- Full Stack Web Development with Web3

Loops and Iterations

Agenda

1. for loop
2. for...in loop
3. for... of Loop
4. while and do...while Loop
5. break Statement

JavaScript for loop

The syntax of the `for` loop is:

```
for (initialExpression; condition; updateExpression) {
    // for loop body
}
```

Here,

1. The **initialExpression** initializes and/or declares variables and executes only once.
2. The **condition** is evaluated.
   - If the condition is `false`, the `for` loop is terminated.
   - If the condition is `true`, the block of code inside of the `for` loop is executed.
3. The **updateExpression** updates the value of **initialExpression** when the condition is `true`.
4. The **condition** is evaluated again. This process continues until the condition is `false`.

Example 1: Display a Text Five Times

```
// program to display text 5 times
const n = 5;

// looping from i = 1 to 5
for (let i = 1; i <= n; i++) {
```

```
    console.log(`I love JavaScript.`);
}
```

Output

```
I love JavaScript.
I love JavaScript.
I love JavaScript.
I love JavaScript.
I love JavaScript.
```

Here is how this program works.

Iteration	Variable	Condition: i <= n	Action
1st	i = 1 n = 5	☑	I love JavaScript. is printed.i is increased to 2.
2nd	i = 2 n = 5	☑	I love JavaScript. is printed.i is increased to 3.
3rd	i = 3 n = 5	☑	I love JavaScript. is printed.i is increased to 4.
4th	i = 4 n = 5	☑	I love JavaScript. is printed.i is increased to 5.
5th	i = 5 n = 5	☑	I love JavaScript. is printed.i is increased to 6.
6th	i = 6 n = 5	☐	The loop is terminated.

Example 2: Display Numbers from 1 to 5

```
// program to display numbers from 1 to 5
const n = 5;

// looping from i = 1 to 5
// in each iteration, i is increased by 1
for (let i = 1; i <= n; i++) {
    console.log(i);     // printing the value of i
}
```

Output

```
1
2
```

```
3
4
5
```

Here is how this program works.

Iteration	Variable	Condition: i <= n	Action
1st	i = 1 n = 5	☑	1 is printed.i is increased to 2.
2nd	i = 2 n = 5	☑	2 is printed.i is increased to 3.
3rd	i = 3 n = 5	☑	3 is printed.i is increased to 4.
4th	i = 4 n = 5	☑	4 is printed.i is increased to 5.
5th	i = 5 n = 5	☑	5 is printed.i is increased to 6.
6th	i = 6 n = 5	☐	The loop is terminated.

Example 3: Display Sum of n Natural Numbers

```
// program to display the sum of natural numbers
let sum = 0;
const n = 100

// looping from i = 1 to n
// in each iteration, i is increased by 1
for (let i = 1; i <= n; i++) {
    sum += i;  // sum = sum + i
}

console.log('sum:', sum);
```

Output

```
sum: 5050
```

Here, the value of sum is **0** initially. Then, a `for` loop is iterated from `i = 1 to 100`. In each iteration, i is added to sum and its value is increased by **1**.

When i becomes **101**, the test condition is `false` and sum will be equal to 0 + 1 + 2 + ... + 100.

The above program to add sum of natural numbers can also be written as

```
// program to display the sum of n natural numbers
let sum = 0;
const n = 100;

// looping from i = n to 1
// in each iteration, i is decreased by 1
for(let i = n; i >= 1; i-- ) {
    // adding i to sum in each iteration
    sum += i; // sum = sum + i
}

console.log('sum:',sum);
```

This program also gives the same output as the **Example 3**. You can accomplish the same task in many different ways in programming; programming is all about logic.
Although both ways are correct, you should try to make your code more readable.

JavaScript Infinite for loop

If the test condition in a `for` loop is always `true`, it runs forever (until memory is full). For example,

```
// infinite for loop
for(let i = 1; i > 0; i++) {
    // block of code
}
```

In the above program, the condition is always `true` which will then run the code for infinite times.
There are also other types of loops. The `for..in` loop in JavaScript allows you to iterate over all property keys of an object.

JavaScript for...in loop

The syntax of the `for...in` loop is:

```
for (key in object) {
```

```
    // body of for...in
}
```

In each iteration of the loop, a key is assigned to the key variable. The loop continues for all object properties.
Note: Once you get keys, you can easily find their corresponding values.

Example 1: Iterate Through an Object

```
const student = {
    name: 'Monica',
    class: 7,
    age: 12
}

// using for...in
for ( let key in student ) {

    // display the properties
    console.log(`${key} => ${student[key]}`);
}
```

Output

```
name => Monica
class => 7
age => 12
```

In the above program, the `for...in` loop is used to iterate over the `student` object and print all its properties.

- The object key is assigned to the variable .

  key
- `student[key]` is used to access the value of .

  key

Example 2: Update Values of Properties

```
const salaries= {
    Jack : 24000,
    Paul : 34000,
```

```
    Monica : 55000
}

// using for...in
for ( let i in salaries) {

    // add a currency symbol
    let salary = "$" + salaries[i];

    // display the values
    console.log(`${i} : ${salary}`);
}
```

Output

```
Jack : $24000,
Paul : $34000,
Monica : $55000
```

In the above example, the `for...in` loop is used to iterate over the properties of the `salaries` object. Then, the string `$` is added to each value of the object.

---

for...in with Strings

You can also use `for...in` loop to iterate over string values. For example,

```
const string = 'code';

// using for...in loop
for (let i in string) {
    console.log(string[i]);
}
```

Output

```
c
o
d
e
```

---

for...in with Arrays

You can also use `for...in` with arrays. For example,

```
// define array
const arr = [ 'hello', 1, 'JavaScript' ];

// using for...in loop
for (let x in arr) {
    console.log(arr[x]);
}
```

Output

```
hello
1
JavaScript
```

You will learn more about the arrays in later tutorials.
Note: You should not use `for...in` to iterate over an array where the index order is important.
One of the better ways to iterate over an array is using the `for...of` loop.

JavaScript for...of loop

The syntax of the `for...of` loop is:

```
for (element of iterable) {
    // body of for...of
}
```

Here,

- **iterable** - an iterable object (array, set, strings, etc).
- **element** - items in the iterable

In plain English, you can read the above code as: for every element in the iterable, run the body of the loop.

---

for...of with Arrays

The `for..of` loop can be used to iterate over an array. For example,

```
// array
const students = ['John', 'Sara', 'Jack'];

// using for...of
for ( let element of students ) {
```

```
    // display the values
    console.log(element);
}
```

Output

```
John
Sara
Jack
```

In the above program, the `for...of` loop is used to iterate over the students array object and display all its values.

---

for...of with Strings

You can use `for...of` loop to iterate over string values. For example,

```
// string
const string = 'code';

// using for...of loop
for (let i of string) {
    console.log(i);
}
```

Output

```
c
o
d
e
```

---

for...of with Sets

You can iterate through Set elements using the `for...of` loop. For example,

```
// define Set
const set = new Set([1, 2, 3]);

// looping through Set
for (let i of set) {
```

```
    console.log(i);
}
```

Output

```
1
2
3
```

---

for...of with Maps

You can iterate through Map elements using the `for...of` loop. For example,

```
// define Map
let map = new Map();

// inserting elements
map.set('name', 'Jack');
map.set('age', '27');

// looping through Map
for (let [key, value] of map) {
    console.log(key + '- ' + value);
}
```

Output

```
name- Jack
age- 27
```

User Defined Iterators

You can create an iterator manually and use the `for...of` loop to iterate through the iterators. For example,

```
// creating iterable object
const iterableObj = {

    // iterator method
    [Symbol.iterator]() {
        let step = 0;
```

```
        return {
            next() {
                step++;
                if (step === 1) {
                    return { value: '1', done: false};
                 }
                else if (step === 2) {
                    return { value: '2', done: false};
                }
                else if (step === 3) {
                    return { value: '3', done: false};
                }
                return { value: '', done: true };
            }
        }
    }
}

// iterating using for...of
for (const i of iterableObj) {
 console.log(i);
}
```

Output

```
1
2
3
```

---

for...of with Generators

Since generators are iterables, you can implement an iterator in an easier way. Then you can iterate through the generators using the `for...of` loop. For example,

```
// generator function
function* generatorFunc() {

    yield 10;
    yield 20;
    yield 30;
}
```

```
const obj = generatorFunc();

// iteration through generator
for (let value of obj) {
    console.log(value);
}
```

Output

```
10
20
30
```

---

for...of Vs for...in

for...of	for...in
The `for...of` loop is used to iterate through the values of an iterable.	The `for...in` loop is used to iterate through the keys of an object.
The `for...of` loop cannot be used to iterate over an object.	You can use `for...in` to iterate over an iterable such arrays and strings but you should avoid using `for...in` for iterables.

while and do...while Loop

The syntax of the `while` loop is:

```
while (condition) {
    // body of loop
}
```

Here,

1. A `while` loop evaluates the **condition** inside the parenthesis `()`.

2. If the **condition** evaluates to `true`, the code inside the `while` loop is executed.
3. The **condition** is evaluated again.
4. This process continues until the **condition** is `false`.
5. When the **condition** evaluates to `false`, the loop stops.

Example 1: Display Numbers from 1 to 5

```
// program to display numbers from 1 to 5
// initialize the variable
let i = 1, n = 5;

// while loop from i = 1 to 5
while (i <= n) {
    console.log(i);
    i += 1;
}
```

Output

```
1
2
3
4
5
```

Here is how this program works.

Iteration	Variable	Condition: i <= n	Action
1st	i = 1 n = 5	☑	1 is printed. i is increased to 2.
2nd	i = 2 n = 5	☑	2 is printed. i is increased to 3.
3rd	i = 3 n = 5	☑	3 is printed. i is increased to 4.
4th	i = 4 n = 5	☑	4 is printed. i is increased to 5.
5th	i = 5 n = 5	☑	5 is printed. i is increased to 6.
6th	i = 6 n = 5	☐	The loop is terminated

Example 2: Sum of Positive Numbers Only

```
// program to find the sum of positive numbers
// if the user enters a negative numbers, the loop ends
// the negative number entered is not added to sum

let sum = 0;

// take input from the user
let number = parseInt(prompt('Enter a number: '));

while(number >= 0) {

    // add all positive numbers
    sum += number;

    // take input again if the number is positive
    number = parseInt(prompt('Enter a number: '));
}

// display the sum
console.log(`The sum is ${sum}.`);
```

Output

```
Enter a number: 2
Enter a number: 5
Enter a number: 7
Enter a number: 0
Enter a number: -3
The sum is 14.
```

In the above program, the user is prompted to enter a number.

Here, `parseInt()` is used because `prompt()` takes input from the user as a string. And when numeric strings are added, it behaves as a string. For example, `'2' + '3' = '23'`. So `parseInt()` converts a numeric string to number.

The `while` loop continues until the user enters a negative number. During each iteration, the number entered by the user is added to the `sum` variable.

When the user enters a negative number, the loop terminates. Finally, the total sum is displayed.

---

do...while Loop

The syntax of `do...while` loop is:

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

Here,

1. The body of the loop is executed at first. Then the **condition** is evaluated.
2. If the **condition** evaluates to `true`, the body of the loop inside the `do` statement is executed again.
3. The **condition** is evaluated once again.
4. If the **condition** evaluates to `true`, the body of the loop inside the `do` statement is executed again.
5. This process continues until the **condition** evaluates to `false`. Then the loop stops.

Note: `do...while` loop is similar to the `while` loop. The only difference is that in `do...while` loop, the body of loop is executed at least once.

Flowchart of do...while Loop

Let's see the working of `do...while` loop.

Example 3: Display Numbers from 1 to 5

```
// program to display numbers
let i = 1;
const n = 5;

// do...while loop from 1 to 5
do {
    console.log(i);
    i++;
} while(i <= n);
```

Output

```
1
2
3
4
5
```

Here is how this program works.

Iteration	Variable	Condition: i <= n	Action
	`i = 1` `n = 5`	not checked	1 is printed. i is increased to 2.
1st	`i = 2` `n = 5`	`true`	2 is printed. i is increased to 3.
2nd	`i = 3` `n = 5`	`true`	3 is printed. i is increased to 4.
3rd	`i = 4` `n = 5`	`true`	4 is printed. i is increased to 5.
4th	`i = 5` `n = 5`	`true`	5 is printed. i is increased to 6.
5th	`i = 6` `n = 5`	`false`	The loop is terminated

Example 4: Sum of Positive Numbers

```
// to find the sum of positive numbers
// if the user enters negative number, the loop terminates
// negative number is not added to sum

let sum = 0;
let number = 0;

do {
    sum += number;
    number = parseInt(prompt('Enter a number: '));
} while(number >= 0)

console.log(`The sum is ${sum}.`);
```

Output 1

```
Enter a number: 2
Enter a number: 4
Enter a number: -500
The sum is 6.
```

Here, the `do...while` loop continues until the user enters a negative number. When the number is negative, the loop terminates; the negative number is not added to the sum variable.

Output 2

```
Enter a number: -80
The sum is 0.
```

The body of the `do...while` loop runs only once if the user enters a negative number.

Infinite while Loop

If **the condition** of a loop is always `true`, the loop runs for infinite times (until the memory is full). For example,

```
// infinite while loop
while(true){
    // body of loop
}
```

Here is an example of an infinite `do...while` loop.

```
// infinite do...while loop
const count = 1;
do {
   // body of loop
} while(count == 1)
```

In the above programs, the **condition** is always `true`. Hence, the loop body will run for infinite times.

---

for Vs while Loop

A `for` loop is usually used when the number of iterations is known. For example,

```
// this loop is iterated 5 times
for (let i = 1; i <=5; ++i) {
   // body of loop
}
```

And `while` and `do...while` loops are usually used when the number of iterations are unknown. For example,

```
while (condition) {
    // body of loop
}
```

break Statement

The `break` statement is used to terminate the loop immediately when it is encountered.
The syntax of the `break` statement is:

```
break [label];
```

Note: `label` is optional and rarely used.

Working of JavaScript break Statement

Example 1: break with for Loop

```
// program to print the value of i
for (let i = 1; i <= 5; i++) {
    // break condition
    if (i == 3) {
        break;
    }
    console.log(i);
}
```

Output

```
1
2
```

In the above program, the `for` loop is used to print the value of i in each iteration. The `break` statement is used as:

```
if(i == 3) {
    break;
}
```

This means, when i is equal to **3**, the `break` statement terminates the loop. Hence, the output doesn't include values greater than or equal to 3.
Note: The `break` statement is almost always used with decision-making statements.

Example 2: break with while Loop

```
// program to find the sum of positive numbers
// if the user enters a negative numbers, break ends the
loop
// the negative number entered is not added to sum

let sum = 0, number;

while(true) {

    // take input again if the number is positive
    number = parseInt(prompt('Enter a number: '));

    // break condition
    if(number < 0) {
        break;
    }

    // add all positive numbers
    sum += number;

}

// display the sum
console.log(`The sum is ${sum}.`);
```

Output

```
Enter a number: 1
Enter a number: 2
Enter a number: 3
Enter a number: -5
The sum is 6.
```

In the above program, the user enters a number. The `while` loop is used to print the total sum of numbers entered by the user.
Here the `break` statement is used as:

```
if(number < 0) {
    break;
}
```

When the user enters a negative number, here -5, the `break` statement terminates the loop and the control flow of the program goes outside the loop.
Thus, the `while` loop continues until the user enters a negative number.

break with Nested Loop

When `break` is used inside of two nested loops, `break` terminates the inner loop. For example,

```
// nested for loops

// first loop
for (let i = 1; i <= 3; i++) {

    // second loop
    for (let j = 1; j <= 3; j++) {
        if (i == 2) {
          break;
        }
        console.log(`i = ${i}, j = ${j}`);
    }
}
```

Output

```
i = 1, j = 1
i = 1, j = 2
i = 1, j = 3
i = 3, j = 1
i = 3, j = 2
i = 3, j = 3
```

In the above program, when `i == 2`, `break` statement executes. It terminates the inner loop and control flow of the program moves to the outer loop.
Hence, the value of i = 2 is never displayed in the output.

---

continue Statement

The `continue` statement is used to skip the current iteration of the loop and the control flow of the program goes to the next iteration.
The syntax of the `continue` statement is:

```
continue [label];
```

Working of JavaScript continue Statement

continue with for Loop

In a `for` loop, `continue` skips the current iteration and control flow jumps to the **updateExpression**.

Example 1: Print the Value of i

```
// program to print the value of i
for (let i = 1; i <= 5; i++) {

    // condition to continue
    if (i == 3) {
        continue;
    }

    console.log(i);
}
```

Output

```
1
2
4
5
```

In the above program, `for` loop is used to print the value of i in each iteration. Notice the `continue` statement inside the loop.

```
if(i == 3) {
    continue;
}
```

This means

- When `i` is equal to **3**, the `continue` statement skips the third iteration.
- Then, `i` becomes **4** and the test **condition** and `continue` statement is evaluated again.
- Hence, **4** and **5** are printed in the next two iterations.

Note: The `break` statement terminates the loop entirely. However, the `continue` statement only skips the current iteration.

---

continue with while Loop

In a `while` loop, `continue` skips the current iteration and control flow of the program jumps back to the `while` condition.
The `continue` statement works in the same way for `while` and `do...while` loops.

Example 2: Calculate Positive Number

```
// program to calculate positive numbers only
// if the user enters a negative number, that number is
skipped from calculation

// negative number -> loop terminate
// non-numeric character -> skip iteration

let sum = 0;
let number = 0;

while (number >= 0) {

    // add all positive numbers
    sum += number;

    // take input from the user
    number = parseInt(prompt('Enter a number: '));

    // continue condition
    if (isNaN(number)) {
        console.log('You entered a string.');
        number = 0; // the value of number is made 0 again
        continue;
    }

}

// display the sum
```

```
console.log(`The sum is ${sum}.`);
```

Output

```
Enter a number: 1
Enter a number: 2
Enter a number: hello
You entered a string.
Enter a number: 5
Enter a number: -2
The sum is 8.
```

In the above program, the user enters a number. The `while` loop is used to print the total sum of positive numbers entered by the user.

Label

Description

You can use a label to identify a loop, and then use the `break` or `continue` statements to indicate whether a program should interrupt the loop or continue its execution.
Note that JavaScript has ***no*** `goto` statement, you can only use labels with `break` or `continue`.
In strict mode code, you can't use `"let"` as a label name. It will throw a `SyntaxError` (let is a reserved identifier).

Examples

Using a labeled continue with for loops

```
let i, j;

loop1:
for (i = 0; i < 3; i++) {      //The first for statement is
labeled "loop1"
   loop2:
   for (j = 0; j < 3; j++) {   //The second for statement is
labeled "loop2"
      if (i === 1 && j === 1) {
         continue loop1;
      }
      console.log('i = ' + i + ', j = ' + j);
   }
```

```
}

// Output is:
//   "i = 0, j = 0"
//   "i = 0, j = 1"
//   "i = 0, j = 2"
//   "i = 1, j = 0"
//   "i = 2, j = 0"
//   "i = 2, j = 1"
//   "i = 2, j = 2"
// Notice how it skips both "i = 1, j = 1" and "i = 1, j =
2"
```

Using a labeled continue statement

Given an array of items and an array of tests, this example counts the number of items that passes all the tests.

```
let itemsPassed = 0;
let i, j;

top:
for (i = 0; i < items.length; i++) {
  for (j = 0; j < tests.length; j++) {
    if (!tests[j].pass(items[i])) {
      continue top;
    }
  }

  itemsPassed++;
}
```

Using a labeled break with for loops

```
let i, j;

loop1:
for (i = 0; i < 3; i++) {      //The first for statement is
labeled "loop1"
   loop2:
   for (j = 0; j < 3; j++) {   //The second for statement is
labeled "loop2"
      if (i === 1 && j === 1) {
```

```
         break loop1;
      }
      console.log('i = ' + i + ', j = ' + j);
   }
}

// Output is:
//   "i = 0, j = 0"
//   "i = 0, j = 1"
//   "i = 0, j = 2"
//   "i = 1, j = 0"
// Notice the difference with the previous continue example
```

Using a labeled break statement

Given an array of items and an array of tests, this example determines whether all items pass all tests.

```
let allPass = true;
let i, j;

top:
for (i = 0; i < items.length; i++) {
  for (j = 0; j < tests.length; j++) {
    if (!tests[j].pass(items[i])) {
      allPass = false;
      break top;
    }
  }
}
```

Using a labeled block with break

You can use labels within simple blocks, but only `break` statements can make use of non-loop labels.

```
foo: {
  console.log('face');
  break foo;
  console.log('this will not be executed');
}
console.log('swap');
```

```
// this will log:

// "face"
// "swap"
```

Thank You

-
-
-
-