Loops and iteration

by 9 contributors:

« Previous Next »

Loops offer a quick and easy way to do something repeatedly. This chapter of the JavaScript Guide introduces the different iteration statements available to JavaScript.

You can think of a loop as a computerized version of the game where you tell someone to take X steps in one direction then Y steps in another; for example, the idea "Go five steps to the east" could be expressed this way as a loop:

```
var step;
for (step = 0; step < 5; step++) {
  // Runs 5 times, with values of step 0 through 4.
  console.log('Walking east one step');
}
```

There are many different kinds of loops, but they all essentially do the same thing: they repeat an action some number of times (and it's actually possible that number could be zero). The various loop mechanisms offer different ways to determine the start and end points of the loop. There are various situations that are more easily served by one type of loop over the others.

The statements for loops provided in JavaScript are:

- for statement
- do...while statement
- while statement
- label statement
- break statement
- continue statement
- for...in statement
- for...of statement

for statement

A `for` loop repeats until a specified condition evaluates to false. The JavaScript for loop is similar to the Java and C `for` loop. A `for` statement looks as follows:

```
for ([initialExpression]; [condition]; [incrementExpression])
  statement
```

When a `for` loop executes, the following occurs:

1. The initializing expression `initialExpression`, if any, is executed. This expression usually initializes one or more loop counters, but the syntax allows an expression of any degree of complexity. This expression can also declare variables.
2. The `condition` expression is evaluated. If the value of `condition` is true, the loop statements execute. If the value of `condition` is false, the `for` loop terminates. If the `condition` expression is omitted entirely, the condition is assumed to be true.
3. The `statement` executes. To execute multiple statements, use a block statement (`{ ... }`) to group those statements.
4. The update expression `incrementExpression`, if there is one, executes, and control returns to step 2.

Example

The following function contains a `for` statement that counts the number of selected options in a scrolling list (a `<select>` element that allows multiple selections). The `for` statement declares the variable `i` and initializes it to zero. It checks that `i` is less than the number of options in the `<select>` element, performs the succeeding `if` statement, and increments `i` by one after each pass through the loop.

```
<form name="selectForm">
  <p>
    <label for="musicTypes">Choose some music types, then click t
    <select id="musicTypes" name="musicTypes" multiple="multiple"
      <option selected="selected">R&B</option>
      <option>Jazz</option>
      <option>Blues</option>
      <option>New Age</option>
      <option>Classical</option>
      <option>Opera</option>
    </select>
  </p>
  <p><input id="btn" type="button" value="How many are selected?"
</form>

<script>
function howMany(selectObject) {
  var numberSelected = 0;
  for (var i = 0; i < selectObject.options.length; i++) {
    if (selectObject.options[i].selected) {
      numberSelected++;
    }
  }
  return numberSelected;
}

var btn = document.getElementById("btn");
btn.addEventListener("click", function(){
  alert('Number of options selected: ' + howMany(document.selectF
});
</script>
```

do...while statement

The `do...while` statement repeats until a specified condition evaluates to false. A `do...while` statement looks as follows:

```
do
  statement
while (condition);
```

`statement` executes once before the condition is checked. To execute multiple statements, use a block statement (`{ ... }`) to group those statements. If `condition` is true, the statement executes again. At the end of every execution, the condition is checked. When the condition is false, execution stops and control passes to the statement following `do...while`.

Example

In the following example, the `do` loop iterates at least once and reiterates until i is no longer less than 5.

```
1 do {
2   i += 1;
3   console.log(i);
4 } while (i < 5);
```

`while` statement

A `while` statement executes its statements as long as a specified condition evaluates to true. A `while` statement looks as follows:

```
while (condition)
  statement
```

If the condition becomes false, `statement` within the loop stops executing and control passes to the statement following the loop.

The condition test occurs before `statement` in the loop are executed. If the

condition returns true, `statement` is executed and the condition is tested again. If the condition returns false, execution stops and control is passed to the statement following `while`.

To execute multiple statements, use a block statement ({ ... }) to group those statements.

Example 1

The following `while` loop iterates as long as `n` is less than three:

```
n = 0;
x = 0;
while (n < 3) {
  n++;
  x += n;
}
```

With each iteration, the loop increments `n` and adds that value to `x`. Therefore, `x` and `n` take on the following values:

- After the first pass: `n` = 1 and `x` = 1
- After the second pass: `n` = 2 and `x` = 3
- After the third pass: `n` = 3 and `x` = 6

After completing the third pass, the condition `n < 3` is no longer true, so the loop terminates.

Example 2

Avoid infinite loops. Make sure the condition in a loop eventually becomes false; otherwise, the loop will never terminate. The statements in the following `while` loop execute forever because the condition never becomes false:

```
while (true) {
  console.log("Hello, world");
}
```

label statement

A label provides a statement with an identifier that lets you refer to it elsewhere in your program. For example, 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.

The syntax of the label statement looks like the following:

```
label :
   statement
```

The value of *`label`* may be any JavaScript identifier that is not a reserved word. The *`statement`* that you identify with a label may be any statement.

Example

In this example, the label `markLoop` identifies a `while` loop.

```
1 markLoop:
2 while (theMark == true) {
3    doSomething();
4 }
```

break statement

Use the `break` statement to terminate a loop, `switch`, or in conjunction with a label statement.

- When you use `break` without a label, it terminates the innermost enclosing `while`, `do-while`, `for`, or `switch` immediately and transfers control to the following statement.
- When you use `break` with a label, it terminates the specified labeled statement.

The syntax of the `break` statement looks like this:

1. `break;`
2. `break label;`

The first form of the syntax terminates the innermost enclosing loop or `switch`; the second form of the syntax terminates the specified enclosing label statement.

Example 1

The following example iterates through the elements in an array until it finds the index of an element whose value is `theValue`:

```
for (i = 0; i < a.length; i++) {
  if (a[i] == theValue) {
    break;
  }
}
```

Example 2: Breaking to a label

```
var x = 0;
var z = 0
labelCancelLoops: while (true) {
  console.log("Outer loops: " + x);
  x += 1;
  z = 1;
  while (true) {
    console.log("Inner loops: " + z);
    z += 1;
    if (z === 10 && x === 10) {
      break labelCancelLoops;
    } else if (z === 10) {
      break;
    }
  }
}
```

`continue` statement

The `continue` statement can be used to restart a `while`, `do-while`, `for`, or `label` statement.

- When you use `continue` without a label, it terminates the current iteration of the innermost enclosing `while`, `do-while`, or `for` statement and continues execution of the loop with the next iteration. In contrast to the `break` statement, `continue` does not terminate the execution of the loop entirely. In a `while` loop, it jumps back to the condition. In a `for` loop, it jumps to the `increment-expression`.
- When you use `continue` with a label, it applies to the looping statement identified with that label.

The syntax of the `continue` statement looks like the following:

1. `continue;`
2. `continue label;`

Example 1

The following example shows a `while` loop with a `continue` statement that executes when the value of `i` is three. Thus, `n` takes on the values one, three, seven, and twelve.

```
i = 0;
n = 0;
while (i < 5) {
  i++;
  if (i == 3) {
    continue;
  }
  n += i;
}
```

Example 2

A statement labeled `checkiandj` contains a statement labeled `checkj`. If `continue` is encountered, the program terminates the current iteration of `checkj` and begins the next iteration. Each time `continue` is encountered, `checkj` reiterates until its condition returns `false`. When `false` is returned, the remainder of the `checkiandj` statement is completed, and `checkiandj` reiterates until its condition returns `false`. When `false` is returned, the program continues at the statement following `checkiandj`.

If `continue` had a label of `checkiandj`, the program would continue at the top of the `checkiandj` statement.

```
checkiandj:
  while (i < 4) {
    console.log(i);
    i += 1;
    checkj:
      while (j > 4) {
        console.log(j);
        j -= 1;
        if ((j % 2) == 0) {
          continue checkj;
        }
        console.log(j + " is odd.");
      }
      console.log("i = " + i);
      console.log("j = " + j);
  }
```

for...in statement

The `for...in` statement iterates a specified variable over all the enumerable properties of an object. For each distinct property, JavaScript executes the specified statements. A `for...in` statement looks as follows:

```
for (variable in object) {
  statements
}
```

Example

The following function takes as its argument an object and the object's name. It then iterates over all the object's properties and returns a string that lists the property names and their values.

```
function dump_props(obj, obj_name) {
  var result = "";
  for (var i in obj) {
    result += obj_name + "." + i + " = " + obj[i] + "<br>";
  }
  result += "<hr>";
  return result;
}
```

For an object `car` with properties `make` and `model`, `result` would be:

```
car.make = Ford
car.model = Mustang
```

Arrays

Although it may be tempting to use this as a way to iterate over `Array` elements, the **for...in** statement will return the name of your user-defined properties in addition to the numeric indexes. Thus it is better to use a traditional `for` loop with a numeric index when iterating over arrays, because the **for...in** statement iterates over user-defined properties in addition to the array elements, if you modify the Array object, such as adding custom properties or methods.

for...of statement

> **This is a new technology, part of the ECMAScript 2015 (ES6) standard.**
> This technology's specification has been finalized, but check the compatibility table for usage and implementation status in various browsers.

The `for...of` statement creates a loop Iterating over iterable objects (including

`Array`, `Map`, `Set`, arguments object and so on), invoking a custom iteration hook with statements to be executed for the value of each distinct property.

```
for (variable of object) {
  statement
}
```

The following example shows the difference between a `for...of` loop and a `for...in` loop. While `for...in` iterates over property names, `for...of` iterates over property values:

```
let arr = [3, 5, 7];
arr.foo = "hello";

for (let i in arr) {
   console.log(i); // logs "0", "1", "2", "foo"
}

for (let i of arr) {
   console.log(i); // logs "3", "5", "7"
}
```

« Previous Next »