Introducción a JavaScript

JavaScript es un lenguaje de programación de los denominados "lenguajes cliente", que se utiliza para dotar a los clientes (navegadores) de funcionalidades adicionales a la hora de manejar aplicaciones web. Algunas de estas funcionalidades permiten añadir contenido dinámicamente en una página (por ejemplo, añadir más apartados en un formulario), o validar los datos de un formulario antes de enviarlo, aunque esto también puede hacerse con HTML5. También, combinado con HTML5, permite hacer muchas cosas en el apartado gráfico. De hecho, se pueden hacer gran variedad de videojuegos simplemente utilizando estos dos lenguajes. Veremos en este tema algunas de las funcionalidades que nos puedan resultar más interesantes a la hora de desarrollar aplicaciones web.

1. Historia y versiones

La necesidad de un lenguaje como JavaScript surgió a principios de los 90, con las primeras páginas web complejas, que ya empezaban a usar formularios. El hecho de tener entonces velocidades de conexión muy lentas (módem) y enviar grandes cantidades de datos al servidor mediante formularios que luego podían ser incorrectos, hizo necesaria la aparición de un lenguaje que trabajara en el cliente, y pudiese dar validez a los datos antes de enviarlos inútilmente.

Sus orígenes parten del navegador Netscape 2.0, y un lenguaje que entonces se llamaba LiveScript. Posteriormente, Netscape se alió con Sun Microsystems y desarrollaron un nuevo lenguaje, ya presente en Netscape 3.0 bajo el nombre de JavaScript, por el auge que tenía en aquella época el lenguaje de programación Java, aunque ambos no tuvieran nada que ver. Por su parte, Microsoft sacó una copia de este lenguaje en su Internet Explorer, llamada JScript, y finalmente Netscape decidió estandarizar la suya (1997) para evitar problemas, a través del estándar ECMAScript.

JavaScript es, por tanto, una implementación de **ECMAScript**, el estándar que define las características de este lenguaje. Este estándar ha ido pasando por una serie de versiones sucesivas, que han ido incorporando nuevas funcionalidades al lenguaje. Podemos consultar en la web oficial la evolución de este estándar, y también conocer algo más sobre las características de cada versión en webs como Wikipedia. Hay que tener también presente que, dado que JavaScript es un lenguaje que se ejecuta en multitud de navegadores (Chrome, Firefox, Opera, Safari, Edge...), es posible que las últimas versiones de ECMAScript aún no sean compatibles con todos ellos, y por tanto, algunas de esas nuevas funcionalidades aún no estén muy extendidas.

2. Añadir JavaScript a las páginas

Al igual que ocurre con CSS, tenemos varias alternativas para añadir código JavaScript a nuestras páginas web. Lo veremos con un sencillo ejemplo que muestra un cuadro con un saludo en la pantalla.

2.1. Añadir código JavaScript en la propia página

La primera opción que veremos para añadir el código JavaScript es incluir una etiqueta `script` en cualquier parte de la página, y dentro el código JavaScript que queramos escribir. En nuestro caso, para mostrar una ventana con un saludo, usamos la instrucción `alert` y entre paréntesis y comillas (dobles o simples), el texto del saludo:

```
<html>
<head>
    ...
    <script>
        alert('Hola, buenas');
    </script>
    ...
</head>
...
```

2.2. Añadir código JavaScript en los elementos HTML

Esta opción es menos usual, porque deja todo el código más ilegible, pero en algunos casos (sobre todo con el uso de formularios) es una herramienta útil. Por ejemplo, si queremos que al hacer clic sobre un párrafo se muestre el cartel de saludo anterior, haríamos algo como esto:

```
<p onclick="alert('Hola, buenas');">Bla bla bla bla bla... </p>
```

En este caso, el texto del *alert* debe ir entre comillas simples, porque las comillas dobles se usan para el atributo *onclick*, que volveremos a ver con más detalle más adelante.

2.3. Añadir código JavaScript desde un fichero externo

Esta opción consiste en sacar el código JavaScript a un fichero de texto externo, como se hace con los estilos CSS también, y guardar este archivo con extensión *.js*. En este caso, nuestro archivo tendría el siguiente código:

```
alert('Hola, buenas');
```

Después, en la(s) página(s) web donde queramos añadir este código, incluimos una etiqueta `script`, que esta vez enlazará con el archivo JavaScript anteriormente creado (que deberá estar en alguna carpeta o subcarpeta de nuestra web), mediante un atributo *src*:

```
<html>
<head>
    ...
    <script src="fichero.js"></script>
    ...
</head>
...
```

2.4. Ubicación de las etiquetas *script*

Si utilizamos una etiqueta `script` para incluir el código JavaScript en nuestras páginas (tanto incorporado en la página como desde un fichero externo), conviene tener presente cuál es la mejor ubicación de dicho fichero.

- Si el fichero sólo contiene funciones que se van a activar o lanzar más adelante, a medida que interactuemos con la página, podemos colocar este *script* en la cabecera (*head*).

```
<html>
    <head>
        ...
        <script src="fichero.js"></script>
        ...
    </head>
...
```

- Si el fichero contiene instrucciones que necesitan ejecutarse inmediatamente cuando la página se cargue, entonces conviene incorporarlo al final de la página, cuando ya se haya cargado todo el contenido de la misma:

```
<html>
    <head>
        ...
    </head>
    <body>
        ...
        <script src="fichero.js"></script>
    </body>
</html>
...
```

2.5. La etiqueta *noscript*

JavaScript es una funcionalidad que puede ser deshabilitada intencionadamente en los navegadores, entre otras cosas, para prevenir ataques maliciosos. HTML proporciona la etiqueta `noscript` para detectar cuándo está desactivado o no disponible este lenguaje, y así prevenir al usuario de que debe activarlo para ejecutar la página correctamente.

```
<noscript>
    <p>Tienes JavaScript desactivado. Deberás activarlo para poder
        utilizar la página correctamente.</p>
</noscript>
```

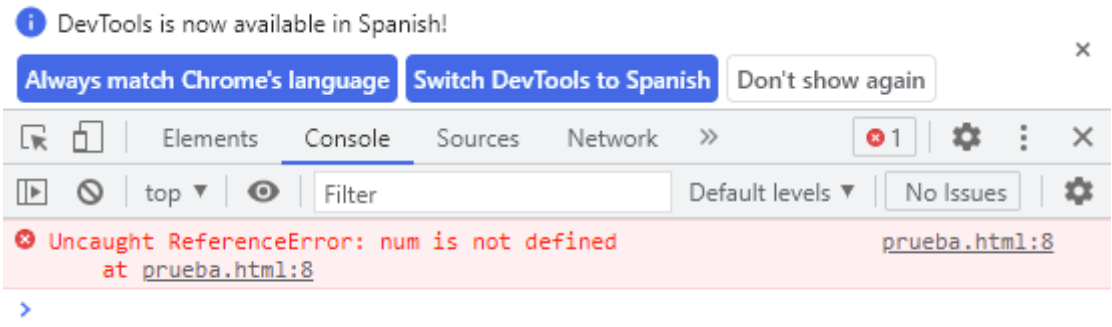
El contenido de la etiqueta sólo se mostrará si JavaScript está desactivado en el navegador.

3. Detectar errores en JavaScript

A diferencia de los lenguajes de programación de aplicaciones de escritorio, como Java o C#, en lenguajes de programación web como JavaScript o PHP es más difícil ver dónde está el error cuando una aplicación o página web no hace lo que debe, ya que no existe un terminal o consola accesible donde poder ver lo que el programa está haciendo.

La mayoría de navegadores proporcionan herramientas para poder depurar nuestro código JavaScript, y ver dónde están los errores cuando cargamos las páginas. Además, disponen de herramientas para revisar el código HTML y CSS y corregir posibles errores de diseño o estructura.

En Google Chrome, por ejemplo, con la tecla F12 (o haciendo clic derecho en la página y eligiendo la opción *Inspeccionar*) mostramos el panel de herramientas para desarrolladores. En él encontraremos varias secciones o pestañas. Una de las más útiles son la *Consola*, que muestra posibles errores en la ejecución del código JavaScript.



3.1. Consola y mensajes de error

Además de la instrucción `alert`, con la que podemos mostrar al usuario mensajes en la página a modo de *popups*, también podemos emplear la instrucción `console.log` o `console.error` para mostrar mensajes directamente en esta consola de depuración. Esto nos puede venir bien para dejar un registro de

mensajes durante la ejecució del programa, y también para informar de errores que se puedan producir y que hayamos detectado.

```
console.error("El email está vacío");
```

> **Ejercicio 1**:
>
> Crea un archivo llamado **saludo_js.html**. Define dentro, embebido, código JavaScript con un mensaje que te salude por tu nombre. Procura cometer algún error de sintaxis (por ejemplo, no cerrar los paréntesis) para ver cómo te informa de ello la consola del navegador.

4. Nociones básicas del lenguaje

Antes de entrar en algunas de las funcionalidades verdaderamente útiles para las aplicaciones web, es necesario conocer algunos de los fundamentos básicos del lenguaje Javascript, como el uso de variables y algunas estructuras de código útiles.

4.1. Tipos de datos básicos

JavaScript puede manejar 3 tipos básicos de información (además de información más compleja que iremos viendo más adelante).

- **Números**: que pueden ser enteros (*integer*) o reales (*float*), pero JavaScript no hace una distinción inicial entre ellos. Se consideran tipo *Number*.
- **Booleanos**: que pueden tomar los valores *true* o *false*. Tipo *Boolean*
- **Textos**: que se pueden representar con comillas simples o dobles, y son tipo *String*. También existen las secuencias de escape habituales en otros lenguajes para representar ciertos símbolos, como el salto de línea `\n`, la tabulación `\t`, o incluso las propias comillas dobles o simples, `\"` y `\'`, para representarlas dentro de un texto.

Normalmente utilizaremos variables para almacenar estos datos en un programa. Veremos a continuación cómo declararlas y utilizarlas.

4.2. Definición y uso de variables

Podemos ver una variable como un recipiente que sirve para almacenar información, de forma que esa información puede variar según el momento. Por ejemplo, podemos almacenar lo que hay guardado en un campo de un formulario, pero si luego el usuario modifica el valor de ese campo, el de la variable también podrá cambiar al nuevo valor.

Las variables en JavaScript se pueden definir con la palabra `var` o `let` seguida de un nombre de variable. Este nombre sólo puede tener letras, números, el carácter de subrayado (_) y el símbolo del dólar ($), y no puede empezar por número. Así, los siguientes nombres de variables serían válidos, salvo los dos últimos (el penúltimo empieza por número, y el último tiene un símbolo no permitido en el nombre):

```
var nombreUsuario;
let apellido1;
var $precio;
var 1apellido;
let nombre.usuario;
```

A las variables les podemos asignar un valor con el operador de asignación `=` y el valor que queramos asignarles. También podemos utilizar el mismo nombre de variable más adelante para hacer alguna operación. El siguiente ejemplo almacena un texto en una variable y luego lo muestra con la instrucción alert que hemos visto en ejemplos anteriores:

```
var texto = "Hola, buenas tardes.";
alert(texto);
```

Observa cómo cada instrucción (cada línea) del código Javascript termina en un punto y coma. Esto realmente no es necesario en muchos casos, pero se recomienda utilizarlo para separar cada instrucción de la siguiente.

Constantes

Podemos emplear la palabra `const` para declarar constantes, es decir, elementos que van a almacenar un valor que no va a cambiar durante la ejecución de un programa.

```
const pi = 3.1416;
```

undefined* y *null

Cuando declaramos una variable y no le damos valor, automáticamente se le asigna el valor `undefined`, que es un tipo de dato especial que se asigna a una variable cuando no tiene valor. No hay que confundirlo con el valor `null`, que también existe en JavaScript, y que es el valor por defecto que toman muchas variables en otros lenguajes. Sin embargo, en JavaScript una variable sólo es nula si explícitamente le damos ese valor.

```
var dato;
console.log(dato);      // undefined
dato = null;
console.log(dato);      // null
```

4.3. Comentarios

Al igual que ocurre con otros lenguajes, JavaScript permite añadir comentarios entre el código, que fundamentalmente tienen como propósito explicar o introducir el código que va a continuación.

En JavaScript, los comentarios pueden ser de dos formas, similares a otros lenguajes. Si queremos que ocupen más de una línea, los encerraremos entre el símbolo `/*` y el símbolo `*/`. Si queremos que ocupen una sola línea, también podemos usar este formato, o empezar el comentario con una doble barra `//`. Veamos un ejemplo:

```
/* Creamos una variable edad para
pedirle al usuario su edad */
var edad = prompt ("Dime tu edad");
// Pasamos esa edad a número
var edadNumerica = parseInt(edad);
```

4.4. Comunicación con el usuario

Ya hemos visto que con la instrucción `alert` podemos sacar mensajes por pantalla en un pequeño cuadro de diálogo, poniendo entre paréntesis el mensaje que queremos mostrar. Con las instrucciones `console.log` y `console.error` estos mensajes se muestran en la consola del navegador, que no está directamente visible o accesible por los usuarios:

```
alert("Hola, buenos días");
console.log("El email está vacío");
```

Alternativamente, también podemos emplear la instrucción `document.write`, que muestra la información directamente en la página HTML donde se ejecuta el código JavaScript:

```
document.write("Esto es un texto en el HTML");
```

Otra instrucción útil para comunicarnos de forma básica con el usuario es `prompt`. Al igual que en el caso anterior, entre paréntesis podemos mostrar un mensaje, pero con esta instrucción también podemos permitir que el usuario escriba algo (en un cuadro de texto), recoger lo que ha escrito y guardarlo en una variable, para luego poderlo procesar y utilizar.

```
let nombre = prompt("Dime tu nombre");
```

Sacaría un cuadro para que el usuario escribiese su nombre, y al pulsar en *Aceptar* quedaría guardado en la variable nombre. Si finalmente el usuario cancela, la variable queda con un valor nulo (*null*).

4.5. Operaciones básicas

Podemos hacer varios tipos de operaciones: aritméticas (sumar, restar...), relacionales (comparar un valor con otro) o lógicas (comprobar si se cumplen varias condiciones).

Operaciones aritméticas

En cuanto a las operaciones aritméticas, existen cinco operaciones básicas: `+` (suma), `-` (resta), `*` (multiplicación), `/` (división) y `%` (resto de división entera). Veamos un ejemplo:

```
var num1 = 2;
var num2 = 3;
var suma = num1 + num2;
alert (suma);
suma = suma * 2;
alert(suma);
```

El código anterior mostraría por pantalla el valor de la variable *suma*, que en este ejemplo es el resultado de sumar la variable *num1* (que vale 2) y la variable *num2* (que vale 3), por lo que mostraría por pantalla 5. Después, la variable suma se multiplica a ella misma por 2 y vuelve a mostrar su valor (con lo que ahora mostraría 10).

El operador `+` también se puede utilizar para enlazar textos (concatenar), de forma que podemos enlazar, por ejemplo, textos fijos y variables. Así, el mensaje anterior lo podríamos sacar con algo de texto más la variable suma:

```
alert("El resultado es " + suma);
```

También podemos emplear los operadores de *autoincremento* y *autodecremento* para aumentar o disminuir en una unidad el valor de una variable, como ocurre en otros muchos lenguajes.

```
var numero = 3;
numero++;
console.log(numero);     // 4
```

Operaciones relacionales

Estas operaciones sirven para comparar dos valores entre sí, y ver si, por ejemplo, uno es mayor que otro, o ambos son iguales. Los operadores relacionales son `>`, `>=` (mayor o igual), `<`, `<=` (menor o igual), `==` (igual a) y `!=` (distinto de). Veamos un ejemplo:

```
var num1 = 3;
alert (num1 > 2);
alert (num1 == 3);
alert (num1 <= 1);
```

Hemos creado una variable *num1* con el valor 3. Después, sacamos tres mensajes por pantalla. En el primero, comprobamos si la variable es mayor que dos (es cierto), en el segundo, vemos si la variable es igual a 3 (también nos dirá que es cierto), y en el tercero, comprobamos si la variable es menor o igual que 1 (nos dirá que es falso).

Podemos emplear estos operadores para comparar valores de cualquier tipo simple, incluyendo cadenas de texto, que se comparan alfabéticamente. Esto supone una ventaja importante frente a otros lenguajes como Java o C#, donde la comparación de textos es algo más compleja.

Adicionalmente, existen los operadores de **identidad**, que son `===` y `!==`. Estos operadores sirven para, además de comprobar si dos datos tienen el mismo valor, ver si son también del mismo tipo. Por ejemplo, los valores "7" y 7 almacenan el mismo valor, pero son de tipos distintos (uno es real y el otro entero).

Operaciones lógicas

Estas operaciones sirven en general para combinar varias comparaciones, aunque también se pueden usar para invertir el valor de una comprobación (si era falsa, pasa a ser cierta, y viceversa). Los operadores relacionales son `!` (operador NOT, para negar una condición y cambiar su valor entre verdadero y falso), `&&` (operador AND, para unir dos comprobaciones en una que será cierta si ambas lo son) y `||` (operador OR, para unir dos comprobaciones en una que será cierta si alguna lo es). Veamos un ejemplo:

```
var num1 = 3;
var num2 = 5;
var num3 = 4;
alert (!(num1 < num2));
alert (num3 > num2 && num3 > num1);
alert (num3 > num2 || num3 > num1);
```

Hemos definido 3 variables con valores diferentes. Después, mostramos 3 mensajes:

- En el primer mensaje que mostramos, tenemos una comparación (`num1 < num2`). Esta comparación es verdadera (3 < 5), pero como tenemos el operador NOT delante (`!`), la hacemos falsa, y el resultado final de la comprobación es FALSO.
- En el segundo mensaje realizamos dos comparaciones: `num3 > num2` (que es falsa, porque 4 < 5), y `num3 > num1` (que es verdadera, porque 4 > 3). Tenemos una comparación falsa y una verdadera, unidas por el operador AND (`&&`). Este operador sólo da un resultado verdadero cuando las dos comprobaciones que une son verdaderas, luego en este caso dará como resultado FALSO.

- En el tercer mensaje, realizamos las mismas dos comparaciones que en el caso anterior, pero unidas por el operador OR (`||`). Este operador da un resultado verdadero cuando alguna de las comprobaciones que une es verdadera (o las dos). Así que, en este caso, el conjunto es VERDADERO.

4.6. Conversiones entre tipos simples

En ocasiones nos puede interesar convertir un tipo simple en otro. Por ejemplo, convertir un texto a entero, o un número en texto. Para ello JavaScript dispone de distintas instrucciones de conversión.

Por ejemplo, la instrucción `Number` permite convertir un valor a algo numérico. Podemos emplearlo para convertir textos a enteros o reales:

```
var texto = "32";
var numero = Number(texto); // 32
```

Alternativamente, también disponemos de las instrucciones `parseInt` y `parseFloat` para convertir a entero o real, respectivamente.

```
var texto = "32";
var numero = parseInt(texto);   // 32
```

Para hacer el paso opuesto (convertir un número, o cualquier otro dato, a texto), podemos emplear la instrucción `String`:

```
var numero = 23;
var texto = String(numero); // "23"
```

Del mismo modo, también podemos emplear la instrucción `Boolean` para convertir un dato a booleano.

```
var texto = "true";
var resultado = Boolean(texto); // true
```

Una de las aplicaciones prácticas que tiene esto es poder convertir datos recogidos del usuario a otros tipos. Por ejemplo, ya hemos visto que con la instrucción `prompt` podemos pedirle al usuario que introduzca algún dato. Estos datos, JavaScript los interpreta directamente como texto. Así, si el usuario escribe "32", JavaScript no lo va a interpretar como que ha escrito el número 32, sino como que ha escrito los símbolos 3 y 2, simplemente.

Si queremos tratar lo que ha escrito el usuario como un número, para luego poder hacer operaciones con ese dato (sumarlo, por ejemplo), deberemos convertir el dato a numérico. El siguiente ejemplo calcula el doble del

número que indique el usuario, usando *parseInt*.

```
var numero = prompt("Dime un número");
var doble = parseInt(numero) * 2;
alert("El doble es " + doble);
```

NaN

En algunas conversiones u operaciones debemos tener especial cuidado con que los datos con que trabajamos sean numéricos, ya que de lo contrario el resultado que se va a producir va a ser *NaN* (abreviatura de *Not a Number*). Por ejemplo, si sumamos un entero y un valor *undefined*, el resultado será *NaN*.

```
var numero;
var otroNumero = parseInt(numero);
// otroNumero es NaN porque numero es *undefined*
```

Disponemos de la instrucción `isNaN` para comprobar si un dato no es numérico (devuelve *true* cuando no lo es):

```
if (isNaN(valor))
{
    alert("El valor no es numérico");
}
```

> **Ejercicio 2**
>
> Crea un documento llamado **operaciones_js.html** y, con JavaScript, pídele al usuario que introduzca dos números (primero uno y luego otro) y saca cuatro mensajes mostrando su suma, resta, multiplicación y división.

5. Control de flujo

Pasemos ahora a analizar qué estructuras proporciona JavaScript para poder ejecutar un bloque de instrucciones u otro en función de ciertas condiciones que se cumplen durante la ejecución del programa.

5.1. Estructuras selectivas

A veces nos interesa realizar una operación si se cumple una determinada condición y no hacerla (o hacer otra distinta) si no se cumple esa condición. Por ejemplo, si el usuario no ha rellenado su e-mail, querremos mostrarle un mensaje indicando que lo rellene, y si lo ha hecho, querremos poder enviar el formulario. Veamos qué estructuras ofrece JavaScript para poder elegir qué bloque de instrucciones ejecutar.

5.1.1. Estructura *if*

Para decidir entre varios caminos a seguir en función de una determinada condición se utiliza la estructura `if`:

```
if (condicion)
{
    instruccion1;
    instruccion2;
    ...
}
```

Esta estructura lleva entre paréntesis una condición (o una combinación de ellas, utilizando los operadores lógicos `&&` y `||`). Si esa condición se cumple, se ejecutarán las instrucciones que tenga entre llaves, y si no se cumple, no se ejecutarán. Por ejemplo, si nos guardáramos el e-mail que ha introducido el usuario en una variable email, podríamos comprobar si está vacío, y si es así, mostrar un mensaje de error:

```
if (email == "")
{
    alert ("Debes rellenar el e-mail antes de continuar");
}
```

En el caso de que queramos hacer otra cosa cuando no se cumple la condición, tenemos que utilizar la estructura `if..else`:

```
if (condicion)
{
    instruccion1a;
    instruccion2a;
    ...
}
else
{
    instruccion1b;
    instruccion2b;
    ...
}
```

En este caso, si la condición se cumple, como antes, ejecutaremos las instrucciones que hay entre las llaves del *if*, y si no se cumple, ejecutaremos las instrucciones que hay entre las llaves del *else*. En el ejemplo anterior, podríamos mostrar un mensaje de error o uno de OK, dependiendo de si el campo del e-mail está relleno o no:

```
if (email == "")
{
    alert ("Debes rellenar el e-mail antes de continuar");
}
else
{
    alert ("Todo correcto. Podemos continuar");
}
```

Si queremos elegir entre más de dos caminos, podemos utilizar la estructura `if..else if.. else if..` y poner una condición en cada *if* para cada camino. Por ejemplo, imaginemos que tenemos una variable edad donde almacenaremos la edad del usuario. Si el usuario no llega a 10 años le diremos que no tiene edad para ver la web. Si no llega a 18 años, le diremos que aún es menor de edad, pero puede ver la web, y si tiene más de 18 años le diremos que está todo correcto:

```
var edad = prompt("Dime tu edad");
edad = parseInt(edad);
if (edad < 10)
{
    alert ("No tienes edad para ver esta web");
}
else if (edad < 18)
{
    alert ("Aún eres menor de edad, pero puedes acceder a esta web");
}
else
{
    alert ("Todo correcto");
}
```

El último *else* podría ser `else if (edad >= 18)`, el efecto sería el mismo en este caso.

> **Ejercicio 3**:
>
> Crea un documento HTML llamado **notaBasica_js.html**. Utilizando JavaScript, crea una variable nota con una nota de examen que le pedirás al usuario. Después, utilizando `if..else`, saca un mensaje por pantalla indicando si el examen está aprobado o no.

> **Ejercicio 4**:
>
> Crea un documento llamado **notaNominal_js.html**, similar al anterior. En este caso, en lugar de decir simplemente si el examen está aprobado o no, deberá decir la nota nominal del examen (suspenso, aprobado, bien, notable o sobresaliente).

5.1.2. Estructura *switch*

La estructura `switch` permite analizar el valor que toma una expresión o variable de entre un conjunto de valores limitado o finito. Para cada posible valor, podemos elegir qué instrucción (o instrucciones) ejecutar. Para ello, cada posible valor que puede tomar la expresión se asocia a un grupo `case`.

El siguiente código muestra al usuario un mensaje u otro en función del valor de la variable *opcion*:

```
switch(opcion)
{
    case 1:
        console.log("Bienvenido");
        break;
    case 2:
        console.log("Hasta pronto");
        break;
    case 3:
        console.log("Acceso no permitido");
        break;
    default:
        console.log("Opción no reconocida");
        break;
}
```

La instrucción `break` al final de cada `case` hace que el programa "salga" del *switch* en cuanto termina de procesar ese caso. Si no la ponemos, seguirá ejecutando las instrucciones del caso siguiente. El caso `default` que se añade al final sirve para recoger cualquier otro posible valor de la expresión que no se haya contemplado en los casos anteriores.

Adicionalmente (y esto no es algo habitual en otros lenguajes) podemos incluir condiciones en los casos (*case*), haciendo que se comporten como *if..else*:

```
switch(true)
{
    case opcion < 2:
        console.log("La opción es menor que 2");
        break;
    case opcion < 10:
        console.log("La opción es mayor o igual que 2 y menor que 10");
        break;
    default:
        console.log("La opción es mayor o igual que 10");
}
```

5.2. Estructuras repetitivas o bucles

En ocasiones nos puede interesar que una instrucción o conjunto de instrucciones se repita automáticamente un número determinado de veces, o mientras se cumpla una determinada condición. Para ello, JavaScript pone a nuestra disposición distintos tipos de estructuras repetitivas.

La estructura `while` permite ejecutar un conjunto de instrucciones mientras se cumpla la condición que tiene entre paréntesis (que puede ser una condición simple o compuesta). Por ejemplo, el siguiente código va sacando mensajes con un número que va del 1 al 5:

```
var numero = 1;
while (numero <= 5)
{
    alert(numero);
    numero = numero + 1;    // También podríamos poner numero++
}
```

Alternativamente, la estructura `do..while` permite hacer lo mismo, pero evalúa la condición después de haber ejecutado el bloque de instrucciones. El ejemplo anterior quedaría así usando *do..while*:

```
var numero = 1;
do
{
    alert(numero);
    numero = numero + 1;    // También podríamos poner numero++
}
while (numero <= 5);
```

Finalmente, podemos hacer uso de la estructura `for`. En este caso, se suele emplear una variable que hace de contador, y dentro de la estructura le damos un valor inicial, indicamos la condición que debe cumplirse para seguir repitiendo el bucle, y cómo vamos a incrementar o decrementar el contador en cada repetición. El ejemplo anterior para contar de 1 a 5 se podría expresar así en un *for*:

```
for (var i = 1; i <= 5; i++)
{
    alert (i);
}
```

> **Ejercicio 5**:
>
> Crea un documento llamado **repeticiones_js.html**. En él, pídele al usuario que escriba dos números. Deberá ir contando del primero al segundo (sacando cada número en un alert). Deberás tener en cuenta cuál es el mayor de los dos, para saber si tienes que ir contando hacia arriba o hacia abajo (utiliza para esta comprobación la estructura *if* que hemos visto anteriormente).

5.3. "var" vs "let"

Comentábamos al principio de este documento que las variables en JavaScript se pueden declarar utilizando las palabras *var* o *let*, seguidas del nombre de la variable. Pero, ¿existe alguna diferencia en el uso de ambos términos? La respuesta es que sí.

Cuando declaramos una variable con `var`, su ámbito se extiende más allá de donde se ha declarado, un efecto que se conoce como *hoisting*, y que se debe a que JavaScript mueve las variables declaradas con *var* al inicio del bloque en que se han declarado. Por ejemplo, en este código declaramos la variable *nombre* dentro de un *if*, pero sigue existiendo después de ese *if*:

```
var numero = 3;
alert(nombre);  // undefined
if (numero >= 0)
{
    var nombre = "Nacho";
}
alert(nombre);  // Nacho
```

Notar que, si intentamos mostrar el valor de la variable *antes* de declararla con *var*, nos va a mostrar el texto *undefined*. Es decir, la variable existe (porque JavaScript la ha movido al principio), pero aún no le hemos dado valor. El mensaje de *alert* después del *if* sí nos mostrará el valor de la variable, aunque la hayamos declarado interna al *if*.

En cambio, cuando declaramos una variable con `let`, su ámbito es local a la zona donde se ha declarado. El mismo ejemplo anterior, utilizando `let`, nos dará un error de variable no declarada (en la consola del navegador):

```
let numero = 3;
if (numero >= 0)
{
    let nombre = "Nacho";
}
alert(nombre);  // Error
```

Lo recomendable es utilizar siempre variables *let*, para segurarnos de que su uso no se va a extender (y solapar sin intención) más allá del ámbito donde se han declarado.