a cabo sus tareas.

Ahora, al llamar la función, le damos esos datos de forma específica a través de los argumentos.

Lo importante es que el tipo de datos del parámetro y del argumento coincidan, o que, al menos, puedan ser convertibles. Además, si en la definición indicas que se necesitan 3 parámetros, en la llamada tienes que utilizar 3 argumentos. Veamos otro ejemplo.

Definición de la función:

```
function Cuadrado(numero)
{
 if (Validar(numero))
      document.write("Cuadrado: " + numero * numero);
 else
      document.write("Valor fuera de rango.");
}
```

Llamada a la función:

```
var respuesta = prompt("Por favor, introduzca un número
entre 0 y 20.", "0");
Cuadrado(respuesta);
```

Recuerda la diferencia entre definir la función y utilizarla o llamarla. Además, en el interior de una función se pueden hacer llamadas a otras funciones.

6. ÁMBITO DE LAS VARIABLES

En la definición de una función se puede utilizar cualquier sentencia de JavaScript, incluida la posibilidad de declarar variables.

En estos casos, se dice que la variable es **local** ya que solo puede ser conocida en el interior de la función.

A diferencia de las variables locales, las variables declaradas fuera de las funciones son **globales**.

Esto quiere decir que pueden ser conocidas y, por lo tanto, utilizadas, tanto dentro como fuera de las funciones.

Pero, ¿qué sucede si se declara una variable global y una variable local con el mismo nombre? Fíjate en el código siguiente:

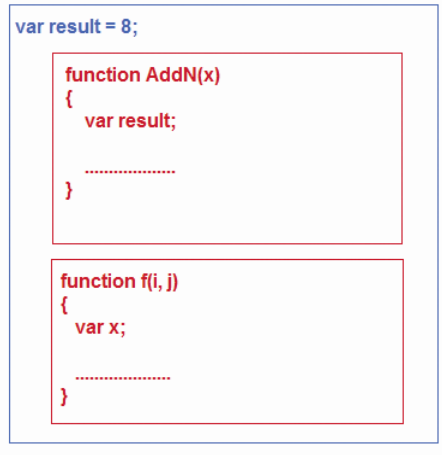
```
<!DOCTYPE HTML>
<html>
<head>
<title>Variables globales y locales</title>
</head>
<body>
<script type="text/javascript">
<!--
function AddN(x)
{
  var result = 0;
  for (var i = 1; i <= x; i++)
  {
     result = result + i;
  }
  return result;
}

var result = 5;
document.write("El resultado es: " + AddN(result));
//-->
</script>
</body>
</html>
```

Esta situación la tenemos en este sencillo *script*. Así, se ha declarado una variable de nombre **result** tanto en el interior de la función como a nivel del *script*. Es decir, una es local y la otra es global.

Dentro de la función siempre tiene preferencia la variable local respecto a la global. Por lo tanto, en el código asociado a la función **AddN**, cuando se utiliza el identificador **result**, se está refiriendo a la variable local que ha declarado en su interior.

Además, observa cómo el parámetro **x** actúa como una variable local más de la función. Por ello, podemos comparar el valor de **i** con el de **x**. En definitiva, el *script* lo único que hace es obtener el resultado de 1 + 2 + 3 + 4 + 5 = 15.

Observa la figura adjunta: la caja exterior representa el *script*, mientras que las cajas interiores son funciones definidas en el *script*.



Las funciones conocen todo lo que está fuera de ellas y lo que está en su interior, mientras que fuera de las funciones solo se pueden utilizar las variables declaradas a nivel de *script*.