JAVASCRIPT : OPERADORES

1. 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.

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 ==**).

```
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;
```

Se puede **encadenar operadores** de **asignación** del siguiente modo: `var a = b = c = 0;`

En este caso a, b y c son igual a cero.

Incremento y decremento

Estos dos operadores solamente son **válidos** para las **variables numéricas** y se utilizan para **incrementar** o **decrementar en una unidad** el valor de una variable. Por ejemplo:

```
var numero = 5;
++numero;
alert(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;
alert(numero); // numero = 6
```

De forma análoga, 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;
alert(numero); // numero = 4
```

El anterior ejemplo es equivalente a:

```
var numero = 5;
numero = numero - 1;
alert(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++;
alert(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).

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**.

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;
alert(!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
```

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
```

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
true	false	true
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
```

Los **operadores &&** y **||** se llaman **operadores en cortocircuito** porque **si no se cumple la condición de un término no se evalúa el resto de la operación**.

Por ejemplo: `(a == b && c != d && h >= k)` tiene tres evaluaciones: la **primera** comprueba si la **variable a es igual a b**. **Si no se cumple** esta condición, el **resultado** de la expresión es **falso** y **no se evalúan las otras dos condiciones** posteriores.

En un caso como `( a < b || c != d || h <= k)` **se evalúa si a es menor que b**. **Si se cumple** esta condición el **resultado** de la expresión es **verdadero** y **no se evalúan las otras dos condiciones** posteriores.

Matemáticos

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

Además de los cuatro operadores básicos, JavaScript define otro operador matemático que no es sencillo de entender cuando se estudia por primera vez, pero que es muy útil en algunas ocasiones.

Se trata del **operador "módulo"**, que calcula el **resto de la división entera de dos números**. Si se divide por ejemplo 10 y 5, la **división es exacta** y da un resultado de 2. El resto de esa división es 0, por lo que **módulo de 10 y 5 es igual a 0**.

Sin embargo, si se divide 9 y 5, la **división no es exacta**, el resultado es 1 y el resto 4, por lo que **módulo de 9 y 5 es igual a 4**.

El **operador módulo** en JavaScript se indica mediante el **símbolo %**, que no debe confundirse con el cálculo del porcentaje:

```
var numero1 = 10;
var numero2 = 5;
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
```

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**.
- 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.)

Además de los operadores habituales existe el **operador ===** que se interpreta como "**es estrictamente igual**" y **!==** que se interpreta como "**no es estrictamente igual**". Básicamente sirve para comparar la **igualdad de dos objetos sin forzar la conversión automática de tipos**.

Es decir, que para que devuelva true los dos objetos **además de representar el mismo valor** deberán ser **exactamente** del **mismo tipo** subyacente.

Por ejemplo, si definimos una **variable** y le asignamos una cadena de **texto** que contiene el carácter **"1"**

```
var texto1 = "1"
```

y hacemos la **comparación**

```
texto1 === 1 // false
```

Obtendremos false, es decir, que **no es igual** (porque **un texto no es igual a un número**).

Sin embargo una **comparación** como

```
texto == 1 // true
```

Devolverá true ya que esta **comparación no es estricta** y trata de **realizar automáticamente conversiones** para comprobar si se puede establecer una **equivalencia** entre los dos **valores**. En este caso se **busca el equivalente numérico del texto** y **luego se hace la comparación**, motivo por el cual se obtiene true.

Comparamos el == y el === con varios **ejemplos**:

```
1 == "1" // true
1 === "1" // false (son iguales pero uno es un número y el otro una
cadena)
-1 == true // true
1 == true // true (tanto el -1 como el 1 se consideran equivalentes
a un verdadero cuando se tratan como booleanos)
```

```
-1 === true // false
1 === true // false
1.0 == 1 // true
1.0 === 1 // true también ¡Ojo! JavaScript no distingue subtipos entre
los números por lo que ambos son numéricos y por lo tanto del mismo  tipo.
```

Operador condicional ternario (?:)

Devuelve una de las dos expresiones posibles, **dependiendo** de una **condición**. Su sintaxis es la siguiente:

```
test ? expresion1 : expresion2
```

Donde los parámetros que incluye:

- **test**: Cualquier expresión **booleana**.
- **expresion1**: Expresión que **se devuelve si test es true**.
- **expresion2**: Expresión que **se devuelve si test es false**. Se puede vincular más de una expresión separándolas mediante comas.

El **operador ?:** se puede utilizar como **forma abreviada** de una instrucción **if...else**. Se utiliza normalmente como parte de una expresión mayor en la que una instrucción if...else no sería práctica. **Ejemplo:**

```
var ahora = new Date();
var saludo = ((now.getHours() > 17) ? "Buenas Tardes" : "Buenos Días");
```

En el ejemplo se crea una cadena que contiene "Buenos días" si es más tarde de las 18h. El **código equivalente** que utiliza una instrucción **if...else** tendría el siguiente aspecto:

```
var ahora = new Date();
var saludo = "";
if (now.getHours() > 17){
 saludo += "Buenas Tardes";
}else{
 saludo += "Buenos Días";
}
```

Operadores a nivel de bit

Una **operación bit a bit** (o **Bitwise**) opera sobre números binarios a **nivel de bits individuales**. Es una acción primitiva sustancialmente **más rápida** que las que se llevan a cabo sobre el valor real de los operandos.

Un detalle a tener en cuenta es que estos Bitwise son, en Javascript, **operadores de 32-bits**, lo que significa que en el manejo de valores binarios, un número como 0101 se procesa internamente como 00000000000000000000000000000101. Sin embargo, **todos los ceros de la izquierda pueden despreciarse** ya que, como en el caso de los números decimales, no tienen ningún significado o valor.

Operador	Uso	Descripción rápida
& (AND)	a & b	Devuelve 1 si ambos operandos son 1
| (OR)	a | b	Devuelve 1 donde uno o ambos operandos son 1
^ (XOR)	a ^ b	Devuelve 1 donde un operando, pero no ambos, es 1
~ (NOT)	~a	Invierte el valor del operando
<< (Desplazamiento a la izquierda)	a << b	Desplaza a la izquierda un número especificado de bits.
>> (Desplazamiento a la derecha)	a >> b	Desplaza a la derecha un número especificado de bits. (mantiene el signo)
>>> (Desplazamiento a la derecha con acarreo)	a >>> b	Desplaza la derecha un número especificado de bits descartando los bits desplazados y sustituyéndolos por ceros. (no mantiene el signo)

OPERADOR BIT A BIT AND

El **AND bit a bit** toma **dos números enteros** y realiza la **operación AND lógica en cada par correspondiente** de bits. El resultado en cada posición es 1 si el bit correspondiente de los dos operandos es 1, y 0 de lo contrario:

Valor A	Valor B	A OR B
0	0	1

0	1	0
1	0	0
1	1	1

OPERADOR BIT A BIT OR

Una operación **OR de bit a bit**, o bitwise, toma **dos números enteros** y realiza la **operación OR inclusivo en cada par correspondiente de bits**. El resultado en cada posición es 1 si el bit correspondiente de cualquiera de los dos operandos es 1, y 0 si ambos bits son 0:

Valor A	Valor B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

OPERADOR EXCLUSIVE OR (XOR)

El **XOR bit a bit**, o bitwise, toma **dos números enteros** y realiza la **operación OR exclusivo en cada par correspondiente de bits**. El resultado en cada posición es 1 si el par de bits son diferentes y cero si el par de bits son iguales:

Valor A	Valor B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

OPERADOR DE NEGACIÓN NOT

El **NOT bit a bit**, o bitwise, o complemento, es una **operación unaria** que realiza la **negación lógica en cada bit**, invirtiendo los bits del número, de tal manera que los ceros se convierten en 1 y viceversa:

Valor A	NOT A

0	1
1	0

DESPLAZAMIENTO LÓGICO DE BITS

El **desplazamiento lógico** se usa para **mover bits hacia la izquierda o hacia la derecha para colocarlos en la posición adecuada**. En programación, el **primer operando** representa el **binario** que **queremos modificar** mientras que el **segundo** indica las **posiciones a desplazar**:

Desplazamiento a la Izquierda	Desplazamiento a la Derecha
00011100 << 1 = 00111000	00011100 >> 1 = 00001110
00011100 << 2 = 01110000	00011100 >> 2 = 00000111

En los **desplazamientos hacia la izquierda**, el **bit más significativo** (más a la izquierda) **se pierde**, y se le **asigna un 0 al menos significativo** (el de la derecha). En los **desplazamientos a la derecha** ocurre **lo contrario**.

Hay que tener en cuenta que los **desplazamientos no son rotaciones**; los **bits que salen por la izquierda se pierden mientras que los que entran por la derecha son rellenados con ceros**. Este tipo de **desplazamientos** se denominan **lógicos** a **diferencia** de los **cíclicos** o **rotacionales**.

Delete

Al contrario de lo que se podría pensar, el operador delete **no tiene nada que ver con liberar memoria**. El operador delete **elimina una propiedad de un objeto**. Su sintaxis es la siguiente:

```
delete expresión
```

donde la **expresión** debe **evaluar** una **referencia** de la **propiedad**, por **ejemplo**:

```
var elObjeto = new Object();
elObjeto.id = "10";
elObjeto.nombre = "Objeto de prueba";

delete elObjeto.id
delete elObjeto['id']
```

en general:

```
delete objeto.propiedad;
```

Donde los parámetros que intervienen son:

• **objeto**: El nombre de un objeto, o una expresión que evalúa a un objeto. • **propiedad**: La propiedad a eliminar.

El operador retorna: En modo estricto **arroja una excepción si la propiedad no es configurable** (**retorna false** en modo no estricto). **Retorna true** en cualquier **otro caso**.

Si la operación delete **funciona correctamente**, **eliminará** la **propiedad del objeto** por completo. Sin embargo, si existe otra propiedad con el mismo nombre en la cadena del prototype del objeto, éste heredará la propiedad del prototype.

delete sólo es **efectivo** en **propiedades** de **objetos**. **No tiene** ningún **efecto** en **variables** o en **nombres** de **funciones**.

Aunque a veces son **mal identificados como variables globales**, las **asignaciones que no especifican al objeto** (ejemplo: x = 5), son en realidad **propiedades que se asignan al objeto global**.

delete no puede **eliminar** ciertas **propiedades** de los **objetos predefinidos** (como Object, Array, Math etc).

void

El **operador void** se usa en cualquiera de los siguientes **modos**:

```
void ( unaExpresion )
 void unaExpresion
```

El operador void especifica una **expresión** que **se evalúa sin devolver un valor.** **unaExpresion** es una expresión JavaScript para evaluar. Los **paréntesis** rodeando la expresión son **opcionales**, pero usarlos es una buena práctica al programar.

Puede usar el operador void para especificar una expresión como un **enlace de hipertexto**. La expresión **se evalúa pero no se carga en lugar del documento actual**.

El siguiente código crea un **enlace** de **hipertexto** que **no hace nada** cuando el usuario hace click en él. Cuando el usuario hace click en el enlace, void(0) se evalúa como 0, pero eso **no tiene ningún efecto en JavaScript**.

```
<a href="javascript:void(0)">Haga click aquí para no hacer nada</a>
```

El siguiente código crea un enlace de hipertexto que **envía un formulario cuando el usuario hace click en él**.

```
<a href="javascript:void(document.form.submit())">
Haga click aquí para enviar</a>
```

Precedencia de operadores

La precedencia de operadores determina el **orden** en el cual los **operadores** son **evaluados**. Los operadores con **mayor precedencia son evaluados primero**. **Ejemplo**:

```
3 + 4 * 5 // devuelve 23
```

El operador de multiplicación (*) tiene una precedencia más alta que el operador de suma (+) y por eso será evaluado primero.

Asociatividad

La asociatividad determina el orden en el cual los operadores con el mismo nivel de precedencia son procesados. Por ejemplo:

```
a OP b OP c
```

La asociatividad de **izquierda a derecha** significa que esa expresión es procesada como (a OP b) OP c, mientras que la asociatividad de **derecha a izquierda** significa que es procesada como a OP (b OP c). Los **operadores de asignación** tienen asociatividad de **derecha a izquierda**, por lo que se puede escribir por ejemplo:

```
a = b = 5;
```

para asignar 5 a las dos variables. Esto es porque el operador de asignación retorna el valor que asignó. Primero b es inicializada a 5. Despues a es inicializada al valor de b.

Orden de precedencia	Operador (Símbolo)	Operador en Palabras
1	!, ++, --, ~	No, Incrementa, Decrementa
2	*, /, %, +, -	Multiplicación, División, Módulos, Adición, Substracción
3	<< , >>, >>>	-
4	<, <=, >, >=	Menor que, Menor que o Igual a, Mayor que, Mayor que o Equivalente a
5	==, !=, ===, !==	Igual, Diferente, Estrictamente Equivalente, Estrictamente No Equivalente
6	&, |, ^, &&, ||	Bitwise AND, Bitwise OR, OR Exclusivo Bitwise, AND Lógico, OR Lógico
7	?:	Operador Ternario
8	Operadores de Asignación =, +=, -=, /=, *=, %=, <<=, >>=, >>>=, &=, ^=	Asigna, la cesión de otros operadores