

# Control Flow Statements

JavaScript is a programming language that allows developers to build complex web applications. Control flow statements are one of the most important features in JavaScript.

Control flow statements are used to control the flow of execution in a program. They are used to make decisions, execute loops, and handle errors. There are three types of control flow statements in JavaScript: conditional statements, loops, and try/catch statements.

# Conditional Statements

- if
- if-else
- nested-if
- if-else-if ladder

## if-statement

It is a conditional statement used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

### **Syntax:**

```
if(condition)  
  
{  
  
// Statements to execute if
```

```
// condition is true
```

```
}
```

The if statement accepts boolean values – if the value is true then it will execute the block of statements under it. If we do not provide the curly braces ‘{’ and ‘}’ after **if( condition )** then by default if statement considers the immediate one statement to be inside its block.

For example,

```
if(condition)
```

```
statement1;
```

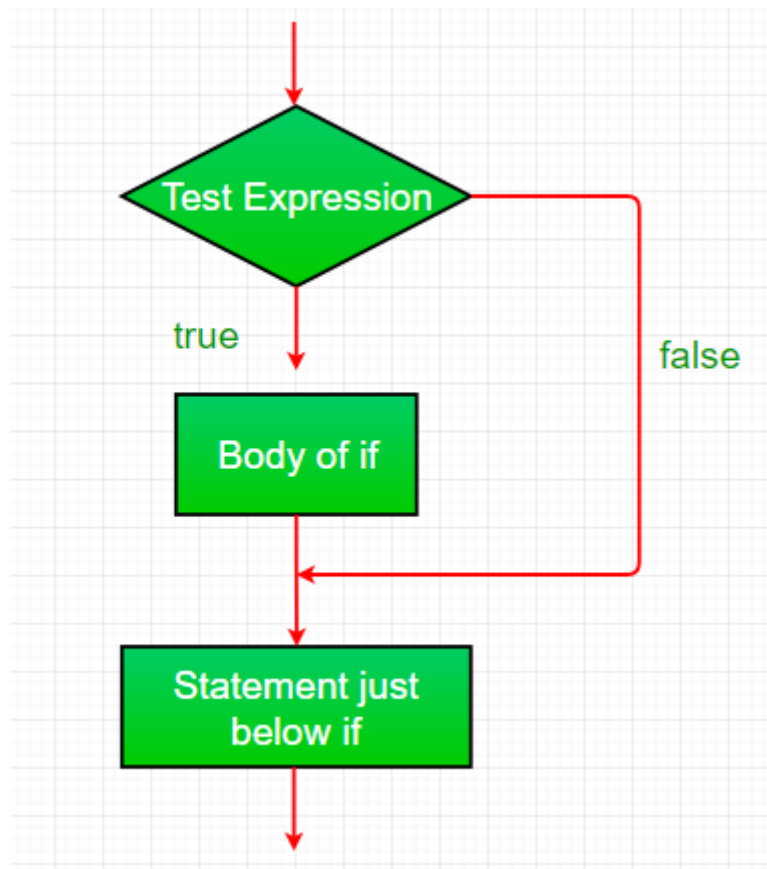
```
statement2;
```

```
// Here if the condition is true, if block
```

```
// will consider only statement1 to be inside
```

```
// its block.
```

## Flow chart:



## Example:

```
// JavaScript program to illustrate If statement  
  
let age = 19;  
  
if (age > 18)  
  
    console.log("Congratulations, You are eligible to  
drive");
```

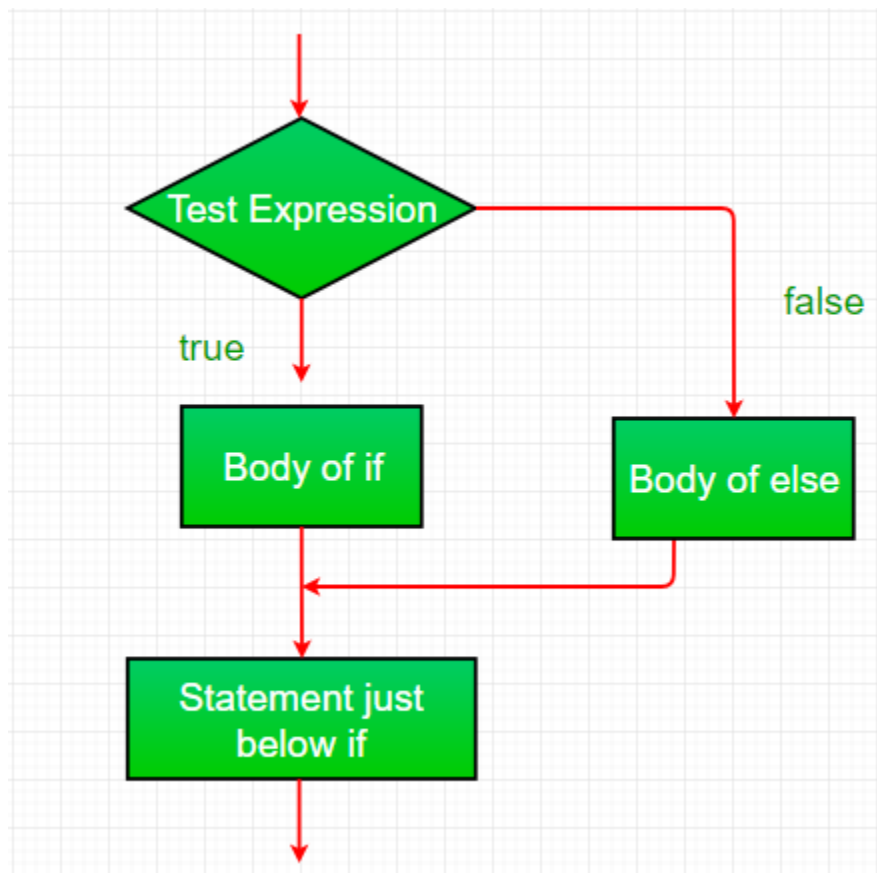
## if-else statement

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false? Here comes the else statement. We can use the else statement with the if statement to execute a block of code when the condition is false.

### **Syntax:**

```
if (condition)  
{  
// Executes this block if  
// condition is true  
}  
else  
{  
// Executes this block if  
// condition is false  
}
```

## Flow chart:



*if-else statement*

## Example

```
// JavaScript program to illustrate If-else statement  
  
let i = 10;  
  
if (i < 15)  
  
  console.log("i is less than 15");  
  
else
```

```
console.log("I am Not in if");
```

**Output:**

**i is less than 15**

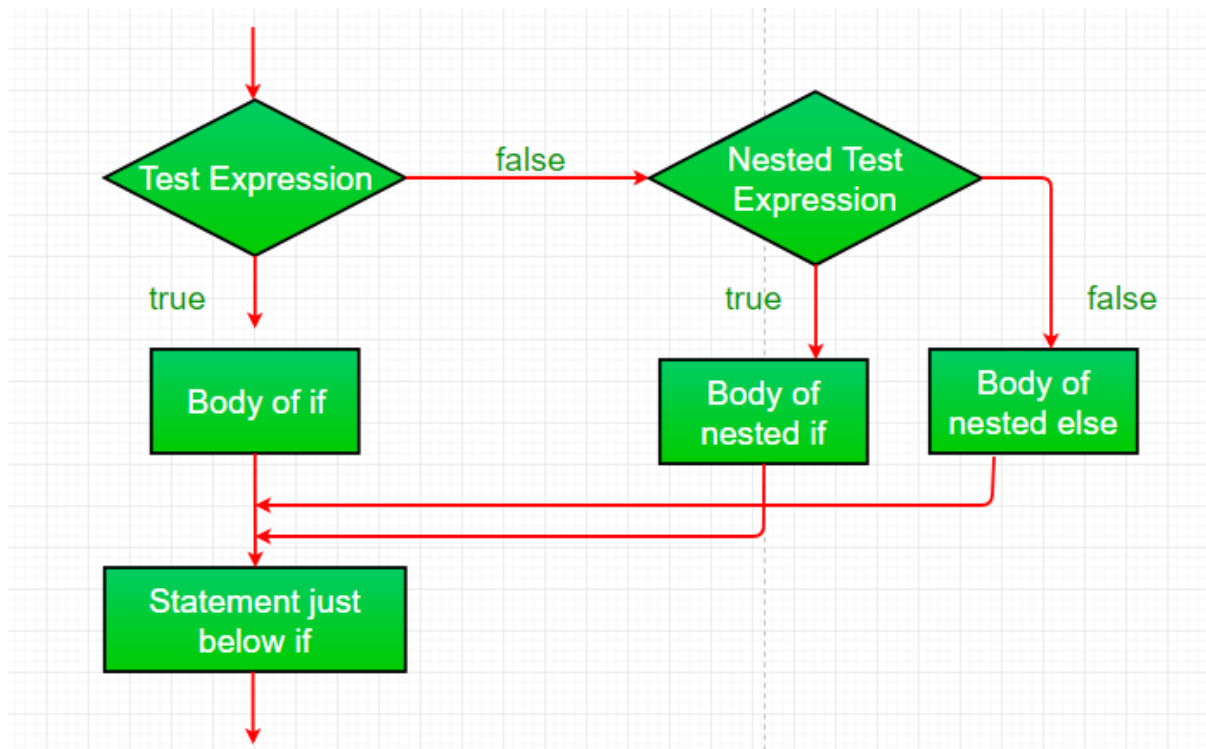
## nested-if statement

JavaScript allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement. A nested if is an if statement that is the target of another if or else.

**Syntax:**

```
if (condition1)
{
    // Executes when condition1 is true
    if (condition2)
    {
        // Executes when condition2 is true
    }
}
```

## Flow chart:



*nested-if statement*

## Example

```
// JavaScript program to illustrate nested-if statement  
let i = 10;  
  
if (i == 10) { // First if statement  
  if (i < 15) {  
    console.log("i is smaller than 15");  
  }  
}
```



```
// Nested - if statement  
// Will only be executed if statement above  
// it is true  
if (i < 12)  
    console.log("i is smaller than 12 too");  
else  
    console.log("i is greater than 15");  
}  
}
```

**Output:**

**i is smaller than 15**

**i is smaller than 12 too**

## if-else-if ladder statement

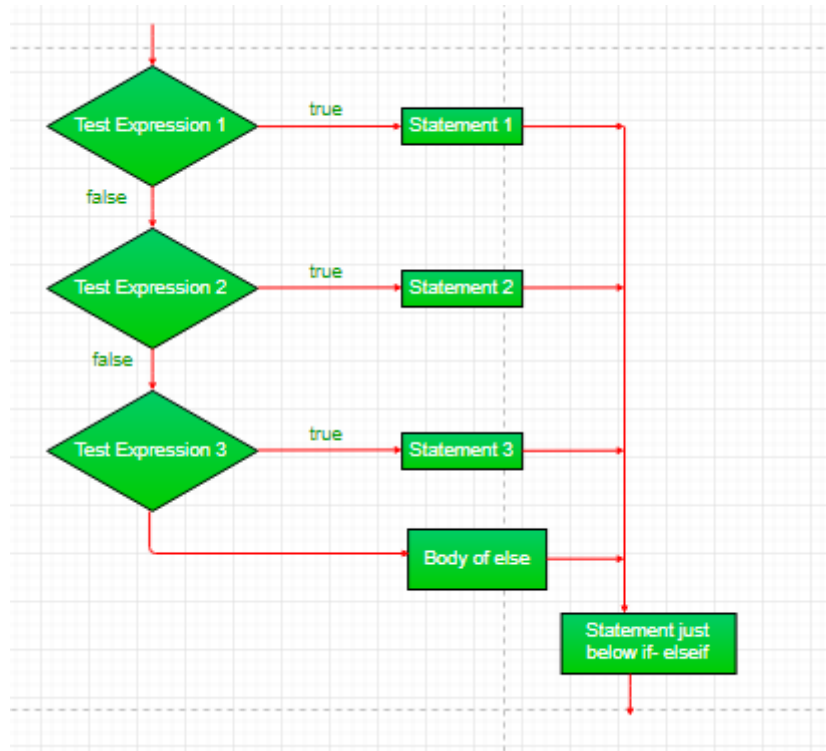
Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement

associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

**Syntax:**

```
if (condition)
    statement;
else if (condition)
    statement;
.
.
else
    statement;
```

## Flow Chart :



*if-else-if ladder statement*

## Example

```
// JavaScript program to illustrate nested-if statement  
let i = 20;  
  
if (i == 10)  
  console.log("i is 10");  
else if (i == 15)
```

```
console.log("i is 15");  
else if (i == 20)  
console.log("i is 20");  
else  
console.log("i is not present");
```

**Output:**

**i is 20**

## switch Statement

The JavaScript switch statement is used in decision making.

The switch statement evaluates an expression and executes the corresponding body that matches the expression's result.

**The syntax of the switch statement is:**

```
switch(variable/expression) {  
case value1:  
// body of case 1  
break;
```

```
case value2:  
    // body of case 2  
    break;  
case valueN:  
    // body of case N  
    break;  
default:  
    // body of default  
}
```

The switch statement evaluates a variable/expression inside parentheses ().

If the result of the expression is equal to value1, its body is executed.

If the result of the expression is equal to value2, its body is executed.

This process goes on. If there is no matching case, the default body executes.

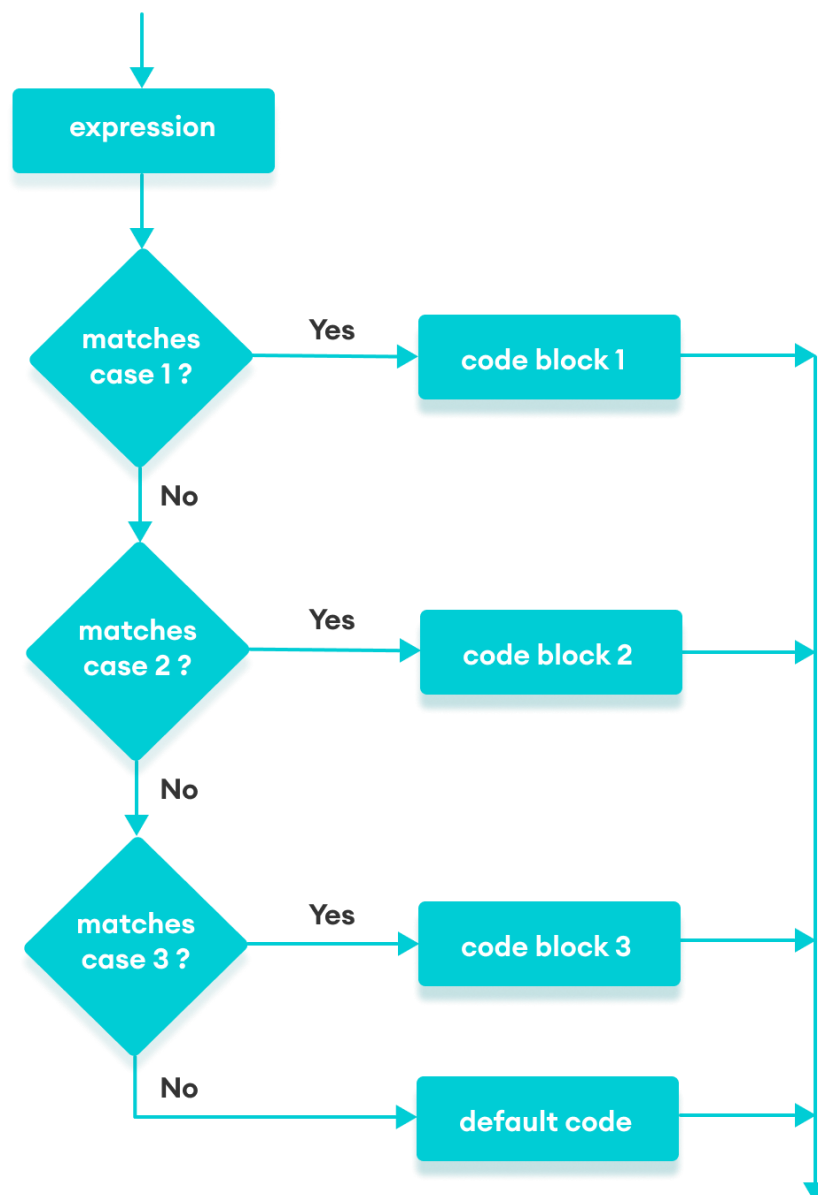
Notes:

The break statement is optional. If the break statement is encountered, the switch statement ends.

If the break statement is not used, the cases after the matching case are also executed.

The default clause is also optional.

### Flowchart of switch Statement



### Flowchart of JavaScript switch statement

---

## Example 1: Simple Program Using switch Statement

```
// program using switch statement
```

```
let a = 2;
```

```
switch (a) {
```

```
  case 1:
```

```
    a = 'one';
```

```
    break;
```

```
  case 2:
```

```
    a = 'two';
```

```
    break;
```

```
  default:
```

```
    a = 'not found';
```

```
    break;
```

```
}
```

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

## Output

**The value is two.**

In the above program, an expression `a == 2` is evaluated with a switch statement.

The **expression's** result is evaluated with case `1` which results in false.

Then the switch statement goes to the second case. Here, the **expression's** result matches with case `2`. So The `value` is two is displayed.

The break statement terminates the block and control flow of the program jumps to outside of the switch block.

### Example 2: Type Checking in switch Statement

```
// program using switch statement
```

```
let a = 1;
```

```
switch (a) {
```

```
  case "1":
```

```
    a = 1;
```

```
    break;
```

```
  case 1:
```

```
    a = 'one';
```

```
    break;
```

```
  case 2:
```



```
a = 'two';  
break;  
default:  
a = 'not found';  
break;  
}  
console.log(`The value is ${a}`);
```

## Output

**The value is one.**

In the above program, an expression `a = 1` is evaluated with a switch statement.

In JavaScript, the switch statement checks the value strictly. So the expression's result does not match with `case "1"`.

Then the switch statement goes to the second case. Here, the expression's result matches with `case 1`. So `The value is one` is displayed.

The `break` statement terminates the block and control flow of the program jumps to outside of the `switch` block.

**Note:** In JavaScript, the switch statement checks the cases strictly (should be of the same data type) with the expression's result. Notice in the above example, **1** does not match with **"1"**.

Let's write a program to make a simple calculator with the `switch` statement.

### Example 3: Simple Calculator

```
// program for a simple calculator
```

```
let result;
```

```
// take the operator input
```

```
const operator = prompt('Enter operator ( either +, -, * or / ) : ');
```

```
// take the operand input
```

```
const number1 = parseFloat(prompt('Enter first number: '));
```

```
const number2 = parseFloat(prompt('Enter second number: '));
```

```
switch(operator) {
```

```
case '+':
```

```
result = number1 + number2;
```

```
console.log(`${number1} + ${number2} = ${result}`);  
break;  
case '-':  
    result = number1 - number2;  
    console.log(`${number1} - ${number2} = ${result}`);  
    break;  
case '*':  
    result = number1 * number2;  
    console.log(`${number1} * ${number2} = ${result}`);  
    break;  
case '/':  
    result = number1 / number2;  
    console.log(`${number1} / ${number2} = ${result}`);  
    break;  
  
default:  
    console.log('Invalid operator');  
    break;  
}
```

## Output

Enter operator: +

**Enter first number: 4**

**Enter second number: 5**

**4 + 5 = 9**

In above program, the user is asked to enter either +, -, \* or /, and two operands. Then, the `switch` statement executes cases based on the user input.

## switch With Multiple Case

In a JavaScript switch statement, cases can be grouped to share the same code.

### **Example 4: switch With Multiple Case**

```
// multiple case switch program
let fruit = 'apple';
switch(fruit) {
case 'apple':
case 'mango':
case 'pineapple':
console.log(`${fruit} is a fruit.`);
break;
```

```
default:  
console.log(`${fruit} is not a fruit.`);  
break;  
}
```

## Output

apple is a fruit.

In the above program, multiple cases are grouped. All the grouped cases share the same code.

If the value of the fruit variable had value mango or pineapple, the output would have been the same.

## 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

```
for (init; condition; update) {  
    // code  
    if (condition to break) {  
        break;  
    }  
    // code  
}
```



---

```
while (condition) {  
    // code  
    if (condition to break) {  
        break;  
    }  
    // code  
}
```



```
// 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

```
// 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.

## Labeled break

When using nested loops, you can also terminate the outer loop with a `label` statement.

However labeled `break` is rarely used in JavaScript because this makes the code harder to read and understand.

## continue

The **`continue`** statement terminates execution of the statements in the current iteration of the current or labeled loop, and continues execution of the loop with the next iteration.

### **Syntax**

**`continue;`**

**continue label;**

**label Optional**

Identifier associated with the label of the statement.

## **Description**

In contrast to the break statement, continue does not terminate the execution of the loop entirely, but instead:

In a while or do...while loop, it jumps back to the condition.

In a for loop, it jumps to the update expression.

In a for...in, for...of, or for await...of loop, it jumps to the next iteration.

The continue statement can include an optional label that allows the program to jump to the next iteration of a labeled loop statement instead of the innermost loop. In this case, the continue statement needs to be nested within this labeled statement.

A continue statement, with or without a following label, cannot be used at the top level of a script, module, function's body, or static initialization block, even when the function or class is further contained within a loop.

## Examples

### Using continue with while

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

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

### Using continue with a label

In the following example, a statement labeled checkIAndJ contains a statement labeled checkJ. If continue is encountered, the program continues at the top

of the checkJ statement. Each time continue is encountered, checkJ reiterates until its condition returns false. When false is returned, the remainder of the checkIAndJ statement is completed.

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

```
let i = 0;
let j = 8;
checkIAndJ: while (i < 4) {
  console.log(`i: ${i}`);
  i += 1;
  checkJ: while (j > 4) {
    console.log(`j: ${j}`);
    j -= 1;
    if (j % 2 === 0) continue checkJ;
    console.log(`${j} is odd.`);
  }
  console.log(`i = ${i}`);
  console.log(`j = ${j}`);
}
```

## Unsyntactic continue statements

continue cannot be used within loops across function boundaries.

```
for (let i = 0; i < 10; i++) {
```

```
  (() => {
```

```
    continue; // SyntaxError: Illegal continue statement: no  
surrounding iteration statement
```

```
  })();
```

```
}
```

When referencing a label, the labeled statement must contain the continue statement.

```
label: for (let i = 0; i < 10; i++) {
```

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

```
}
```

```
for (let i = 0; i < 10; i++) {
```

```
  continue label; // SyntaxError: Undefined label 'label'
```

```
}
```

The labeled statement must be a loop.

```
label: {  
  for (let i = 0; i < 10; i++) {  
    continue label; // SyntaxError: Illegal continue statement:  
    'label' does not denote an iteration statement  
  }  
}
```

## label

A **labeled statement** is any statement that is prefixed with an identifier. You can jump to this label using a break or continue statement nested within the labeled statement.

### Syntax

**JS**Copy to Clipboard

**label:**

**statement**

**label**

**Any JavaScript identifier that is not a reserved word.**



## Examples

### Using a labeled continue with for loops

```
// The first for statement is labeled "loop1"
```

```
loop1: for (let i = 0; i < 3; i++) {
```

```
// The second for statement is labeled "loop2"
```

```
loop2: for (let j = 0; j < 3; j++) {
```

```
if (i === 1 && j === 1) {
```

```
  continue loop1;
```

```
}
```

```
  console.log(`i = ${i}, j = ${j}`);
```

```
}
```

```
}
```

```
// Logs:
```

```
// 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 break with for loops

```
let i, j;
```

```
// The first for statement is labeled "loop1"
```

```
loop1: for (i = 0; i < 3; i++) {
```

```
// The second for statement is labeled "loop2"
```

```
loop2: for (j = 0; j < 3; j++) {
```

```
if (i === 1 && j === 1) {
```

```
break loop1;
```

```
}
```

```
console.log(`i = ${i}, j = ${j}`);
```

```
}
```

```
}
```

```
// Logs:
```

```
// i = 0, j = 0
```

```
// i = 0, j = 1
```

**// i = 0, j = 2**

**// i = 1, j = 0**

# Loops

Loops are used to execute the same block of code again and again, as long as a certain condition is met. The basic idea behind a loop is to automate the repetitive tasks within a program to save the time and effort.

**JavaScript now supports five different types of loops:**

- **while** — loops through a block of code as long as the condition specified evaluates to true.
- **do...while** — loops through a block of code once; then the condition is evaluated. If the condition is true, the statement is repeated as long as the specified condition is true.
- **for** — loops through a block of code until the counter reaches a specified number.
- **for...in** — loops through the properties of an object.
- **for...of** — loops over iterable objects such as arrays, strings, etc.

## for loop

In programming, loops are used to repeat a block of code.

For example, if you want to show a message 100 times, then you can use a loop. It's just a simple example; you can achieve much more with loops.

The syntax of the `for` loop is:

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

Here,

The **initialExpression** initializes and/or declares variables and executes only once.

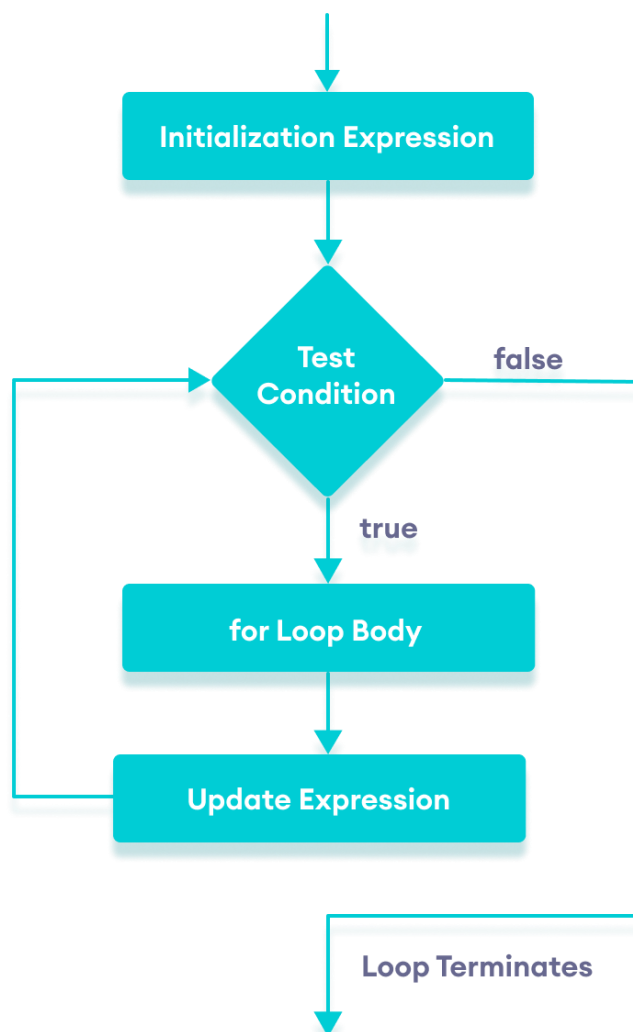
The **condition** is evaluated.

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.

The **updateExpression** updates the value of **initialExpression** when the condition is **true**.

The **condition** is evaluated again. This process continues until the condition is false.



Flowchart of JavaScript for loop

## 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.**

## 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
```

```
}
```

Run Code

**Output**

1

2

3

4

5

### 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.

## JavaScript while and do...while Loop

In programming, loops are used to repeat a block of code. For example, if you want to show a message 100 times, then you can use a loop. It's just a simple example; you can achieve much more with loops.

## JavaScript while Loop

The syntax of the `while` loop is:

```
while (condition) {
```

## // body of loop

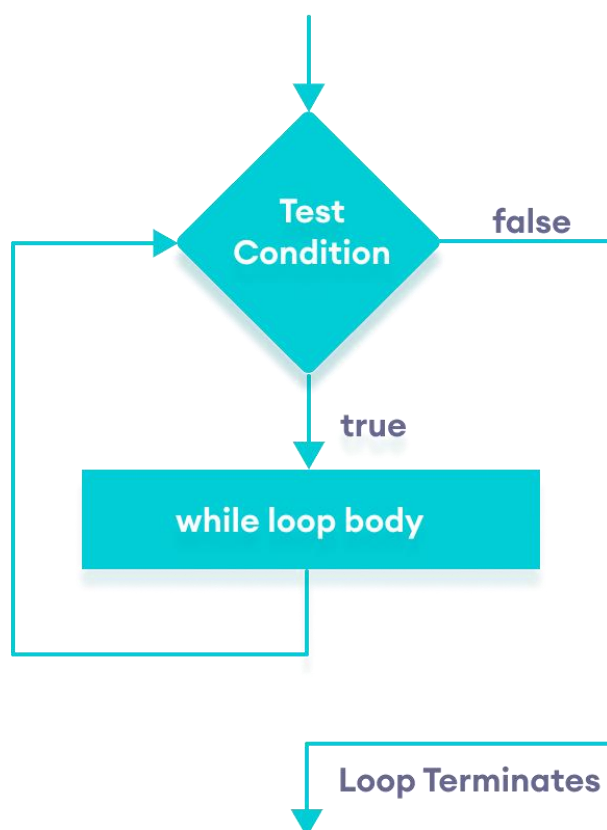
A `while` loop evaluates the **condition** inside the parenthesis `()`.

If the **condition** evaluates to `true`, the code inside the `while` loop is executed.

The **condition** is evaluated again.

This process continues until the **condition** is `false`.

When the **condition** evaluates to `false`, the loop stops.



---

**Flowchart of JavaScript while loop**

```
// 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**

## 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)
```

The body of the loop is executed at first. Then the **condition** is evaluated.

If the **condition** evaluates to `true`, the body of the loop inside the `do` statement is executed again.

The **condition** is evaluated once again.

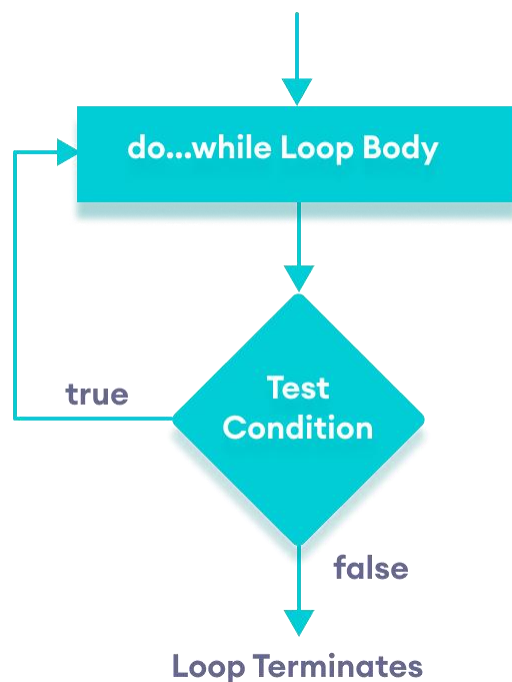
If the **condition** evaluates to `true`, the body of the loop inside the `do` statement is executed again.



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



### 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);
```

Run Code

**Output**

1

2

3

4

5

### 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
```

```
// 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.

## 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]}`);
```

```
}
```

Run Code

## 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`.

```
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

```
// define array
```

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

```
// using for...in loop
```

```
for (let x in arr) {
```

```
  console.log(arr[x]);
```

```
}
```

## Output

hello

1

JavaScript

**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

In JavaScript, there are three ways we can use a `for` loop.

- **JavaScript for loop**
- **JavaScript for...in loop**
- **JavaScript for...of loop**

The `for...of` loop was introduced in the later versions of **JavaScript ES6**.

The `for..of` loop in JavaScript allows you to iterate over iterable objects (arrays, sets, maps, strings etc).

## 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**

## for...of Vs for...in

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