

## Control flow

JavaScript supports a compact set of statements, specifically control flow statements, that you can use to incorporate a great deal of interactivity in your application. Any JavaScript expression is also a statement.

## Expressions and operators

At a high level, an *expression* is a valid unit of code that resolves to a value. There are **two types** of expressions: those that have side effects (such as assigning values) and those that purely *evaluate*. The expression `x = 7` is an example of the **first type**. This expression uses the `=` *operator* to assign the value seven to the variable `x`. The expression itself evaluates to 7.

The expression `3 + 4` is an example of the **second type**. This expression uses the `+` operator to add 3 and 4 together and produces a value, 7. However, if it's not eventually part of a bigger construct, its result will be immediately discarded — this is usually a programmer mistake because the evaluation doesn't produce any effects.

## Block statement

The most basic statement is a block statement, which is used to group statements. The block is delimited by a pair of curly braces:

JS

---

```
{  
  statement1;  
  statement2;  
  // ... statementN;  
}
```

### Example

Block statements are commonly used with control flow statements (if, for, while).

JS

---

```
while (x < 10) {  
    x++;  
}
```

## Conditional statements

A conditional statement is a set of commands that executes if a specified condition is **true**. JavaScript supports two conditional statements: `if...else` and `switch`.

### `if...else` statement

Use the `if` statement to execute a statement if a logical condition is **true**. Use the optional `else` clause to execute a statement if the condition is **false**. An `if` statement looks like this:

JS

---

```
if (condition) {  
    statement1;  
}  
else  
{  
    statement2;  
}
```

Here, the condition can be any expression that evaluates to **true** or **false**. If condition evaluates to **true**, `statement1` is executed. **Otherwise**, `statement2` is executed. `statement1` and `statement2` can be any statement, including further nested `if` statements.

You can also compound the statements using `else if` to have multiple conditions tested in sequence, as follows:

JS

---

```
if (condition1) { statement1;  
} else if (condition2) { statement2;  
} else if (conditionN) { statementN; }  
else { statementLast; }
```

In the case of multiple conditions, only the first logical condition which evaluates to **true** will be executed. To execute multiple statements, group them within a block statement ( `{ /* ... */ }` ).

In general, it's good practice to always use **block statements**—especially when nesting `if` statements:

JS

---

```
if (condition) {  
    // Statements for when condition is true  
    // ...  
} else {  
    // Statements for when condition is false  
    // ...  
}
```

## Example

In the following example, the function `checkData` returns **true** if the number of characters in a `Text` object is three. Otherwise, it displays an **alert** and returns **false**.

JS

---

```
function checkData() {  
    if (document.form1.threeChar.value.length === 3) {  
        return true;  
    }  
    else {  
        alert('Enter exactly three characters. ${document.form1.threeChar.value} is not valid.', );  
        return false;  
    }  
}
```

## switch statement

A switch statement allows a program to evaluate an expression and attempt to match the expression's value to a case label. If a match is found, the program executes the associated statement.

A switch statement looks like this:

JS

---

```
switch (expression) {  
    case label1: statements1;  
    break;  
    case label2: statements2;  
    break;
```

```
// ...
```

```
default:
```

JavaScript evaluates the above switch statement as follows:

- The program first looks for a case clause with a label matching the value of expression and then transfers control to that clause, executing the associated statements.
- If no matching label is found, the program looks for the optional default clause:
- If a default clause is found, the program transfers control to that clause, executing the associated statements.
- If no default clause is found, the program resumes execution at the statement following the end of switch. (The default clause is written as the last clause, but it does not need to be so.)

## break statements

The optional break statement associated with each case clause ensures that the program breaks out of switch once the matched statement is executed, and then continues execution at the statement following switch. If break is omitted, the program continues execution inside the switch statement

## Example

In the following example, if fruitType evaluates to "**Bananas**", the program matches the value with case "**Bananas**" and executes the associated statement. When break is encountered, the program exits the switch and continues execution from the statement following switch. If break were omitted, the statement for case "Cherries" would also be executed.

JS

---

```
switch (fruitType) {  
  case "Oranges":  
    console.log("Oranges are $0.59 a pound.");    break;  
  case "Apples":  
    console.log("Apples are $0.32 a pound.");      break;  
  case "Bananas":  
    console.log("Bananas are $0.48 a pound.");      break;  
  case "Cherries":
```

```

    console.log("Cherries are $3.00 a pound.");    break;
case "Mangoes":
    console.log("Mangoes are $0.56 a pound.");    break;
case "Papayas":
    console.log("Papayas are $2.79 a pound.");    break;
default:
    console.log(`Sorry, we are out of ${fruitType}.`);
}
console.log("Is there anything else you'd like?");

```

## Loops and iteration

Loops offer a quick and easy way to do something repeatedly. 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:

JS

---

```

for (let 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. (Note that it's 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](#)
- [labeled 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:

JS

---

```
for (initialization; condition; afterthought)
statement
```

When a for loop executes, the following occurs:

- ☐ The initializing expression initialization, 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.
- ☐ The condition expression is evaluated. If the value of condition is true, the loop statements execute. Otherwise, the for loop terminates. (If the condition expression is omitted entirely, the condition is assumed to be true.)
- ☐ The statement executes. To execute multiple statements, use a [block statement](#)( { } ) to group those statements.
- ☐ If present, the update expression afterthought is executed.
- ☐ Control returns to Step 2.

## Example

In the example below, the function contains a for statement that counts the number of selected options in a scrolling list (a [<select>](#) element that allows multiple selections).

---

```
<select id="mySelect" multiple>
  <option value="1">Option 1</option>
  <option value="2">Option 2</option>
  <option value="3">Option 3</option>
</select>
```

```
<script>
const selectElement = document.getElementById('mySelect');
const selectedCount = countSelected(selectElement);
console.log(selectedCount); // Will output the number of selected options
</script>
```

The for statement declares the variable `i` and initializes it to 0 . It checks that `i` is less than the number of options in the `<select>` element, performs the succeeding if statement, and increments `i` by 1 after each pass through the loop.

JS

---

```
function countSelected(selectObject) {
let numberSelected = 0;
  for (let i = 0; i < selectObject.options.length; i++)
  {   if (selectObject.options[i].selected)
      {
        numberSelected++;
      }
  }
  return numberSelected;
}
```

This function **countSelected** takes a select element (like a dropdown or multi-select box) as a parameter and counts how many options are selected. Here's how it works:

1. It initializes a counter **numberSelected** to 0
2. It loops through all options in the select element using a for loop from 0 to `selectObject.options.length`
3. For each option, it checks if it's selected using the `selected` property
4. If an option is selected, it increments the counter using `numberSelected++`
5. Finally, it returns the total count of selected options

## do...while statement

The [do...while](#) statement repeats until a specified condition evaluates to **false**. A do...while statement looks as follows:

JS

---

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

Statement is always executed 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** .

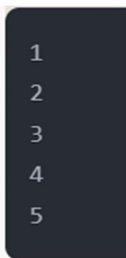
## Example

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

JS

---

```
let i = 0;  
do { i += 1;  
  console.log(i);  
}  
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:

JS

---

```
while (condition) statement(s)
```

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 is 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 3:

```
JS
let n = 0;
let 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:

```
JS
// Infinite loops are bad!
while (true) {
  console.log("Hello, world!");
}
```

### labeled 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 labeled statement looks like the following:

JS

---

**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. For examples of using labeled statements, see the examples of **break** and **continue** below.

### break statement

Use the [break](#) statement to terminate a loop, switch , or in conjunction with a labeled statement.

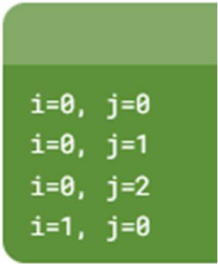
- When you use break without a label, it terminates the innermost enclosing **while**, **do-while**, **for**, or **switch** immediately and transfers control to the following statement.
- When you use break with a label, it terminates the specified labeled statement.

The syntax of the break statement looks like this:

JS

---

```
outerLoop: for (let i = 0; i < 3; i++) {  
  innerLoop: for (let j = 0; j < 3; j++) {  
    if (i === 1 && j === 1) {  
      break outerLoop; // Breaks the outerLoop completely  
    }  
    console.log(`i=${i}, j=${j}`);  
  }  
}
```



```
i=0, j=0  
i=0, j=1  
i=0, j=2  
i=1, j=0
```

When i=1 and j=1 are reached, break outerLoop is executed, and both innerLoop and outerLoop stop.

### Example 1

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

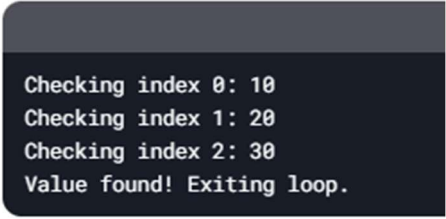
JS

---

```
const a = [10, 20, 30, 40, 50];
const theValue = 30;

for (let i = 0; i < a.length; i++) {
  console.log(`Checking index ${i}: ${a[i]}`);
  if (a[i] === theValue) {
    console.log("Value found! Exiting loop.");
    break;
  }
}
```

**Output:**



```
Checking index 0: 10
Checking index 1: 20
Checking index 2: 30
Value found! Exiting loop.
```

- It loops through an array a (from the first element to the last).
- If it finds an element in a that matches theValue, it stops the loop immediately using break.
- If no element matches theValue, the loop completes all iterations.

## Example 2: Breaking to a label

JS

---

```
let x = 0; let 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:

JS

---

```
continue;  
continue label;
```

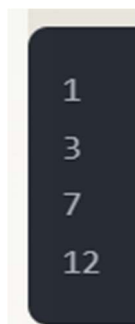
### Example 1

The following example shows a while loop with a continue statement that executes when the value of i is 3. Thus, n takes on the values 1, 3, 7, and 12.

JS

---

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



```
1  
3  
7  
12
```

If you comment out the continue the loop would run till the end and you would see 1,3,6,10,15 .

### Example 2

The code has two nested while loops with labels checkIandJ and checkJ, and uses a continue statement with a label.

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.

JS

```
let i = 0; let j = 10;
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);
}
```

1. First iteration of outer loop ( $i = 0$ ):

- Prints  $i$  (0)
- Inner loop runs:
  - Prints  $j$  values (10,9,8,7,6,5)
  - For even numbers (10,8,6), it continues without printing "is odd"
  - For odd numbers (9,7,5), it prints "is odd"
  - Loop stops when  $j$  becomes 4
- Prints final values ( $i = 1, j = 4$ )

2. Remaining iterations ( $i = 1,2,3$ ):

- Only prints  $i$  and the final values
- Inner loop doesn't run because  $j$  is already 4

The `continue` statement with `checkJ` label causes the loop to skip to the next iteration of the inner loop whenever  $j$  is even, which is why we only see "is odd" printed for odd numbers.

```
0
10
9 is odd.
8
7 is odd.
6
5 is odd.
i = 1
j = 4
1
i = 2
j = 4
2
i = 3
j = 4
3
i = 4
j = 4
```