DWC

(Desarrollo Web en entorno cliente)

Índice

1.- Estructuras condicionales

A veces, necesitamos realizar diferentes acciones basadas en diferentes condiciones.

Para hacer eso, podemos usar la condicional `if` y el operador condicional `?`, también llamado operador de "signo de interrogación".

Además, para comprobaciones múltiples podemos utilizar la estructura `switch`

2.- La condicional If

`If` evalúa una condición entre paréntesis y, si el resultado es `true`, ejecuta un bloque de código.

Por ejemplo:

```
let year = prompt('In which year was ECMAScript-2015 specification published?', '');

if (year == 2015) alert( 'You are right!' );
```

En el ejemplo anterior, la condición es una simple verificación de igualdad (`year == 2015`), pero puede ser mucho más compleja.

Si queremos ejecutar más de una declaración, tenemos que “envolver” nuestro bloque de código dentro de llaves:

```
if (year == 2015) {
  alert( "That's correct!" );
  alert( "You're so smart!" );
}
```

Se recomienda envolver su bloque de código con llaves `{}` cada vez que use una instrucción, incluso si solo hay una instrucción para ejecutar. Hacerlo mejora la legibilidad.

2.1.- Conversión booleana

`If` evalúa la expresión entre paréntesis y convierte el resultado en booleano.

Recordemos las reglas de conversión vistas:

- Un número `0`, una cadena vacía `""`, `null`, `undefined`, y `NaN` todos se convierten `false`.
- El resto de valores se convierten a `true`.

Entonces, el código bajo esta condición nunca se ejecutaría:

```
if (0) { // 0 is false
  ...
}
```

y dentro de esta condición, siempre:

```
if (1) { // 1 is true
  ...
}
```

También podemos pasar un valor booleano preevaluado a `if`, como este:

```
let cond = (year == 2015); // equality evaluates to true or false

if (cond) {
  ...
}
```

2.2.- La cláusula else

La declaración `if` puede contener un bloque opcional "else". Se ejecuta cuando la condición es falsa.

Por ejemplo:

```
let year = prompt('In which year was the ECMAScript-2015 specification published?', '');

if (year == 2015) {
  alert( 'You guessed it right!' );
} else {
  alert( 'How can you be so wrong?' ); // any value except 2015
}
```

2.3.- If anidados

A veces, nos gustaría probar varias variantes de una condición. La `else if`cláusula nos permite hacer eso.

Por ejemplo:

```
let year = prompt('In which year was the ECMAScript-2015 specification published?', '');

if (year < 2015) {
  alert( 'Too early...' );
} else if (year > 2015) {
  alert( 'Too late' );
} else {
  alert( 'Exactly!' );
}
```

En el código anterior, JavaScript comprueba primero `year < 2015`. Si eso es falso, pasa a la siguiente condición `year > 2015`. Si eso también es falso, muestra el último `alert`.

Puede haber más bloques `else if`. El `else` final es opcional.

3.- Operador condicional '?'

A veces, necesitamos asignar una variable dependiendo de una condición.

Por ejemplo:

```
let accessAllowed;
let age = prompt('How old are you?', '');

if (age > 18) {
  accessAllowed = true;
} else {
  accessAllowed = false;
}
alert(accessAllowed);
```

El llamado operador "condicional" o "signo de interrogación" nos permite hacerlo de una manera más corta y simple.

El operador está representado por un signo de interrogación `?`. A veces se llama "ternario", porque el operador tiene tres operandos. En realidad, es el único operador en JavaScript que tiene tantos.

La sintaxis es:

```
let result = condition ? value1 : value2;
```

`condition` es evaluado: si es verdadero, entonces `value1` se devuelve, de lo contrario se devuelve `value2`.

Por ejemplo:

```
let accessAllowed = (age > 18) ? true : false;
```

Técnicamente, podemos omitir los paréntesis `age > 18`. El operador de signo de interrogación tiene una precedencia baja, por lo que se ejecuta después de la comparación.

Este ejemplo hará lo mismo que el anterior:

```
// the comparison operator "age > 18" executes first anyway
// (no need to wrap it into parentheses)
let accessAllowed = age > 18 ? true : false;
```

Pero los paréntesis hacen que el código sea más legible, por lo que recomendamos usarlos.

En el ejemplo anterior, puede evitar usar el operador de signo de interrogación porque la comparación en sí misma devuelve `true/false`:

```
// the same
let accessAllowed = age > 18;
```

3.1.- Múltiple '?'

Una secuencia de operadores de signo de interrogación `?` puede devolver un valor que depende de más de una condición.

Por ejemplo:

```
let age = prompt('age?', 18);
```

```
let message = (age < 3) ? 'Hi, baby!' :
  (age < 18) ? 'Hello!' :
  (age < 100) ? 'Greetings!' :
  'What an unusual age!';

alert( message );
```

Puede ser difícil al principio comprender lo que está sucediendo. Pero después de una mirada más cercana, podemos ver que es solo una secuencia ordinaria de pruebas:

1. El primer signo de interrogación verifica si `age < 3`.
2. Si es cierto, devuelve `Hi, baby!'`. De lo contrario, continúa con la expresión después de los dos puntos `'':''`, comprobando `age < 18`.
3. Si eso es cierto, devuelve`' Hello!'`. De lo contrario, continúa con la expresión después de los dos puntos siguientes `'': ''`, comprobando `age < 100`.
4. Si eso es cierto, devuelve `Greetings!'`. De lo contrario, continúa con la expresión después de los últimos dos puntos `'':''`, regresando `'What an unusual age!'`.

Así es como se ve esto usando `if..else`:

```
if (age < 3) {
  message = 'Hi, baby!';
} else if (age < 18) {
  message = 'Hello!';
} else if (age < 100) {
  message = 'Greetings!';
} else {
  message = 'What an unusual age!';
}
```

3.2.- Uso no tradicional de '?'

A veces, el signo de interrogación ? se usa como reemplazo de if:

```
let company = prompt('Which company created JavaScript?', '');

(company == 'Netscape') ? alert('Right!') : alert('Wrong.');
```

Dependiendo de la condición company == 'Netscape', la primera o la segunda expresión después de la ? se ejecuta y muestra una alert.
No asignamos un resultado a una variable aquí. En su lugar, ejecutamos un código diferente según la condición.
No se recomienda utilizar el operador de signo de interrogación de esta manera.

La notación es más corta que la declaración if equivalente, que atrae a algunos programadores. Pero es menos legible.
Aquí está el mismo código que usa if para la comparación:

```
let company = prompt('Which company created JavaScript?', '');

if (company == 'Netscape') {
  alert('Right!');
} else {
  alert('Wrong.');
}
```

Nuestros ojos miran el código verticalmente. Los bloques de código que abarcan varias líneas son más fáciles de entender que un conjunto de instrucciones horizontal largo.
El propósito del operador de signo de interrogación ? es devolver un valor u otro dependiendo de su condición. Se recomienda usarlo exactamente para eso.

4.- La condicional múltiple “Switch”

Una declaración switch puede reemplazar múltiples comprobaciones if.

Ofrece una forma más descriptiva de comparar un valor con múltiples variantes.

La condicional múltiple switch presenta uno o más bloques case y un default opcional.

```
switch(x) {
  case 'value1':  // if (x === 'value1')
    ...
    [break]

  case 'value2':  // if (x === 'value2')
    ...
    [break]

  default:
    ...
    [break]
}
```

- Se verifica el valor x de una estricta igualdad con el valor del primer case (es decir, value1, luego con el segundo (value2) y así sucesivamente.
- Si se encuentra la igualdad, switch comienza a ejecutar el código a partir del correspondiente case, hasta el break más cercano (o hasta el final de switch).
- Si ningún caso coincide, entonces el código default se ejecuta (si existe).

```
let a = 2 + 2;

switch (a) {
  case 3:
    alert( 'Too small' );
    break;
  case 4:
```

```
    alert( 'Exactly!' );
    break;
  case 5:
    alert( 'Too large' );
    break;
  default:
    alert( "I don't know such values" );
}
```

Si no existe `break`, la ejecución continúa con la siguiente `case` sin ninguna comprobación.

Un ejemplo sin `break`:

```
let a = 2 + 2;

switch (a) {
  case 3:
    alert( 'Too small' );
  case 4:
    alert( 'Exactly!' );
  case 5:
    alert( 'Too big' );
  default:
    alert( "I don't know such values" );
}
```

En el ejemplo anterior veremos la ejecución secuencial de tres `alerts`:

```
alert( 'Exactly!' );
alert( 'Too big' );
alert( "I don't know such values" );
```

Cualquier expresión puede ser un argumento `switch/case`.

Ambos `switch` y `case` permiten expresiones arbitrarias.

Por ejemplo:

```
let a = "1";
let b = 0;

switch (+a) {
  case b + 1:
    alert("this runs, because +a is 1, exactly equals b+1");
    break;

  default:
    alert("this doesn't run");
}
```

Aquí `+a` da `1`, que se compara con `b + 1` en el `case`, y se ejecuta el código correspondiente.

4.1.- Agrupación de "cases"

Se pueden agrupar varios `case` distintos y que compartan el mismo código.

Por ejemplo, si queremos que se ejecute el mismo código `case 3` y `case 5`:

```
let a = 3;

switch (a) {
  case 4:
    alert('Right!');
    break;
  case 3: // (*) grouped two cases
  case 5:
    alert('Wrong!');
    alert("Why don't you take a math class?");
    break;

  default:
    alert('The result is strange. Really.');
}
```

Ahora 3y 5 muestran el mismo mensaje.

La capacidad de "agrupar" cases es un efecto secundario de cómo switch/case funciona sin break. Aquí la ejecución de case 3 comienza desde la línea (*)y continúa case 5, porque no hay break.

4.2.- El tipo en el case

Enfaticemos que la verificación de igualdad siempre es estricta. Los valores deben ser del mismo tipo para que coincidan.

Por ejemplo, consideremos el código:

```
let arg = prompt("Enter a value?");
switch (arg) {
  case '0':
  case '1':
    alert( 'One or zero' );
    break;

  case '2':
    alert( 'Two' );
    break;

  case 3:
    alert( 'Never executes!' );
    break;
  default:
    alert( 'An unknown value' );
}
```

1. Para 0, 1 se ejecuta la primera alert.
2. Para 2 se ejecuta la segunda alert.
3. Pero para 3, el resultado de prompt es una cadena "3", que no es estrictamente igual === al número 3. ¡Así que tenemos un código muerto case 3! En este caso cuando en el prompt se introduzca 3 se ejecutará default.

5.- Bucles

A menudo necesitamos repetir acciones.

Por ejemplo, generar productos de una lista uno tras otro o simplemente ejecutar el mismo código para cada número del 1 al 10.

Los bucles son una forma de repetir el mismo código varias veces.

5.1.- El bucle "while"

El `while` bucle tiene la siguiente sintaxis:

```
while (condition) {
  // code
  // so-called "loop body"
}
```

Si bien `condition` es verdad, `code` se ejecuta el cuerpo del bucle.

Por ejemplo, el siguiente ciclo sale imientras `i < 3`:

```
let i = 0;
while (i < 3) { // shows 0, then 1, then 2
  alert( i );
  i++;
}
```

Una ejecución única del cuerpo del bucle se llama *iteración* . El bucle en el ejemplo anterior hace tres iteraciones.

Si `i++` faltara en el ejemplo anterior, el ciclo se repetiría (en teoría) para siempre.

En la práctica, el navegador proporciona formas de detener dichos bucles, y en JavaScript del lado del servidor, podemos eliminar el proceso.

Cualquier expresión o variable puede ser una condición de bucle, no solo comparaciones: la condición es evaluada y convertida a booleana por `while`.

Por ejemplo, una forma más corta de escribir `while (i != 0)`es `while (i)`:

```
let i = 3;
while (i) { // when i becomes 0, the condition becomes falsy, and the loop stops
  alert( i );
  i--;
```

```
}
```

Las llaves no son necesarias para un cuerpo de una sola línea.

Si el cuerpo del bucle tiene una sola declaración, podemos omitir las llaves {…}:

```
let i = 3;
while (i) alert(i--);
```

5.2.- El bucle "do ... while"

La verificación de condición se puede mover *debajo* del cuerpo del bucle utilizando la `do..while` sintaxis:

```
do {
  // loop body
} while (condition);
```

El bucle primero ejecutará el cuerpo, luego verificará la condición y, si bien es cierto, lo ejecutará una y otra vez.

Por ejemplo:

```
let i = 0;
do {
  alert( i );
  i++;
} while (i < 3);
```

Esta forma de sintaxis solo debe usarse cuando desea que el cuerpo del bucle se ejecute **al menos una vez,** independientemente de que la condición sea verdadera. Por lo general, se prefiere la otra forma: `while(…) {…}`.

5.3.- El bucle "for"

El forbucle es más complejo, pero también es el bucle más utilizado.

Se parece a esto:

```
for (begin; condition; step) {
  // ... loop body ...
}
```

Aprendamos el significado de estas partes con el ejemplo. El siguiente bucle muestra un alert(i) para ide 0 hasta (pero no incluyendo) 3:

```
for (let i = 0; i < 3; i++) { // shows 0, then 1, then 2
  alert(i);
}
```

Examinemos la declaración for parte por parte:

parte		
empezar	i = 0	Se ejecuta una vez al ingresar al bucle.
condición	i < 3	Comprobado antes de cada iteración de bucle. Si es falso, el ciclo se detiene.
cuerpo	alert(i)	Se ejecuta una y otra vez mientras la condición es verdadera.
paso	i++	Se ejecuta después del cuerpo en cada iteración.

Declaración de variable en línea

Aquí, la variable "contador" ise declara en el bucle. Esto se llama una declaración de variable "en línea". Dichas variables son visibles solo dentro del bucle.

```
for (let i = 0; i < 3; i++) {
  alert(i); // 0, 1, 2
}
alert(i); // error, no such variable
```

En lugar de definir una variable, podríamos usar una existente:

```
let i = 0;

for (i = 0; i < 3; i++) { // use an existing variable
  alert(i); // 0, 1, 2
}

alert(i); // 3, visible, because declared outside of the loop
```

5.3.1.- Omitir partes del for

Cualquier parte de `for` se puede omitir.

Por ejemplo, podemos omitir `begin` si no necesitamos hacer nada al inicio del ciclo.

Como aquí:

```
let i = 0; // we have i already declared and assigned

for (; i < 3; i++) { // no need for "begin"
  alert( i ); // 0, 1, 2
}
```

También podemos eliminar `step`:

```
let i = 0;

for (; i < 3;) {
  alert( i++ );
}
```

Esto hace que el bucle sea idéntico a `while (i < 3)`.

De hecho, podemos eliminar todo, creando un bucle infinito:

```
for (;;) {
  // repeats without limits
}
```

Tenga en cuenta que los dos puntos ; y comas deben estar presentes en el for . De lo contrario, habría un error de sintaxis.

5.3.2.- Salir de un bucle

Normalmente, un bucle sale cuando su condición se vuelve falsa.

Pero podemos forzar la salida en cualquier momento usando la directiva break especial.

Por ejemplo, el siguiente bucle le pide al usuario una serie de números, "rompiendo" cuando no se ingresa ningún número:

```
let sum = 0;
while (true) {
  let value = +prompt("Enter a number", '');
  if (!value) break; // (*)
  sum += value;
}
alert( 'Sum: ' + sum );
```

La directiva break se activa en la línea (*) si el usuario ingresa una línea vacía o cancela la entrada. Detiene el bucle de inmediato, pasando el control a la primera línea después del bucle el alert.

La combinación "bucle infinito + break según sea necesario" es ideal para situaciones en las que la condición de un bucle debe verificarse no al principio o al final del bucle, sino en el medio o incluso en varios lugares de su cuerpo.

5.3.3.- Continuar a la siguiente iteración

La directiva continue es una "versión más ligera" de break. No detiene todo el ciclo. En cambio, detiene la iteración actual y obliga al bucle a iniciar una nueva (si la condición lo permite).

Podemos usarlo si hemos terminado con la iteración actual y nos gustaría pasar a la siguiente.

El siguiente bucle usa continue para generar solo valores impares:

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

```
  // if true, skip the remaining part of the body
  if (i % 2 == 0) continue;
  alert(i); // 1, then 3, 5, 7, 9
}
```

Para valores pares de i, la directiva continue deja de ejecutar el cuerpo y pasa el control a la siguiente iteración de for (con el siguiente número). Entonces el alert solo se llama para valores impares.

La directiva continue ayuda a disminuir el anidamiento

Un bucle que muestra valores impares podría verse así:

```
for (let i = 0; i < 10; i++) {
  if (i % 2) {
    alert( i );
  }
}
```

Desde un punto de vista técnico, esto es idéntico al ejemplo anterior. Seguramente, podemos envolver el código en un bloque if en lugar de usar continue.

Pero como efecto secundario, esto creó un nivel más de anidamiento (la llamada alert dentro de las llaves). Si el código dentro de if es más largo que unas pocas líneas, eso puede disminuir la legibilidad general.

El condicional '?' y break y continue

Las directivas como break/continue no están permitidas con el operador ?

Por ejemplo, si tomamos este código:

```
if (i > 5) {
  alert(i);
} else {
  continue;
}
```

y lo reescríbimos usando un signo de interrogación:

```
(i > 5) ? alert(i) : continue; // continue isn't allowed here
```

deja de funcionar: hay un error de sintaxis.

Esta es solo otra razón para no usar el operador de signo de interrogación ? en lugar de if.

5.3.4.- Uso de etiquetas

A veces necesitamos salir de múltiples bucles anidados a la vez.

Por ejemplo, en el siguiente código, hacemos un bucle i y j solicitamos las coordenadas (i, j)de (0,0)a (2,2):

```
for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    let input = prompt(`Value at coords (${i},${j})`, '');
    // what if we want to exit from here to Done (below)?
  }
}

alert('Done!');
```

Necesitamos una forma de detener el proceso si el usuario cancela la entrada.

Una *etiqueta* es un identificador con dos puntos antes de un bucle:

```
labelName: for (...) {
  ...
}
```

La declaración break <labelName> en el bucle a continuación se desglosa en la etiqueta:

```
outer: for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    let input = prompt(`Value at coords (${i},${j})`, '');
    // if an empty string or canceled, then break out of both loops
    if (!input) break outer; // (*)
    // do something with the value...
```

```
  }
}
alert('Done!');
```

En el código anterior, `break outer` busca hacia arriba la etiqueta nombrada `outer` y sale de ese bucle.

6.- Resumen de Bucles

Hay 3 tipos de bucles:

- `while` - La condición se verifica antes de cada iteración.
- `do..while` - La condición se verifica después de cada iteración.
- `for (;;)` - La condición se verifica antes de cada iteración, hay configuraciones adicionales disponibles.

Para hacer un bucle "infinito", generalmente se utiliza la construcción `while(true)`. Tal bucle, como cualquier otro, se puede detener con la directiva `break`.
Si no queremos hacer nada en la iteración actual y nos gustaría reenviar a la siguiente, podemos usar la directiva `continue`.
Una etiqueta es la única forma de `break/continue` para salir de un bucle anidado para ir a uno externo.

Más adelante veremos más estructuras relacionadas con lo bucles que permiten trabajar con colecciones de elementos u objetos iterables