ES6 Loops

Looping statements in the programming languages help to execute the set of instructions/functions repeatedly while a condition evaluates to true. Loops are an ideal way to perform the execution of certain conditions repeatedly. In a loop, the repetition is termed as **iteration.**

You can see the classification of loops in the following figure:

Let us try to understand the loops in the above image.

Definite Loops

A definite loop has a definite/fixed number of iterations. In [ES6](https://www.javatpoint.com/es6), there are three types of definite loops that are listed below:

|  |  |
| --- | --- |
| **Definite Loop** | **Description** |
| **for( ; ; ) Loop** | It executes the block of code for a definite number of times. |
| **for…in Loop** | It iterates through the properties of the object. |
| **for…of loop** | Unlike object literals, it iterates the iterables (arrays, string, etc.). |

Let us try to elaborate on the above loops.

The for ( ; ; ) loop

The **for ( ; ; ) loop** is used to iterate a part of the program multiple times. If you have a fixed number of iterations, then it is always recommended to use 'for' loop.

**Syntax**

1. **for**(initialization;condition;incr/decr){
2. //statement or code to be executed
3. }

The **'for' loop** includes some parts that are defined as follows:

* **Initialization:** It is the initial condition that is executed once at the starting of the loop. In this part, we initialize the variable, or it can also be used for the already initialized variable. It is an optional statement.
* **Condition:** It is executed each time for testing the condition of the loop. It continues the execution of the loop until the condition is false. It returns only Boolean values that are either true or false. It is also an optional statement.
* **Increment/Decrement:** It can increment or decrement the value of the variable, and it is also an optional statement.
* **Statement:** It represents the body of the loop, which is executed every time until the conditional expression is false.

**Flowchart**

**Example**

In the following, there are three examples of **'for' loop** in which we are showing the **simple 'for' loop, 'for' loop** having **multiple expressions** and **infinite 'for' loop.**

* The table of 2 by using for loop.

1. var i;
2. **for**(i=1;i<=10;i++)
3. {
4. console.log("2 x "+ i +" =", 2\*i);
5. }

**Output**

2 x 1 = 2

2 x 2 = 4

2 x 3 = 6

2 x 4 = 8

2 x 5 = 10

2 x 6 = 12

2 x 7 = 14

2 x 8 = 16

2 x 9 = 18

2 x 10 = 20

* **for loop** having **multiple expressions**  
  You can combine the multiple assignments and final expressions in a single for loop by using the **comma (, ) operator.**  
  Let us try to print the Fibonacci series by using the single for loop.

1. "use strict"
2. **for**(let temp, a = 0, b = 1; b<40; temp = a, a = b, b = a + temp)
3. console.log(b);

**Output**

1

1

2

3

5

8

13

21

34

* Infinite for loop  
  The example of **infinite for loop** is given as follows:

1. **for**(;;)
2. {
3. console.log("infinitive loop");  // It will print infinite times
4. }

**Output**

infinite loop

infinite loop

infinite loop

infinite loop

infinite loop

infinite loop

infinite loop

infinite loop

infinite loop

infinite loop

infinite loop

infinite loop

.

.

.

.

infinite loop

For terminating it, you can use ctrl + c.

The for…in loop

The **for…in loop** is similar to **for loop,** which iterates through the properties of an object, i.e., when you require to visit the properties or keys of the object, then you can use **for…in loop.** It is a better choice when you are working with objects or dictionaries where the order of index is not essential.

**Syntax**

1. **for** (variable\_name in object\_name) //Here in is the keyword
2. {
3. // statement or block to execute
4. }

In every iteration, one property from the object is assigned to the name of the variable, and this loop continues till all of the object properties get covered.

**Example**

1. function Mobile(model\_no){
2. **this**.Model = model\_no;
3. **this**.Color = 'White';
4. **this**.RAM = '4GB';
5. }
6. var Samsung = **new** Mobile("Galaxy");
7. **for**(var props in Samsung)
8. {
9. console.log(props+ " : " +Samsung[props]);
10. }

**Output**

Model : Galaxy

Color : White

RAM : 4GB

If you pass the function in the properties of the object then this loop will give you the complete function in the output. You can it's illustration in the following code:

1. function Mobile(model\_no){
2. **this**.Model = model\_no;
3. **this**.Color = 'White';
4. **this**.RAM = '4GB';
5. **this**.Price = function price() // The loop will give you this function as it is written here.
6. {
7. console.log(**this**.model + "Price = Rs. 3300");
8. }
9. }
10. var Samsung = **new** Mobile("Galaxy");
11. **for**(var props in Samsung)
12. {
13. console.log(props+ " : " +Samsung[props]);
14. }

**Output**

Model : Galaxy

Color : White

RAM : 4GB

Price : function price()

{

console.log(this.model + "Price = Rs. 3300");

}

So, you can also visit the methods by using the **for…in loop.**

The for…of loop

Unlike the object literals, this loop is used to iterate the iterables (arrays, string, etc.).

**Syntax**

1. **for**(variable\_name of object\_name) // Here of is a keyword
2. {
3. //statement or block to execute
4. }

In every iteration, one property from the iterables gets assigned to the variable\_name, and the loop continues till the end of the iteration.

**Example**

1. var fruits = ['Apple', 'Banana', 'Mango', 'Orange'];
2. **for**(let value of fruits)
3. {
4. console.log(value);
5. }
7. /\*
8. You can also write the above code like:
10. for(let value of ['Apple', 'Banana', 'Mango', 'Orange'])
11. {
12. console.log(value);
13. }
15. \*/

**Output**

Apple

Banana

Mango

Orange

Indefinite Loops

An indefinite loop has infinite iterations. It is used when the number of iterations within the loop is intermediate or unknown.

There are two types of indefinite loops that are listed below:

|  |  |
| --- | --- |
| **Indefinite Loops** | **Description** |
| **while Loop** | It executes the instructions each time till the defined condition evaluates to true. |
| **do…while Loop** | It is similar to the while loop, but the key difference is that the do…while loop executes the loop at once irrespective of the terminator condition. |

Let us try to elaborate on the above loops.

The while loop

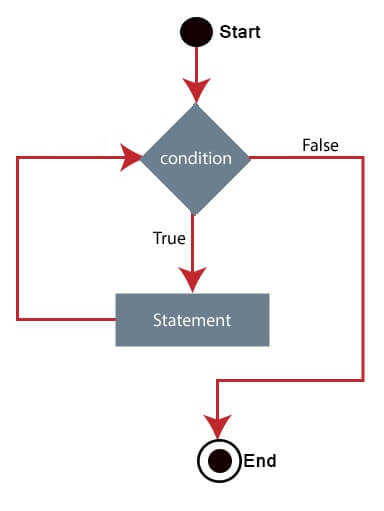
A while loop is a control flow statement that allows the repeated execution of code based on the given Boolean condition. It consists of a block of code and an expression/condition.

The while loop checks the expression/condition before the execution of block; that's why this control structure is often known as a pre-test loop.

**Syntax**

1. **while** (condition) {
2. statements;
3. }

**Flowchart**



**Example**

1. var y = 0;
2. **while** (y < 4) {
3. console.log(y);
4. y++;
5. }

**Output**

0

1

2

3

**Points to remember**

* The **condition** is always required in the while loop because it is necessary to run the loop. If the condition returns true, then the loop will start over again, but if it returns false, the loop will stop.
* If the condition will always true, then the loop will never end.

The do…while loop

It is a control flow statement that executes a block of code at least once, and then it will depend upon the condition whether or not the loop executes the block repeatedly.

The do…while loop checks the condition after the execution of the block, that's why this control structure is also known as the **post-test loop.** It is also possible that the condition always evaluates to true, which will create an **infinite loop.**

**Syntax**

1. **do**
2. {
3. // block of statements to be executed;
4. }
5. **while** (expression);

**Flowchart**

**Example**

1. var count = 6, fact = 1;
2. **do** {
3. fact = fact \* count--;
4. } **while** (count > 0);
6. console.log(fact);

**Output**

720

If you perform this example by using the **while loop,** then this will be written as:

1. var count = 6, fact = 1;
2. **while** (count > 0)
3. {
4. fact = fact \* count--;
5. }
6. console.log(fact);

**Output**

720

The key difference between the above two examples is that the **while loop** is entered only if the condition passed to it will evaluate to true. But the **do…while loop** executes the statement once, it happens because of the starting iteration of the **do…while loop** does not consider as the Boolean expression. Then for the further iterations, the while will check the condition and takes the control out from the loop.

The Loop control statements

The loop control statements are used for interrupting or control the flow of the execution. These statements change the execution from its normal sequence. JavaScript provides you the full control to handle the loops and switch statements.

There may be some circumstances when you require to come out from the loop without reaching its bottom. There may also be some situations when you need to skip the part of the code and start the further iterations of the loop. So, for handling such situations in JavaScript, we have **a break** and **continue** statements.

|  |  |
| --- | --- |
| **Loop control statements** | **Description** |
| **The break statement** | The break statement is used to bring the control of the program out from the loop. |
| **The continue statement** | It skips the subsequent statements of the current iteration and brings control of the program to the beginning of the loop. |

Let us try to elaborate on the above control statements.

The break statement

It is used to take control of the program out from the loop. You can use the break statements within the loops or in the switch statements. Using a break statement within the loop causes the program to exit from the loop.

**Syntax**

1. **break**;

**Example**

1. var n = 1;
2. **while**(n<=7)
3. {
4. console.log("n="+n);
5. **if**(n==4)
6. {
7. **break**;
8. }
9. n++;
10. }

The above code will print the four values of **n** for the range of numbers within 1 to 7.

When the value of n is **4,** then the loop forces the control to exit the loop because of the **break statement.** You will get the following output after the successful execution of the above code.

**Output**

n=1

n=2

n=3

n=4

The continue statement

Unlike the break statement, the continue statement does not exit from the loop. It terminates the current iteration of the loop and starts the next iteration.

**Syntax**

1. **continue**;

**Example**

1. var n = 0;
2. **while**(n<=5)
3. {
4. n++;
5. **if**(n==3)
6. {
7. **continue**;
8. }
9. console.log("n = "+n);
10. }

The above example will display the values of **n,** but it will skip the current iteration if the value of **n** will be **3.** You will get the following output after the successful execution of the above code.

**Output**

n = 1

n = 2

n = 4

n = 5

n = 6

Use of Labels for controlling the flow

A **label** is nothing but an identifier followed by a **colon (:),** and it applied on a block of code or a statement. You can use the label with a **break** and **continue** to control the flow.

You cannot use the line breaks in between the break and continue statement and its label name. Also, there should not be any statement in between the label name and an **associated loop.**

**The syntax for defining the label**

1. labelname:
2. Statement

**labelname:** Any identifier of JavaScript, which is not a reserved word.

**Statement:** It is a JavaScript statement.

Note: In strict mode, you cannot use "let" as the label name. It will cause a syntax error because let is a reserved identifier.

|  |  |
| --- | --- |
| **Label** | **Description** |
| **Label with the break statement** | It is used to exit from the loop or from the switch statement without using a label reference, but with label reference, it is used to jump out from any code block. |
| **Label with continue statement** | It is used to skip one iteration of the loop with or without using the label reference. |

Label with the break statement

Without using the label reference, you can use the break only to exit from the loop or from the switch, but by using the label reference, you can use the break to jump out from any code block.

**Syntax**

1. **break** labelname;

**Example**

1. var x, y;
3. loop1:         //The first for statement is labeled as "loop1."
5. **for** (x = 1; x < 4; x++) {
6. loop2:    //The second for statement is labelled as "loop2"
7. **for** (y = 1; y < 4; y++) {
8. **if** (x === 2 && y === 2) {
9. **break** loop1;
10. }
11. console.log('x = ' + x + ', y = ' + y);
12. }
13. }

**Output**

x = 1, y = 1

x = 1, y = 2

x = 1, y = 3

x = 2, y = 1

Label with continue statement

The continue statement can only be used for skipping one loop iteration either with the label reference or without using the label reference.

**Syntax**

1. **continue** labelname;

**Example**

1. var x, y;
3. loop1:   //The first for statement is labelled as "loop1"
4. **for** (x = 1; x < 4; x++) {
5. loop2:  //The second for statement is labelled as "loop2"
6. **for** (y = 1; y < 4; y++) {
7. **if** (x === 2 && y === 2) {
8. **continue** loop1;
9. }
10. console.log('x = ' + x + ', y = ' + y);
11. }
12. }

You can notice in the following output of the above code that it skips both "x = 2, y = 2" and "x = 2, y = 3".

**Output**

x = 1, y = 1

x = 1, y = 2

x = 1, y = 3

x = 2, y = 1

x = 3, y = 1

x = 3, y = 2

x = 3, y = 3

Labeled function declarations

Before ECMAScript 6, the **LabeledStatement** specification didn't allow for the association of a label statement with a **FunctionDeclaration.** However, the labeled **FunctionDeclaration** was the allowable extension in **non-strict code,** and most of the browser-hosted implementations of ECMAScript supported that extension.

But in ECMAScript 2015 (ES6), the grammar productions for the **LabeledStatement** allow the use of **FunctionDeclaration** as a **LabeledItem,** but it includes an error rule that causes a syntax error if that occurs.

For the web browser compatibility, that rule is modified with the addition of underlined text:

**LabeledItem : FunctionDeclaration**

It will cause a syntax error if any of the source code of strict mode matches this rule.

Starting with ECMAScript 2015, the labelled function declarations standardized for non-strict code:

1. L: function hello() {}

If you write the above code in strict mode then this will throw a syntax error:

1. 'use strict';
2. L: function hello() {}
3. // SyntaxError: In strict mode code, functions can only be declared at top level or inside a block.

The **Generator Functions** can neither be labeled in **non-strict mode** nor in **strict mode.**

1. L: function\* hello()
2. {
3. }
4. // SyntaxError: Generator Functions cannot be labelled