Modulo 4. Control de flujo, condicionales y bucles

If

La instrucción if es la primera y más simple instrucción de control de flujo disponible en JavaScript. Tiene algunas formas, pero en su forma básica, verifica una condición determinada y, según su valor booleano, ejecuta un bloque de código o lo omite. La sintaxis se ve así:

```
if (condición) {
    bloque de código
}
```

If else

La instrucción if es muy útil, pero ¿qué pasa si también queremos ejecutar algún código cuando no se cumple una condición dada? Por supuesto, podríamos usar otra declaración if y cambiar la condición.

```
let isUserReady = confirm("¿Estás listo?");

if (isUserReady) {
    console.log("¡Usuario listo!");
}

if (isUserReady == false) {
    console.log("¡Usuario no esta listo!");
}
```

La palabra clave `else` es una parte opcional de la instrucción if y nos permite agregar un segundo bloque de código que se ejecutará solo cuando NO se cumpla la condición.

```
if (condición) {
  condición - código verdadero
} else {
  condición - código falso
}

Usando la nueva sintaxis, podemos reescribir nuestro ejemplo anterior:
```

```
let isUserReady = confirm("¿Estás listo?");

if (isUserReady) {
    console.log("¡Usuario listo!");
} else {
    console.log("¡Usuario no esta listo!");
}
```

If else if

Las instrucciones `if` y `if... else` nos dan una gran flexibilidad en cómo se puede comportar el código en relación con cualquier otra cosa. Pero ramificar el flujo de ejecución del código solo en dos ramas a veces no es suficiente. Hay una solución simple para esto en JavaScript: podemos anidar instrucciones `if ... else`. ¿Como funciona esto? Un segmento else puede tener una sentencia `if` o `if... else` dentro de él, y es posible anidar cualquier número de sentencias `if... else` si es necesario.

```
if (condición_1) {
  code
} else if (condición_2) {
  code
} else if (condición_3) {
  code
} else {
  code
}
```

Operador ternario

El operador ternario sirve como un if else, de esta forma este codigo

```
let price = 100;
let shippingCost;
if (price > 50) {
    shippingCost = 0;
} else {
    shippingCost = 5;
}
console.log(`price = ${price}, shipping = ${shippingCost}`); // -> price = 100, shipping = 0
```

Se puede reducir a esto

```
let price = 100;
let shippingCost = price > 50 ? 0 : 5;

console.log(`price = ${price}, shipping = ${shippingCost}`); // -> price = 100, shipping = 0
```

Switch

El último tipo de sentencia que se utiliza para la ejecución de código condicional es una setencia switch. En comparación con la instrucción if, no es una sentencia que se use con mucha frecuencia. Es algo similar a las sentencias if... else anidadas, pero en lugar de evaluar diferentes expresiones, switch evalúa una expresión condicional y luego intenta hacer coincidir su valor con uno de los casos dados. Esta es la sintaxis de la sentencia switch:

```
switch (expresión) {
    case primera_coincidencia:
        código
        break;
    case segunda_coincidencia:
        código
        break;
    default:
        código
}
```

Comienza con la palabra clave reservada `switch` seguida de la expresión a evaluar entre paréntesis. El siguiente es el bloque de código que tiene una o más cláusulas de caso (técnicamente es posible tener cero casos, pero esto no sería muy útil) seguido directamente por un valor correspondiente a este caso y un carácter de dos puntos. Después de los dos puntos, hay un bloque de código que se ejecutará cuando la expresión se evalúe con este valor de caso. El bloque de código termina opcionalmente con la palabra clave `break`. Todos los casos siguen la misma plantilla.

Además, puede estar presente un caso especial llamado `default` (por convención ubicado al final de la instrucción switch; sin embargo, no es obligatorio). El caso `default` se ejecuta cuando ninguno de los casos coincide con la expresión. La evaluación en sí se realiza con un operador de comparación estricto `(===)` por lo que no solo debe coincidir el valor, sino también el tipo de valor de caso y la expresión.

Bucles

En la programación, los bucles son la segunda forma de las sentencias de control de flujo. Junto con las sentencias de ejecución condicional como el if y switch, permiten una libertad casi infinita sobre cómo puede funcionar la aplicación desde un punto de vista algorítmico. Si bien las sentencias condicionales pueden cambiar el comportamiento del código (permitiéndonos decidir durante la ejecución del programa si se debe ejecutar o no una determinada parte del código), los bucles son una manera fácil de repetir cualquier fragmento del código tantas veces como queramos, o hasta que se cumpla alguna condición. Existen diferentes tipos de bucles en JavaScript, pero podemos dividirlos en dos categorías:

- Bucles que se repiten un número determinado de veces. (for)
- Bucles que continúan hasta que se cumple alguna condición. (while)

While

El bucle `while` es tan versátil que alguien lo suficientemente persistente podría reemplazar todas las demás sentencias de control de flujo con bucles while, incluso sentencias `if`. Por supuesto, sería engorroso en el mejor de los casos. El bucle while es uno de los bucles que normalmente usamos cuando no sabemos exactamente cuántas veces se debe repetir algo, pero sabemos cuándo parar. La sintaxis del bucle `while` es la siguiente:

```
while(condición) {
    bloque de código
}
```

La expresión entre paréntesis se evalúa al comienzo de cada iteración del bucle. Si la condición se evalúa como verdadera, se ejecutará el código entre llaves. Luego, la ejecución vuelve al comienzo del bucle y la condición se evalúa nuevamente. El bucle iterará y el código se ejecutará siempre que la condición se evalúe como verdadera. Esto significa, por supuesto, que con una condición mal formada, un bucle `while` puede convertirse en un bucle infinito, un bucle sin final. Dependiendo del contexto, eso puede ser lo que queremos lograr. Casi todos los juegos de computadora, por ejemplo, son básicamente bucles infinitos que se repiten: leer la entrada del jugador, actualizar el estado del juego, mostrar en la pantalla tantas veces por segundo como sea necesario. Esta es una gran simplificación, pero no obstante es cierta.

Do while

El bucle `do ... while` es muy similar al bucle simple `while`, la principal diferencia es que en un bucle while, la condición se verifica antes de cada iteración , y en el bucle `do ... while`, la condición se verifica después de cada iteración. Esto no parece una gran diferencia, pero la consecuencia de esto es que un código de bucle `do ... while` siempre se ejecuta al menos una vez antes de que se realice la primera verificación de condición, y un < codel>while nunca se puede ejecutar si la condición inicial se evalúa como falsa al comienzo del bucle. La sintaxis del bucle `do ... while` es la siguiente:

```
do {
    bloque de código
} while(condición);
```

For

El bucle for forma parte del segundo tipo de bucles, el que es mejor en situaciones en las que sabemos cuántas veces se debe ejecutar el bucle (aunque esto no es necesario). La sintaxis del bucle for es un poco más complicada y tiene el siguiente aspecto:

```
for (inicialización; condición; incremento) {
    bloque de código
}
```

En los paréntesis después de la palabra for, esta vez no encontraremos una sola condición, como sucedió en el bucle while. El interior de los paréntesis se divide en tres campos por punto y coma, y a cada campo se le asigna un significado diferente. En cada uno de estos campos aparece una sentencia, que se interpretará de la siguiente manera:

- Inicialización del bucle.
- Condición del bucle.
- Incremento del bucle.

Las tres sentencias son opcionales, pero de hecho rara vez se nos escapa alguna.

La inicialización se ejecuta solo una vez, antes de la primera iteración del bucle. Por lo general, se usa para inicializar (o declarar e inicializar) una variable que se usará como contador de bucle. Podemos usar cualquier variable existente como contador, pero en general es una buena práctica declarar una nueva, ya que esto hace que el código sea

más limpio y más fácil de leer y entender. La declaración de la inicialización es opcional y se puede dejar vacía, excepto si termina con un punto y coma.

Una condición es una expresión que se evalúa como un valor booleano antes de cada iteración del bucle. Si esta expresión se evalúa como verdadera, el bucle ejecutará su código. En el caso de que la condición se evalúe como falsa, el bucle finaliza y no se ejecutarán más iteraciones, y la ejecución del código salta a la primera sentencia después del bucle for. La condición también es opcional, pero si se deja vacía, se asumirá que siempre es verdadera, lo que generará un bucle infinito si no se utiliza ningún otro medio para salir del bucle.

El incremento siempre se ejecuta al final de la iteración del bucle actual y, la mayoría de las veces, se usa para incrementar (o disminuir, según la necesidad) un contador que se usa en una condición. Puede ser cualquier expresión, no solo incremento/decremento. Esto también se puede dejar vacío.

```
for (let i = 0; i < 10; i++) {
    console.log(i);
}
```

For of

Además del bucle for regular, hay dos versiones específicas, una de las cuales, for... of, está dedicada para usar con arreglos. En un bucle de este tipo, no especificamos explícitamente ninguna condición ni número de iteraciones, ya que se realiza exactamente tantas veces como elementos haya en el arreglo indicado. Este es un pequeño ejemplo

```
let values = [10, 30, 50, 100];
let sum = 0;
for (let number of values) {
    sum += number;
}
console.log(sum); // -> 190
```

Entre corchetes después de la palabra for, no encontrarás tres campos separados por punto y coma. Hay una declaración de variable, seguida de la palabra of y luego un arreglo, cuyos elementos recorreremos (variable o literal). En nuestro ejemplo, for (let number of values) significa que la variable number contendrá los elementos subsiguientes del arreglo values en cada iteración. La sintaxis de este bucle es la siguiente:

```
for (variable of arreglo) {
    bloque de código
}
```

For in

También existe una versión del bucle `for` que nos permite recorrer campos de objetos. La sintaxis es for ... in. Itera a través de todos los campos del objeto indicado, colocando los nombres de estos campos (o claves) en la variable. En el ejemplo, usamos un objeto que contiene datos sobre un usuario:

```
let user = {
    name: "Calvin",
    surname: "Hart",
    age: 66,
    email: "CalvinMHart@teleworm.us"
};

for (let key in user) {
    console.log(key); // -> name, surname, age, email
};
```

En cada iteración del bucle, los nombres de los campos sucesivos del objeto de usuario se asignan a la variable key, es decir: nombre, apellido, edad, correo electrónico. Luego los escribimos en la consola. Pero, ¿y si queremos recuperar los valores almacenados en los campos con estos nombres? Para obtener acceso al campo especificado, usamos la notación de puntos, es decir, después del nombre del objeto, escribimos un punto y el nombre del campo (key). La clave dada en esta notación siempre se trata como un literal. En el bucle for... in, este enfoque no funcionará porque el nombre del campo (key) se coloca en una variable. Afortunadamente, tenemos una solución alternativa, la notación entre paréntesis. Te permite referirte al campo del objeto seleccionado usando corchetes (como en los arreglos). En el corchete detrás del nombre del objeto, colocamos el nombre del campo, que puede ser un literal o una variable que contenga ese nombre.

```
for (let key in user) {
    console.log(`${key} -> ${user[key]}`);
};
```

Break y continue

Continue se puede usar en bucles (pero no en un switch). Cuando se usa, se aplica al bucle circundante más cercano. La instrucción continue, a diferencia de break, no finaliza todo el bucle, sino que inicia la siguiente iteración de este bucle. Podemos pensar en ello como saltar directamente al final de la iteración actual.

Tanto break como continue no se usan con mucha frecuencia. Definitivamente no deberíamos usarlos cuando podemos terminar el bucle con una condición construida correctamente. Son útiles en situaciones de emergencia, cuando sucede algo inusual en bucles con iteraciones más complejas.

```
for (let i = 0; i < 10; i++) {
    if (i == 3) {
        continue;
    }
    console.log(i);
}
```

Cuando un intérprete de JavaScript llega a un `break`, saltará fuera de la sentencia `switch` actual.

La última parte importante es que si se necesita un código más complejo dentro de un caso determinado, debemos colocar ese código en bloques de código separados rodeándolo adicionalmente con llaves. Esto aumentará la legibilidad del código y permitirá la declaración de variables solo en el alcance del caso dado.

```
let gate = prompt("Elige una puerta: a, b, o c");
let win = false;

switch (gate) {
    case "a": {
        let message = "Puerta A";
        console.log(message);
        break;
    }
    case "b": {
        let message = "Puerta B";
        console.log(message);
        break;
    }
    case "c": {
        let message = "Puerta C";
        console.log(message);
        break;
    }
    default:
        alert("No existe la puerta " + String(gate));
}
```

```
if (win) {
    alert("¡Ganador!");
}
```