SINTAXIS BÁSICA

Antes de comenzar a desarrollar programas y utilidades con JavaScript, es necesario conocer los elementos básicos con los que se construyen las aplicaciones. Este capítulo explica en detalle y comenzando desde cero los conocimientos básicos necesarios para poder comprender la sintaxis básica de Javascript. En el próximo capítulo veremos aspectos más avanzados como objetos, herencia, arrays o expresiones regulares.

3.1 ESPACIOS EN BLANCO

No se tienen en cuenta los espacios en blanco y las nuevas líneas: como sucede con XHTML, el intérprete de JavaScript ignora cualquier espacio en blanco sobrante, por lo que el código se puede ordenar de forma adecuada para entenderlo mejor (tabulando las líneas, añadiendo espacios, creando nuevas líneas, etc.) Sin embargo, en ocasiones estos espacios en blanco son totalmente necesarios, por ejemplo, para reparar nombres de variables o palabras reservadas. Por ejemplo:

```
var that = this;
```

Aquí el espacio en blanco entre `var` y `that` no puede ser eliminado, pero el resto sí.

3.2 COMENTARIOS

JavaScript ofrece dos tipos de comentarios, de bloque gracias a los caracteres `/* */` y de línea comenzando con `//`. El formato `/* */`de comentarios puede causar problemas en ciertas condiciones, como en las expresiones regulares, por lo que hay que tener cuidado al utilizarlo. Por ejemplo:

```
/*

    var rm_a = /a*/.match(s);

*/
```

provoca un error de sintaxis. Por lo tanto, suele ser recomendable utilizar únicamente los comentarios de línea, para evitar este tipo de problemas.

3.3 VARIABLES

Las variables en JavaScript se crean mediante la palabra reservada `var`. De esta forma, podemos declarar variables de la siguiente manera:

```
var numero_1 = 3;

var numero_2 = 1;

var resultado = numero_1 + numero_2;
```

La palabra reservada `var` solamente se debe indicar al declarar por primera vez la variable. Cuando se utilizan las variables en el resto de instrucciones del script, solamente es necesario indicar su nombre. En otras palabras, en el ejemplo anterior sería un error indicar lo siguiente:

```
var numero_1 = 3;

var numero_2 = 1;

var resultado = var numero_1 + var numero_2;
```

En JavaScript no es obligatorio inicializar las variables, ya que se pueden declarar por una parte y asignarles un valor posteriormente. Por tanto, el ejemplo anterior se puede rehacer de la siguiente manera:

```
var numero_1;

var numero_2;



numero_1 = 3;

numero_2 = 1;
```

```
var resultado = numero_1 + numero_2;
```

Una de las características más sorprendentes de JavaSript para los programadores habituados a otros lenguajes de programación es que tampoco es necesario declarar las variables. En otras palabras, se pueden utilizar variables que no se han definido anteriormente mediante la palabra reservada `var`. El ejemplo anterior también es correcto en JavaScript de la siguiente forma:

```
var numero_1 = 3;

var numero_2 = 1;

resultado = numero_1 + numero_2;
```

La variable `resultado` no está declarada, por lo que JavaScript crea una variable global (más adelante se verán las diferencias entre variables locales y globales) y le asigna el valor correspondiente. De la misma forma, también sería correcto el siguiente código:

```
numero_1 = 3;

numero_2 = 1;

resultado = numero_1 + numero_2;
```

En cualquier caso, se recomienda declarar todas las variables que se vayan a utilizar.

3.3.1 NOMBRES DE VARIABLES

El nombre de una variable también se conoce como **identificador** y debe cumplir las siguientes normas:

- Sólo puede estar formado por letras, números y los símbolos `$` (dólar) y `_` (guión bajo).
- El primer carácter no puede ser un número.

Por tanto, las siguientes variables tienen nombres correctos:

```
var $numero1;

var _$letra;

var $$$otroNumero;

var $_a__$4;
```

Sin embargo, las siguientes variables tienen identificadores incorrectos:

```
var 1numero;       // Empieza por un número

var numero;1_123;  // Contiene un carácter ";"
```

A continuación se indica el listado de palabras reservadas en JavaScript, y que no podremos utilizar para nombrar nuestras variables, parámetros, funciones, operadores o etiquetas:

- `abstract`
- `boolean break byte`
- `case catch char class const continue`
- `debugger default delete do double`
- `else enum export extends`

- `false final finally float for function`
- `goto`
- `if implements import in instanceof int interface`
- `long`
- `native new null`
- `package private protected public`
- `return`
- `short static super switch synchronized`
- `this throw throws transient true try typeof`
- `var volatile void`
- `while with`

3.3.2 TIPOS DE VARIABLES

JavaScript divide los distintos tipos de variables en dos grupos: tipos primitivos y tipos de referencia o clases.

3.3.2.1 TIPOS PRIMITIVOS

JavaScript define cinco tipos primitivos: `undefined`, `null`, `boolean`, `number` y `string`. Además de estos tipos, JavaScript define el operador `typeof` para averiguar el tipo de una variable.

3.3.2.1.1 VARIABLES DE TIPO UNDEFINED

El tipo `undefined` corresponde a las variables que han sido definidas y todavía no se les ha asignado un valor:

```
var variable1;

typeof variable1;  // devuelve "undefined"
```

3.3.2.1.2 VARIABLES DE TIPO NULL

Se trata de un tipo similar a `undefined`, y de hecho en JavaScript se consideran iguales (`undefined == null`). El tipo `null` se suele utilizar para representar objetos que en ese momento no existen.

```
var nombreUsuario = null;
```

3.3.2.1.3 VARIABLES DE TIPO BOOLEAN

Se trata de una variable que sólo puede almacenar uno de los dos valores especiales definidos y que representan el valor *"verdadero"* y el valor *"falso"*.

```
var variable1 = true;

var variable2 = false;
```

Los valores `true` y `false` son valores especiales, de forma que no son palabras ni números ni ningún otro tipo de valor. Este tipo de variables son esenciales para crear cualquier aplicación, tal y como se verá más adelante.

Cuando es necesario convertir una variable numérica a una variable de tipo `boolean`, JavaScript aplica la siguiente conversión: el número `0` se convierte en `false` y cualquier otro número distinto de `0` se convierte en `true`.

Por este motivo, en ocasiones se asocia el número `0` con el valor `false` y el número `1` con el valor `true`. Sin embargo, es necesario insistir en que `true` y `false` son valores especiales que no se corresponden ni con números ni con ningún otro tipo de dato.

3.3.2.2 CONVERSIÓN ENTRE TIPOS DE VARIABLES

JavaScript es un lenguaje de programación *"no tipado"*, lo que significa que una misma variable puede guardar diferentes tipos de datos a lo largo de la ejecución de la aplicación. De esta forma, una variable se podría inicializar con un valor numérico, después podría almacenar una cadena de texto y podría acabar la ejecución del programa en forma de variable booleana.

No obstante, en ocasiones es necesario que una variable almacene un dato de un determinado tipo. Para asegurar que así sea, se puede convertir una variable de un tipo a otro, lo que se denomina *typecasting*:

Así, JavaScript incluye un método llamado `toString()` que permite convertir variables de cualquier tipo a variables de cadena de texto, tal y como se muestra en el siguiente ejemplo:

```
var variable1 = true;

variable1.toString();  // devuelve "true" como cadena de texto

var variable2 = 5;

variable2.toString();  // devuelve "5" como cadena de texto
```

JavaScript también incluye métodos para convertir los valores de las variables en valores numéricos. Los métodos definidos son `parseInt()` y `parseFloat()`, que convierten la variable que se le indica en un número entero o un número decimal respectivamente.

La conversión numérica de una cadena se realiza carácter a carácter empezando por el de la primera posición. Si ese carácter no es un número, la función devuelve el valor NaN. Si el primer carácter es un número, se continúa con los siguientes caracteres mientras estos sean números.

```
var variable1 = "hola";

parseInt(variable1);  // devuelve NaN

var variable2 = "34";

parseInt(variable2);  // devuelve 34

var variable3 = "34hola23";

parseInt(variable3);  // devuelve 34

var variable4 = "34.23";

parseInt(variable4);  // devuelve 34
```

En el caso de `parseFloat()`, el comportamiento es el mismo salvo que también se considera válido el carácter . que indica la parte decimal del número:

```
var variable1 = "hola";

parseFloat(variable1);  // devuelve NaN

var variable2 = "34";

parseFloat(variable2);  // devuelve 34.0

var variable3 = "34hola23";

parseFloat(variable3);  // devuelve 34.0

var variable4 = "34.23";

parseFloat(variable4);  // devuelve 34.23
```

3.3.2.3 TIPOS DE REFERENCIA

Aunque JavaScript no define el concepto de clase, los tipos de referencia se asemejan a las clases de otros lenguajes de programación. Los objetos en JavaScript se crean mediante la palabra reservada `new` y el nombre de la clase que se va a instanciar. De esta forma, para crear un objeto de tipo `String` se indica lo siguiente (los paréntesis solamente son obligatorios cuando se utilizan argumentos, aunque se recomienda incluirlos incluso cuando no se utilicen):

```
var variable1 = new String("hola mundo");
```

JavaScript define una clase para cada uno de los tipos de datos primitivos. De esta forma, existen objetos de tipo `Boolean` para las variables booleanas, `Number` para las variables numéricas y `String` para las variables de

cadenas de texto. Las clases `Boolean`, `Number` y `String` almacenan los mismos valores de los tipos de datos primitivos y añaden propiedades y métodos para manipular sus valores.

```
var longitud = "hola mundo".length;
```

La propiedad `length` sólo está disponible en la clase `String`, por lo que en principio no debería poder utilizarse en un dato primitivo de tipo cadena de texto. Sin embargo, JavaScript convierte el tipo de dato primitivo al tipo de referencia `String`, obtiene el valor de la propiedad `length` y devuelve el resultado. Este proceso se realiza de forma automática y transparente para el programador.

En realidad, con una variable de tipo `String` no se pueden hacer muchas más cosas que con su correspondiente tipo de dato primitivo. Por este motivo, no existen muchas diferencias prácticas entre utilizar el tipo de referencia o el tipo primitivo, salvo en el caso del resultado del operador `typeof` y en el caso de la función `eval()`, como se verá más adelante.

La principal diferencia entre los tipos de datos es que los datos primitivos se manipulan por valor y los tipos de referencia se manipulan, como su propio nombre indica, por referencia. Los conceptos *"por valor"* y *"por referencia"* son iguales que en el resto de lenguajes de programación, aunque existen diferencias importantes (no existe por ejemplo el concepto de puntero).

Cuando un dato se manipula por valor, lo único que importa es el valor en sí. Cuando se asigna una variable por valor a otra variable, se copia directamente el valor de la primera variable en la segunda. Cualquier modificación que se realice en la segunda variable es independiente de la primera variable.

De la misma forma, cuando se pasa una variable por valor a una función (como se explicará más adelante) sólo se pasa una copia del valor. Así, cualquier modificación que realice la función sobre el valor pasado no se refleja en el valor de la variable original.

En el siguiente ejemplo, una variable se asigna por valor a otra variable:

```
var variable1 = 3;

var variable2 = variable1;



variable2 = variable2 + 5;

// Ahora variable2 = 8 y variable1 sigue valiendo 3
```

La `variable1` se asigna por valor en la `variable1`. Aunque las dos variables almacenan en ese momento el mismo valor, son independientes y cualquier cambio en una de ellas no afecta a la otra. El motivo es que los tipos de datos primitivos siempre se asignan (y se pasan) por valor.

Sin embargo, en el siguiente ejemplo, se utilizan tipos de datos de referencia:

```
// variable1 = 25 diciembre de 2009

var variable1 = new Date(2009, 11, 25);

// variable2 = 25 diciembre de 2009

var variable2 = variable1;



// variable2 = 31 diciembre de 2010

variable2.setFullYear(2010, 11, 31);

// Ahora variable1 también es 31 diciembre de 2010
```

En el ejemplo anterior, se utiliza un tipo de dato de referencia que se verá más adelante, que se llama `Date` y que se utiliza para manejar fechas. Se crea una variable llamada `variable1` y se inicializa la fecha a 25 de diciembre de 2009. Al constructor del objeto `Date` se le pasa el año, el número del mes (siendo `0` = enero, `1` = febrero, ..., `11` = diciembre) y el día (al contrario que el mes, los días no empiezan en `0` sino en `1`). A continuación, se asigna el valor de la `variable1` a otra variable llamada `variable2`.

Como `Date` es un tipo de referencia, la asignación se realiza por referencia. Por lo tanto, las dos variables quedan "unidas" y hacen referencia al mismo objeto, al mismo dato de tipo `Date`. De esta forma, si se modifica el valor de `variable2` (y se cambia su fecha a 31 de diciembre de 2010) el valor de `variable1` se verá automáticamente modificado.

3.4 NÚMEROS

En JavaScript únicamente existe un tipo de número. Internamente, es representado como un dato de `64 bits` en coma flotante, al igual el tipo de dato `double` en Java. A diferencia de otros lenguajes de programación, no existe una diferencia entre un número entero y otro decimal, por lo que `1` y `1.0`son el mismo valor. Esto es significativo ya que evitamos los problemas desbordamiento en tipos de dato *pequeños*, al no existir la necesidad de conocer el tipo de dato.

3.4.1 TIPOS DE NÚMEROS

Si el número es entero, se indica su valor directamente.

```
var variable1 = 10;
```

Si el número es decimal, se debe utilizar el punto (.) para separar la parte entera de la decimal.

```
var variable2 = 3.14159265;
```

Además del sistema numérico decimal, también se pueden indicar valores en el sistema octal (si se incluye un cero delante del número) y en sistema hexadecimal (si se incluye un cero y una x delante del número).

```
var variable1 = 10;

var variable_octal = 034;

var variable_hexadecimal = 0xA3;
```

JavaScript define tres valores especiales muy útiles cuando se trabaja con números. En primer lugar se definen los valores `Infinity` y `-Infinity` para representar números demasiado grandes (positivos y negativos) y con los que JavaScript no puede trabajar.

```
var variable1 = 3, variable2 = 0;

console.log(variable1/variable2);  // muestra "Infinity"
```

El otro valor especial definido por JavaScript es `NaN`, que es el acrónimo de *"Not a Number"*. De esta forma, si se realizan operaciones matemáticas con variables no numéricas, el resultado será de tipo `NaN`.

Para manejar los valores `NaN`, se utiliza la función relacionada `isNaN()`, que devuelve `true` si el parámetro que se le pasa no es un número:

```
var variable1 = 3;

var variable2 = "hola";

isNaN(variable1);  // false

isNaN(variable2);  // true

isNaN(variable1 + variable2);  // true
```

Por último, JavaScript define algunas constantes matemáticas que representan valores numéricos significativos:

Constante	Valor	Significado
`Math.E`	`2.718281828459045`	Constante de Euler, base de los logaritmos naturales y también llamado *número e*
`Math.LN2`	`0.6931471805599453`	Logaritmo natural de `2`
`Math.LN10`	`2.302585092994046`	Logaritmo natural de `10`
`Math.LOG2E`	`1.4426950408889634`	Logaritmo en base `2` de `Math.E`
`Math.LOG10E`	`0.4342944819032518`	Logaritmo en base `10` de `Math.E`
`Math.PI`	`3.141592653589793`	Pi, relación entre el radio de una circunferencia y su diámetro
`Math.SQRT1_2`	`0.7071067811865476`	Raíz cuadrada de `1/2`
`Math.SQRT2`	`1.4142135623730951`	Raíz cuadrada de `2`

De esta forma, para calcular el área de un círculo de radio r, se debe utilizar la constante que representa al número Pi:

```
var area = Math.PI * r * r;
```

3.5 CADENAS DE TEXTO

Las variables de tipo cadena de texto permiten almacenar cuaquier sucesión de caracteres, por lo que se utilizan ampliamente en la mayoría de aplicaciones JavaScript. Cada carácter de la cadena se encuentra en una posición a la que se puede acceder individualmente, siendo el primer carácter el de la posición `0`.

El valor de las cadenas de texto se indica encerrado entre comillas simples o dobles:

```
var variable1 = "hola";

var variable2 = 'mundo';

var variable3 = "hola mundo, esta es una frase más larga";
```

Las cadenas de texto pueden almacenar cualquier carácter, aunque algunos no se pueden incluir directamente en la declaración de la variable. Si por ejemplo se incluye un `ENTER` para mostrar el resto de caracteres en la línea siguiente, se produce un error en la aplicación:

```
var variable = "hola mundo, esta es

                                una frase más larga";
```

La variable anterior no está correctamente definida y se producirá un error en la aplicación. Por tanto, resulta evidente que algunos caracteres *especiales* no se pueden incluir directamente. De la misma forma, como las comillas (doble y simple) se utilizan para encerrar los contenidos, también se pueden producir errores:

```
var variable1 = "hola 'mundo'";

var variable2 = 'hola "mundo"';

var variable3 = "hola 'mundo', esta es una "frase" más larga";
```

Si el contenido de texto tiene en su interior alguna comilla simple, se encierran los contenidos con comillas dobles (como en el caso de la `variable1` anterior). Si el contenido de texto tiene en su interior alguna comilla doble, se encierran sus contenidos con comillas simples (como en el caso de la `variable2` anterior). Sin embargo, en el caso de la `variable3` su contenido tiene tanto comillas simples como comillas dobles, por lo que su declaración provocará un error.

Para resolver estos problemas, JavaScript define un mecanismo para incluir de forma sencilla caracteres especiales (`ENTER`, `Tabulador`) y problemáticos (comillas). Esta estrategia se denomina "mecanismo de escape", ya que se sustituyen los caracteres problemáticos por otros caracteres seguros que siempre empiezan con la barra `\`:

Si se quiere incluir...	**Se debe sustituir por...**
Una nueva línea	`\n`
Un tabulador	`\t`
Una comilla simple	`\'`
Una comilla doble	`\"`
Una barra inclinada	`\\`

Utilizando el mecanismo de escape, se pueden corregir los ejemplos anteriores:

```
var variable = "hola mundo, esta es \n una frase más larga";

var variable3 = "hola 'mundo', esta es una \"frase\" más larga";
```

OPERADORES

Los operadores permiten manipular el valor de las variables, realizar operaciones matemáticas con sus valores y comparar diferentes variables. De esta forma, los operadores permiten a los programas realizar cálculos complejos y tomar decisiones lógicas en función de comparaciones y otros tipos de condiciones.

4.1 ASIGNACIÓN

El operador de asignación es el más utilizado y el más sencillo. Este operador se utiliza para guardar un valor específico en una variable. El símbolo utilizado es `=` (no confundir con el operador `==` que se verá más adelante):

```
var numero1 = 3;
```

A la izquierda del operador, siempre debe indicarse el nombre de una variable. A la derecha del operador, se pueden indicar variables, valores, condiciones lógicas, etc:

```
var numero1 = 3;

var numero2 = 4;



/* Error, la asignación siempre se realiza a una variable,

    por lo que en la izquierda no se puede indicar un número */

5 = numero1;



// Ahora, la variable numero1 vale 5

numero1 = 5;



// Ahora, la variable numero1 vale 4

numero1 = numero2;
```

4.2 ARITMÉTICOS

JavaScript permite realizar manipulaciones matemáticas sobre el valor de las variables numéricas. Los operadores definidos son: suma (`+`), resta (`-`), multiplicación (`*`), división (`/`) y módulo (`%`). Ejemplo:

```
var numero1 = 10;

var numero2 = 5;



resultado = numero1 / numero2;  // resultado = 2

resultado = 3 + numero1;        // resultado = 13

resultado = numero2 - 4;        // resultado = 1

resultado = numero1 * numero 2; // resultado = 50

resultado = numero1 % numero2;  // resultado = 0



numero1 = 9;

numero2 = 5;

resultado = numero1 % numero2;  // resultado = 4
```

Los operadores matemáticos también se pueden combinar con el operador de asignación para abreviar su notación:

```
var numero1 = 5;

numero1 += 3;   // numero1 = numero1 + 3 = 8

numero1 -= 1;   // numero1 = numero1 - 1 = 4

numero1 *= 2;   // numero1 = numero1 * 2 = 10

numero1 /= 5;   // numero1 = numero1 / 5 = 1

numero1 %= 4;   // numero1 = numero1 % 4 = 1
```

Existen dos operadores especiales que solamente son válidos para las variables numéricas y se utilizan para incrementar o decrementar en una unidad el valor de una variable.

Ejemplo:

```
var numero = 5;

++numero;

console.log(numero);  // numero = 6
```

El operador de incremento se indica mediante el prefijo `++` en el nombre de la variable. El resultado es que el valor de esa variable se incrementa en una unidad. Por tanto, el anterior ejemplo es equivalente a:

```
var numero = 5;

numero = numero + 1;

console.log(numero);  // numero = 6
```

De forma equivalente, el operador decremento (indicado como un prefijo -- en el nombre de la variable) se utiliza para decrementar el valor de la variable:

```
var numero = 5;

--numero;

console.log(numero);  // numero = 4
```

El anterior ejemplo es equivalente a:

```
var numero = 5;

numero = numero - 1;

console.log(numero);  // numero = 4
```

Los operadores de incremento y decremento no solamente se pueden indicar como prefijo del nombre de la variable, sino que también es posible utilizarlos como sufijo. En este caso, su comportamiento es similar pero muy diferente. En el siguiente ejemplo:

```
var numero = 5;

numero++;

console.log(numero);  // numero = 6
```

El resultado de ejecutar el script anterior es el mismo que cuando se utiliza el operador `++numero`, por lo que puede parecer que es equivalente indicar el operador `++` delante o detrás del identificador de la variable. Sin embargo, el siguiente ejemplo muestra sus diferencias:

```
var numero1 = 5;
```

```
var numero2 = 2;

numero3 = numero1++ + numero2;

// numero3 = 7, numero1 = 6



var numero1 = 5;

var numero2 = 2;

numero3 = ++numero1 + numero2;

// numero3 = 8, numero1 = 6
```

Si el operador `++` se indica como prefijo del identificador de la variable, su valor se incrementa **antes** de realizar cualquier otra operación. Si el operador `++` se indica como sufijo del identificador de la variable, su valor se incrementa **después** de ejecutar la sentencia en la que aparece.

Por tanto, en la instrucción `numero3 = numero1++ + numero2;`, el valor de `numero1` se incrementa después de realizar la operación (primero se suma y `numero3` vale `7`, después se incrementa el valor de `numero1` y vale `6`). Sin embargo, en la instrucción `numero3 = ++numero1 + numero2;`, en primer lugar se incrementa el valor de `numero1` y después se realiza la suma (primero se incrementa `numero1` y vale `6`, después se realiza la suma y `numero3` vale `8).

4.3 LÓGICOS

Los operadores lógicos son imprescindibles para realizar aplicaciones complejas, ya que se utilizan para tomar decisiones sobre las instrucciones que debería ejecutar el programa en función de ciertas condiciones. El resultado de cualquier operación que utilice operadores lógicos siempre es un valor lógico o `booleano`.

4.3.1 NEGACIÓN

Uno de los operadores lógicos más utilizados es el de la negación. Se utiliza para obtener el valor contrario al valor de la variable:

```
var visible = true;

console.log(!visible);  // Muestra "false" y no "true"
```

La negación lógica se obtiene prefijando el símbolo `!` al identificador de la variable. El funcionamiento de este operador se resume en la siguiente tabla:

Variable	**!variable**
`True`	`false`
`False`	`true`

Si la variable original es de tipo *booleano*, es muy sencillo obtener su negación. Sin embargo, ¿qué sucede cuando la variable es un número o una cadena de texto? Para obtener la negación en este tipo de variables, se realiza en primer lugar su conversión a un valor booleano:

- Si la variable contiene un número, se transforma en `false` si vale `0` y en `true` para cualquier otro número (positivo o negativo, decimal o entero).
- Si la variable contiene una cadena de texto, se transforma en `false` si la cadena es vacía (`""`) y en `true` en cualquier otro caso.

```
var cantidad = 0;

vacio = !cantidad;  // vacio = true
```

```
cantidad = 2;

vacio = !cantidad;  // vacio = false


var mensaje = "";

mensajeVacio = !mensaje;  // mensajeVacio = true


mensaje = "Bienvenido";

mensajeVacio = !mensaje;  // mensajeVacio = false
```

4.3.2 AND

La operación lógica `AND` obtiene su resultado combinando dos valores booleanos. El operador se indica mediante el símbolo `&&` y su resultado solamente es `true` si los dos operandos son `true`:

variable1	variable2	variable1 && variable2
`true`	`True`	`true`
`true`	`false`	`false`
`false`	`True`	`false`
`false`	`false`	`false`

```
var valor1 = true;

var valor2 = false;

resultado = valor1 && valor2; // resultado = false


valor1 = true;

valor2 = true;

resultado = valor1 && valor2; // resultado = true
```

4.3.3 OR

La operación lógica `OR` también combina dos valores booleanos. El operador se indica mediante el símbolo `||` y su resultado es true si alguno de los dos operandos es `true`:

variable1	variable2	variable1 || variable2
`true`	`true`	`true`

variable1	variable2	variable1 || variable2
`true`	`false`	`false`
`false`	`true`	`true`
`false`	`false`	`false`

```
var valor1 = true;

var valor2 = false;

resultado = valor1 || valor2; // resultado = true



valor1 = false;

valor2 = false;

resultado = valor1 || valor2; // resultado = false
```

4.4 RELACIONALES

Los operadores relacionales definidos por JavaScript son idénticos a los que definen las matemáticas: mayor que (`>`), menor que (`<`), mayor o igual (`>=`), menor o igual (`<=`), igual que (`==`) y distinto de (`!=`).

Los operadores que relacionan variables son imprescindibles para realizar cualquier aplicación compleja. El resultado de todos estos operadores siempre es un valor booleano:

```
var numero1 = 3;

var numero2 = 5;

resultado = numero1 > numero2;  // resultado = false

resultado = numero1 < numero2;  // resultado = true



numero1 = 5;

numero2 = 5;

resultado = numero1 >= numero2; // resultado = true

resultado = numero1 <= numero2; // resultado = true

resultado = numero1 == numero2; // resultado = true

resultado = numero1 != numero2; // resultado = false
```

Se debe tener especial cuidado con el operador de igualdad (`==`), ya que es el origen de la mayoría de errores de programación, incluso para los usuarios que ya tienen cierta experiencia desarrollando scripts. El operador `==` se utiliza para comparar el valor de dos variables, por lo que es muy diferente del operador `=`, que se utiliza para asignar un valor a una variable:

```
// El operador "=" asigna valores

var numero1 = 5;

resultado = numero1 = 3;  // numero1 = 3 y resultado = 3
```

```
// El operador "==" compara variables

var numero1 = 5;

resultado = numero1 == 3; // numero1 = 5 y resultado = false
```

Los operadores relacionales también se pueden utilizar con variables de tipo cadena de texto:

```
var texto1 = "hola";

var texto2 = "hola";

var texto3 = "adios";



resultado = texto1 == texto3; // resultado = false

resultado = texto1 != texto2; // resultado = false

resultado = texto3 >= texto2; // resultado = false
```

Cuando se utilizan cadenas de texto, los operadores "mayor que" (`>`) y "menor que" (`<`) siguen un razonamiento no intuitivo: se compara letra a letra comenzando desde la izquierda hasta que se encuentre una diferencia entre las dos cadenas de texto. Para determinar si una letra es mayor o menor que otra, las mayúsculas se consideran menores que las minúsculas y las primeras letras del alfabeto son menores que las últimas (`a` es menor que `b`, `b` es menor que `c`, `A` es menor que `a`, etc.)

4.5 `TYPEOF`

El operador `typeof` se emplea para determinar el tipo de dato que almacena una variable. Su uso es muy sencillo, ya que sólo es necesario indicar el nombre de la variable cuyo tipo se quiere averiguar:

```
var myFunction = function() {

  console.log('hola');

};



var myObject = {

    foo : 'bar'

};



var myArray = [ 'a', 'b', 'c' ];



var myString = 'hola';



var myNumber = 3;



typeof myFunction;    // devuelve 'function'

typeof myObject;      // devuelve 'object'

typeof myArray;       // devuelve 'object' -- tenga cuidado
```

```
typeof myString;      // devuelve 'string'

typeof myNumber;      // devuelve 'number'


typeof null;          // devuelve 'object' -- tenga cuidado


if (myArray.push && myArray.slice && myArray.join) {

    // probablemente sea un vector

    // (este estilo es llamado, en inglés, "duck typing")

}


if (Object.prototype.toString.call(myArray) === '[object Array]') {

    // definitivamente es un vector;

    // esta es considerada la forma más robusta

    // de determinar si un valor es un vector.

}
```

Los posibles valores de retorno del operador son: `undefined`, `boolean`, `number`, `string` para cada uno de los tipos primitivos y `object` para los valores de referencia y también para los valores de tipo `null`.

El operador `typeof` no distingue entre las variables declaradas pero no inicializadas y las variables que ni siquiera han sido declaradas:

```
var variable1;



// devuelve "undefined", aunque la variable1 ha sido declarada

typeof variable1;

// devuelve "undefined", la variable2 no ha sido declarada

typeof variable2;
```

4.6 `INSTANCEOF`

El operador `typeof` no es suficiente para trabajar con tipos de referencia, ya que devuelve el valor `object` para cualquier objeto independientemente de su tipo. Por este motivo, JavaScript define el operador `instanceof` para determinar la clase concreta de un objeto.

```
var variable1 = new String("hola mundo");

typeof variable1;             // devuelve "object"

variable1 instanceof String;  // devuelve true
```

El operador `instanceof` sólo devuelve como valor `true` o `false`. De esta forma, `instanceof` no devuelve directamente la clase de la que ha instanciado la variable, sino que se debe comprobar cada posible tipo de clase individualmente.

ESTRUCTURAS DE CONTROL

5.1 ESTRUCTURA IF...ELSE

La estructura más utilizada en JavaScript y en la mayoría de lenguajes de programación es la estructura `if`. Se emplea para tomar decisiones en función de una condición. Su definición formal es:

```
if(condicion) {

  ...

}
```

Si la condición se cumple (es decir, si su valor es `true`) se ejecutan todas las instrucciones que se encuentran dentro del bloque `{...}`. Si la condición no se cumple (es decir, si su valor es `false`) no se ejecuta ninguna instrucción contenida en `{...}` y el programa continúa ejecutando el resto de instrucciones del script.

Ejemplo:

```
var mostrarMensaje = true;



if(mostrarMensaje) {

  console.log("Hola Mundo");

}
```

En el ejemplo anterior, el mensaje sí que se muestra al usuario ya que la variable `mostrarMensaje` tiene un valor de `true` y por tanto, el programa entra dentro del bloque de instrucciones del `if`.

El ejemplo se podría reescribir también como:

```
var mostrarMensaje = true;



if(mostrarMensaje == true) {

  console.log("Hola Mundo");

}
```

En este caso, la condición es una comparación entre el valor de la variable `mostrarMensaje` y el valor `true`. Como los dos valores coinciden, la igualdad se cumple y por tanto la condición es cierta, su valor es `true` y se ejecutan las instrucciones contenidas en ese bloque del `if`.

La comparación del ejemplo anterior suele ser el origen de muchos errores de programación, al confundir los operadores `==` y `=`. Las comparaciones siempre se realizan con el operador `==`, ya que el operador `=` solamente asigna valores:

```
var mostrarMensaje = true;



// Se comparan los dos valores

if(mostrarMensaje == false) {

  ...

}
```

```
// Error - Se asigna el valor "false" a la variable

if(mostrarMensaje = false) {

  ...

}
```

La condición que controla el `if()` puede combinar los diferentes operadores lógicos y relacionales mostrados anteriormente:

```
var mostrado = false;



if(!mostrado) {

  console.log("Es la primera vez que se muestra el mensaje");

}
```

Los operadores `AND` y `OR` permiten encadenar varias condiciones simples para construir condiciones complejas:

```
var mostrado = false;

var usuarioPermiteMensajes = true;



if(!mostrado && usuarioPermiteMensajes) {

  console.log("Es la primera vez que se muestra el mensaje");

}
```

La condición anterior está formada por una operación `AND` sobre dos variables. A su vez, a la primera variable se le aplica el operador de negación antes de realizar la operación `AND`. De esta forma, como el valor de `mostrado` es `false`, el valor `!mostrado` sería `true`. Como la variable `usuarioPermiteMensajes` vale `true`, el resultado de `!mostrado && usuarioPermiteMensajes` sería igual a `true && true`, por lo que el resultado final de la condición del `if()` sería `true` y por tanto, se ejecutan las instrucciones que se encuentran dentro del bloque del `if()`.

En ocasiones, las decisiones que se deben realizar no son del tipo *"si se cumple la condición, hazlo; si no se cumple, no hagas nada"*. Normalmente las condiciones suelen ser del tipo *"si se cumple esta condición, hazlo; si no se cumple, haz esto otro"*.

Para este segundo tipo de decisiones, existe una variante de la estructura `if` llamada `if...else`. Su definición formal es la siguiente:

```
if(condicion) {

  ...

}

else {

  ...

}
```

Si la condición se cumple (es decir, si su valor es `true`) se ejecutan todas las instrucciones que se encuentran dentro del `if()`. Si la condición no se cumple (es decir, si su valor es `false`) se ejecutan todas las instrucciones contenidas en `else { }`. Ejemplo:

```
var edad = 18;
```

```
if(edad >= 18) {

  console.log("Eres mayor de edad");

} else {

  console.log("Todavía eres menor de edad");

}
```

Si el valor de la variable `edad` es mayor o igual que el valor numérico `18`, la condición del `if()` se cumple y por tanto, se ejecutan sus instrucciones y se muestra el mensaje `"Eres mayor de edad"`. Sin embargo, cuando el valor de la variable `edad` no es igual o mayor que `18`,la condición del `if()` no se cumple, por lo que automáticamente se ejecutan todas las instrucciones del bloque `else { }`. En este caso, se mostraría el mensaje `"Todavía eres menor de edad"`.

El siguiente ejemplo compara variables de tipo cadena de texto:

```
var nombre = "";

if(nombre == "") {

  console.log("Aún no nos has dicho tu nombre");

} else {

  console.log("Hemos guardado tu nombre");

}
```

La condición del `if()` anterior se construye mediante el operador `==`, que es el que se emplea para comparar dos valores (no confundir con el operador `=` que se utiliza para asignar valores). En el ejemplo anterior, si la cadena de texto almacenada en la variable `nombre` es vacía (es decir, es igual a `""`) se muestra el mensaje definido en el `if()`. En otro caso, se muestra el mensaje definido en el bloque `else { }`.

La estructura `if...else` se puede encadenar para realizar varias comprobaciones seguidas:

```
if(edad < 12) {

  console.log("Todavía eres muy pequeño");

} else if(edad < 19) {

  console.log("Eres un adolescente");

} else if(edad < 35) {

  console.log("Aun sigues siendo joven");

} else {

  console.log("Piensa en cuidarte un poco más");

}
```

No es obligatorio que la combinación de estructuras `if...else` acabe con la instrucción `else`, ya que puede terminar con una instrucción de tipo `else if()`.

5.2 ESTRUCTURA SWITCH

La estructura `switch` es muy útil cuando la condición que evaluamos puede tomar muchos valores. Si utilizasemos una sentencia `if...else`, tendríamos que repetir la condición para los distintos valores.

```
if(dia == 1) {

  console.log("Hoy es lunes.");
```

```
} else if(dia == 2) {

  console.log("Hoy es martes.");

} else if(dia == 3) {

  console.log("Hoy es miércoles.");

} else if(dia == 4) {

  console.log("Hoy es jueves.");

} else if(dia == 5) {

  console.log("Hoy es viernes.");

} else if(dia == 6) {

  console.log("Hoy es sábado.");

} else if(dia == 0) {

  console.log("Hoy es domingo.");

}
```

En este caso es más conveniente utilizar una estructura de control de tipo `switch`, ya que permite ahorrarnos trabajo y producir un código más limpio. Su definición formal es la siguiente:

```
switch(dia) {

  case 1: console.log("Hoy es lunes."); break;

  case 2: console.log("Hoy es martes."); break;

  case 3: console.log("Hoy es miércoles."); break;

  case 4: console.log("Hoy es jueves."); break;

  case 5: console.log("Hoy es viernes."); break;

  case 6: console.log("Hoy es sábado."); break;

  case 0: console.log("Hoy es domingo."); break;

}
```

La cláusula `case` no tiene por qué ser una constante, sino que puede ser una expresión al igual que en la estructura `if`. El comportamiento por defecto de la estructura `switch`es seguir evaluando el resto de cláusulas, aún cuando una de ellas haya cumplido la condición. Para evitar ese comportamiento, es necesario utilizar la sentencia `break` en las cláusulas que deseemos.

5.3 ESTRUCTURA WHILE

La estructura `while` ejecuta un simple bucle, mientras se cumpla la condición. Su definición formal es la siguiente:

```
var veces = 0;
```

```
while(veces < 7) {

  console.log("Mensaje " + veces);

  veces++;

}
```

La idea del funcionamiento de un bucle `while` es la siguiente: "mientras la condición indicada se siga cumpliendo, repite la ejecución de las instrucciones definidas dentro del `while`. Es importante modificar los valores de las variables incluidas dentro de la condición, ya que otra manera, el bucle se repetiría de manera indefinida, perjudicando la ejecución de la página y bloqueando la ejecución del resto del script.

```
var veces = 0;



while(veces < 7) {

  console.log("Mensaje " + veces);

  veces = 0;

}
```

En este ejemplo, se mostraría de manera infinita una alerta con el texto "Mensaje 0".

5.4 ESTRUCTURA FOR

La estructura `for` permite realizar bucles de una forma muy sencilla. Su definición formal es la siguiente:

```
for(inicializacion; condicion; actualizacion) {

  ...

}
```

La idea del funcionamiento de un bucle `for` es la siguiente: *"mientras la condición indicada se siga cumpliendo, repite la ejecución de las instrucciones definidas dentro del for. Además, después de cada repetición, actualiza el valor de las variables que se utilizan en la condición"*.

- La "inicialización" es la zona en la que se establece los valores iniciales de las variables que controlan la repetición.
- La "condición" es el único elemento que decide si continua o se detiene la repetición.
- La "actualización" es el nuevo valor que se asigna después de cada repetición a las variables que controlan la repetición. var mensaje = "Hola, estoy dentro de un bucle";

```
for(var i = 0; i < 5; i++) {

  console.log(mensaje);

}
```

La parte de la inicialización del bucle consiste en:

```
var i = 0;
```

Por tanto, en primer lugar se crea la variable `i` y se le asigna el valor de `0`. Esta zona de inicialización solamente se tiene en consideración justo antes de comenzar a ejecutar el bucle. Las siguientes repeticiones no tienen en cuenta esta parte de inicialización.

La zona de condición del bucle es:

```
i < 5
```

Los bucles se siguen ejecutando mientras se cumplan las condiciones y se dejan de ejecutar justo después de comprobar que la condición no se cumple. En este caso, mientras la variable `i` valga menos de `5` el bucle se ejecuta indefinidamente.

Como la variable `i` se ha inicializado a un valor de `0` y la condición para salir del bucle es que `i` sea menor que `5`, si no se modifica el valor de `i` de alguna forma, el bucle se repetiría indefinidamente.

Por ese motivo, es imprescindible indicar la zona de actualización, en la que se modifica el valor de las variables que controlan el bucle:

```
i++
```

En este caso, el valor de la variable `i` se incrementa en una unidad después de cada repetición. La zona de actualización se ejecuta después de la ejecución de las instrucciones que incluye el `for`.

Así, durante la ejecución de la quinta repetición el valor de `i` será `4`. Después de la quinta ejecución, se actualiza el valor de `i`, que ahora valdrá `5`. Como la condición es que `i` sea menor que `5`, la condición ya no se cumple y las instrucciones del `for` no se ejecutan una sexta vez.

Normalmente, la variable que controla los bucles `for` se llama `i`, ya que recuerda a la palabra índice y su nombre tan corto ahorra mucho tiempo y espacio.

El ejemplo anterior que mostraba los días de la semana contenidos en un array se puede rehacer de forma más sencilla utilizando la estructura `for`:

```
var dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"];



for(var i=0; i<7; i++) {

  console.log(dias[i]);

}
```

5.5 ESTRUCTURA FOR...IN

Una estructura de control derivada de `for` es la estructura `for...in`. Su definición exacta implica el uso de objetos, permitiendo recorrer las propiedades de un objeto. En cada iteración, un nuevo nombre de propiedad del objeto es asignada a la variable:

```
for(propiedad in object) {

  if (object.hasOwnProperty(propiedad)) {

    ...

  }

}
```

Suele ser conveniente comprobar que la propiedad pertenece efectivamente al objeto, a través de `object.hasOwnProperty(propiedad)`. De la misma manera que podemos recorrer las propiedades de un objeto, es posible adaptar este comportamiento a los arrays:

```
for(indice in array) {

  ...

}
```

Si se quieren recorrer todos los elementos que forman un array, la estructura `for...in` es la forma más eficiente de hacerlo, como se muestra en el siguiente ejemplo:

```
var dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"];
```

```
for(i in dias) {

  console.log(dias[i]);

}
```

La variable que se indica como indice es la que se puede utilizar dentro del bucle `for...in` para acceder a los elementos del array. De esta forma, en la primera repetición del bucle la variable `i` vale `0` y en la última vale `6`.

Esta estructura de control es la más adecuada para recorrer arrays (y objetos), ya que evita tener que indicar la inicialización y las condiciones del bucle `for` simple y funciona correctamente cualquiera que sea la longitud del array. De hecho, sigue funcionando igual aunque varíe el número de elementos del array.

5.6 ESTRUCTURA TRY

La estructura `try` consiste en un bloque de código que se ejecuta de manera normal, y captura cualquier excepción que se pueda producir en ese bloque de sentencias. Su definición formal es la siguiente:

```
try {

    funcion_que_no_existe();

} catch(ex) {

    console.log("Error detectado: " + ex.description);

}
```

En este ejemplo, llamamos a una función que no está definida, y por lo tanto provoca una excepción en JavaScript. Este error es *capturado* por la cláusula `catch`, que contiene una serie de sentencias que indican que acciones realizar con esa excepción que acaba de producirse. Si no se produce ninguna excepción en el bloque `try`, no se ejecuta el bloque dentro de `catch`.

5.6.1 LA CLÁUSULA FINALLY

La cláusula `finally` contiene las sentencias a ejecutar después de los bloques `try` y `catch`. Las sentencias incluidas en este bloque se ejecutan siempre, se haya producido una excepción o no. Un ejemplo clásico de utilización de la cláusula `finally`, es la de liberar recursos que el script ha solicitado.

```
abrirFichero()

try {

   escribirFichero(datos);

} catch(ex) {

    // Tratar la excepción

} finally {

   cerrarFichero(); // siempre se cierra el recurso

}
```

FUNCIONES Y PROPIEDADES BÁSICAS

JavaScript incorpora una serie de herramientas y utilidades (llamadas funciones y propiedades, como se verá más adelante) para el manejo de las variables. De esta forma, muchas de las operaciones básicas con las variables, se pueden realizar directamente con las utilidades que ofrece JavaScript.

6.1 FUNCIONES ÚTILES PARA CADENAS DE TEXTO

A continuación se muestran algunas de las funciones más útiles para el manejo de cadenas de texto:

`length`, calcula la longitud de una cadena de texto (el número de caracteres que la forman)

```
var mensaje = "Hola Mundo";

var numeroLetras = mensaje.length; // numeroLetras = 10
```

`+`, se emplea para concatenar varias cadenas de texto

```
var mensaje1 = "Hola";

var mensaje2 = " Mundo";

var mensaje = mensaje1 + mensaje2; // mensaje = "Hola Mundo"
```

Además del operador `+`, también se puede utilizar la función `concat()`

```
var mensaje1 = "Hola";

var mensaje2 = mensaje1.concat(" Mundo"); // mensaje2 = "Hola Mundo"
```

Las cadenas de texto también se pueden unir con variables numéricas:

```
var variable1 = "Hola ";

var variable2 = 3;

var mensaje = variable1 + variable2; // mensaje = "Hola 3"
```

Cuando se unen varias cadenas de texto es habitual olvidar añadir un espacio de separación entre las palabras:

```
var mensaje1 = "Hola";

var mensaje2 = "Mundo";

var mensaje = mensaje1 + mensaje2; // mensaje = "HolaMundo"
```

Los espacios en blanco se pueden añadir al final o al principio de las cadenas y también se pueden indicar forma explícita:

```
var mensaje1 = "Hola";

var mensaje2 = "Mundo";

var mensaje = mensaje1 + " " + mensaje2; // mensaje = "Hola Mundo"
```

`toUpperCase()`, transforma todos los caracteres de la cadena a sus correspondientes caracteres en mayúsculas:

```
var mensaje1 = "Hola";

var mensaje2 = mensaje1.toUpperCase(); // mensaje2 = "HOLA"
```

`toLowerCase()`, transforma todos los caracteres de la cadena a sus correspondientes caracteres en minúsculas:

```
var mensaje1 = "HolA";

var mensaje2 = mensaje1.toLowerCase(); // mensaje2 = "hola"
```

`charAt(posicion)`, obtiene el carácter que se encuentra en la posición indicada:

```
var mensaje = "Hola";

var letra = mensaje.charAt(0); // letra = H

letra = mensaje.charAt(2);     // letra = l
```

`indexOf(caracter)`, calcula la posición en la que se encuentra el carácter indicado dentro de la cadena de texto. Si el carácter se incluye varias veces dentro de la cadena de texto, se devuelve su primera posición empezando a buscar desde la izquierda. Si la cadena no contiene el carácter, la función devuelve el valor `-1`:

```
var mensaje = "Hola";

var posicion = mensaje.indexOf('a'); // posicion = 3

posicion = mensaje.indexOf('b');     // posicion = -1
```

Su función análoga es lastIndexOf():

`lastIndexOf(caracter)`, calcula la última posición en la que se encuentra el carácter indicado dentro de la cadena de texto. Si la cadena no contiene el carácter, la función devuelve el valor `-1`:

```
var mensaje = "Hola";

var posicion = mensaje.lastIndexOf('a'); // posicion = 3

posicion = mensaje.lastIndexOf('b');     // posicion = -1
```

La función `lastIndexOf()` comienza su búsqueda desde el final de la cadena hacia el principio, aunque la posición devuelta es la correcta empezando a contar desde el principio de la palabra.

`substring(inicio, final)`, extrae una porción de una cadena de texto. El segundo parámetro es opcional. Si sólo se indica el parámetro `inicio`, la función devuelve la parte de la cadena original correspondiente desde esa posición hasta el final:

```
var mensaje = "Hola Mundo";

var porcion = mensaje.substring(2); // porcion = "la Mundo"

porcion = mensaje.substring(5);     // porcion = "Mundo"

porcion = mensaje.substring(7);     // porcion = "ndo"
```

Si se indica un `inicio` negativo, se devuelve la misma cadena original:

```
var mensaje = "Hola Mundo";

var porcion = mensaje.substring(-2); // porcion = "Hola Mundo"
```

Cuando se indica el inicio y el final, se devuelve la parte de la cadena original comprendida entre la posición inicial y la inmediatamente anterior a la posición final (es decir, la posición `inicio` está incluida y la posición `final` no):

```
var mensaje = "Hola Mundo";

var porcion = mensaje.substring(1, 8); // porcion = "ola Mun"

porcion = mensaje.substring(3, 4);     // porcion = "a"
```

Si se indica un `final` más pequeño que el `inicio`, JavaScript los considera de forma inversa, ya que automáticamente asigna el valor más pequeño al `inicio` y el más grande al `final`:

```
var mensaje = "Hola Mundo";

var porcion = mensaje.substring(5, 0); // porcion = "Hola "

porcion = mensaje.substring(0, 5);     // porcion = "Hola "
```

`split(separador)`, convierte una cadena de texto en un array de cadenas de texto. La función parte la cadena de texto determinando sus trozos a partir del carácter `separador` indicado:

```
var mensaje = "Hola Mundo, soy una cadena de texto!";

var palabras = mensaje.split(" ");

// palabras = ["Hola", "Mundo,", "soy", "una", "cadena", "de", "texto!"];
```

Con esta función se pueden extraer fácilmente las letras que forman una palabra:

```
var palabra = "Hola";

var letras = palabra.split(""); // letras = ["H", "o", "l", "a"]
```

6.2 FUNCIONES ÚTILES PARA ARRAYS

A continuación se muestran algunas de las funciones más útiles para el manejo de arrays:

`length`, calcula el número de elementos de un array

```
var vocales = ["a", "e", "i", "o", "u"];

var numeroVocales = vocales.length; // numeroVocales = 5
```

`concat()`, se emplea para concatenar los elementos de varios arrays

```
var array1 = [1, 2, 3];

array2 = array1.concat(4, 5, 6);   // array2 = [1, 2, 3, 4, 5, 6]

array3 = array1.concat([4, 5, 6]); // array3 = [1, 2, 3, 4, 5, 6]
```

`join(separador)`, es la función contraria a `split()`. Une todos los elementos de un array para formar una cadena de texto. Para unir los elementos se utiliza el carácter `separador` indicado

```
var array = ["hola", "mundo"];

var mensaje = array.join(""); // mensaje = "holamundo"

mensaje = array.join(" ");    // mensaje = "hola mundo"
```

`pop()`, elimina el último elemento del array y lo devuelve. El array original se modifica y su longitud disminuye en 1 elemento.

```
var array = [1, 2, 3];

var ultimo = array.pop();

// ahora array = [1, 2], ultimo = 3
```

`push()`, añade un elemento al final del array. El array original se modifica y aumenta su longitud en 1 elemento. (También es posible añadir más de un elemento a la vez)

```
var array = [1, 2, 3];

array.push(4);

// ahora array = [1, 2, 3, 4]
```

`shift()`, elimina el primer elemento del array y lo devuelve. El array original se ve modificado y su longitud disminuida en 1 elemento.

```
var array = [1, 2, 3];

var primero = array.shift();

// ahora array = [2, 3], primero = 1
```

`unshift()`, añade un elemento al principio del array. El array original se modifica y aumenta su longitud en 1 elemento. (También es posible añadir más de un elemento a la vez)

```
var array = [1, 2, 3];

array.unshift(0);

// ahora array = [0, 1, 2, 3]
```

`reverse()`, modifica un array colocando sus elementos en el orden inverso a su posición original:

```
var array = [1, 2, 3];

array.reverse();

// ahora array = [3, 2, 1]
```

6.3 FUNCIONES ÚTILES PARA NÚMEROS

A continuación se muestran algunas de las funciones y propiedades más útiles para el manejo de números.

`NaN`, (del inglés, "Not a Number") JavaScript emplea el valor `NaN` para indicar un valor numérico no definido (por ejemplo, la división `0/0`).

```
var numero1 = 0;

var numero2 = 0;

console.log(numero1/numero2); // se muestra el valor NaN
```

`isNaN()`, permite proteger a la aplicación de posibles valores numéricos no definidos

```
var numero1 = 0;

var numero2 = 0;

if(isNaN(numero1/numero2)) {

  console.log("La división no está definida para los números indicados");

} else {

  console.log("La división es igual a => " + numero1/numero2);

}
```

`Infinity`, hace referencia a un valor numérico infinito y positivo (también existe el valor `-Infinity` para los infinitos negativos)

```
var numero1 = 10;

var numero2 = 0;

console.log(numero1/numero2); // se muestra el valor Infinity
```

`toFixed(digitos)`, devuelve el número original con tantos decimales como los indicados por el parámetro `digitos` y realiza los redondeos necesarios. Se trata de una función muy útil por ejemplo para mostrar precios.

```
var numero1 = 4564.34567;

numero1.toFixed(2); // 4564.35

numero1.toFixed(6); // 4564.345670

numero1.toFixed(); // 4564
```