**Control flow and error handling**

* [« Previous](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types)
* [Next »](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration)

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. This chapter provides an overview of these statements.

The [JavaScript reference](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements) contains exhaustive details about the statements in this chapter. The semicolon (;) character is used to separate statements in JavaScript code.

Any JavaScript expression is also a statement. See [Expressions and operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators) for complete information about expressions.

[**Block statement**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#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 brackets:

{

statement1;

statement2;

// …

statementN;

}

Copy to Clipboard

[**Example**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#example)

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

while (x < 10) {

x++;

}

Copy to Clipboard

Here, { x++; } is the block statement.

**Note:** [var](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var)-declared variables are not block-scoped, but are scoped to the containing function or script, and the effects of setting them persist beyond the block itself. For example:

var x = 1;

{

var x = 2;

}

console.log(x); // 2

Copy to Clipboard

This outputs 2 because the var x statement within the block is in the same scope as the var x statement before the block. (In C or Java, the equivalent code would have output 1.)

This scoping effect can be mitigated by using [let](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let) or [const](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const).

[**Conditional statements**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#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**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#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:

if (condition) {

statement1;

} else {

statement2;

}

Copy to Clipboard

Here, the condition can be any expression that evaluates to true or false. (See [Boolean](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean#description) for an explanation of what evaluates to true and false.)

If condition evaluates to true, statement\_1 is executed. Otherwise, statement\_2 is executed. statement\_1 and statement\_2 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:

if (condition1) {

statement1;

} else if (condition2) {

statement2;

} else if (conditionN) {

statementN;

} else {

statementLast;

}

Copy to Clipboard

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 ({ /\* … \*/ }).

**Best practice**

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

if (condition) {

// Statements for when condition is true

// …

} else {

// Statements for when condition is false

// …

}

Copy to Clipboard

In general it's good practice to not have an if...else with an assignment like x = y as a condition:

if (x = y) {

/\* statements here \*/

}

However, in the rare case you find yourself wanting to do something like that, the [while](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/while) documentation has a [Using an assignment as a condition](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/while#using_an_assignment_as_a_condition) section with guidance on a general best-practice syntax you should know about and follow.

**Falsy values**

The following values evaluate to false (also known as [Falsy](https://developer.mozilla.org/en-US/docs/Glossary/Falsy) values):

* false
* undefined
* null
* 0
* NaN
* the empty string ("")

All other values—including all objects—evaluate to true when passed to a conditional statement.

**Note:** Do not confuse the primitive boolean values true and false with the true and false values of the [Boolean](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean) object!

For example:

const b = new Boolean(false);

if (b) {

// this condition evaluates to true

}

if (b == true) {

// this condition evaluates to false

}

Copy to Clipboard

**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.

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;

}

}

Copy to Clipboard

[**switch statement**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#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:

switch (expression) {

case label1:

statements1;

break;

case label2:

statements2;

break;

// …

default:

statementsDefault;

}

Copy to Clipboard

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.
  + (By convention, 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 (and will evaluate the next case, and so on).

**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.

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('Mangoes and 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?");

Copy to Clipboard

[**Exception handling statements**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#exception_handling_statements)

You can throw exceptions using the throw statement and handle them using the try...catch statements.

* [throw statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#throw_statement)
* [try...catch statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#try...catch_statement)

[**Exception types**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#exception_types)

Just about any object can be thrown in JavaScript. Nevertheless, not all thrown objects are created equal. While it is common to throw numbers or strings as errors, it is frequently more effective to use one of the exception types specifically created for this purpose:

* [ECMAScript exceptions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error#error_types)
* [DOMException](https://developer.mozilla.org/en-US/docs/Web/API/DOMException) and [DOMError](https://developer.mozilla.org/en-US/docs/Web/API/DOMError)

[**throw statement**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#throw_statement)

Use the throw statement to throw an exception. A throw statement specifies the value to be thrown:

throw expression;

Copy to Clipboard

You may throw any expression, not just expressions of a specific type. The following code throws several exceptions of varying types:

throw 'Error2'; // String type

throw 42; // Number type

throw true; // Boolean type

throw {toString() { return "I'm an object!"; } };

Copy to Clipboard

[**try...catch statement**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#try...catch_statement)

The try...catch statement marks a block of statements to try, and specifies one or more responses should an exception be thrown. If an exception is thrown, the try...catch statement catches it.

The try...catch statement consists of a try block, which contains one or more statements, and a catch block, containing statements that specify what to do if an exception is thrown in the try block.

In other words, you want the try block to succeed—but if it does not, you want control to pass to the catch block. If any statement within the try block (or in a function called from within the try block) throws an exception, control *immediately* shifts to the catch block. If no exception is thrown in the try block, the catch block is skipped. The finally block executes after the try and catch blocks execute but before the statements following the try...catch statement.

The following example uses a try...catch statement. The example calls a function that retrieves a month name from an array based on the value passed to the function. If the value does not correspond to a month number (1 – 12), an exception is thrown with the value 'InvalidMonthNo' and the statements in the catch block set the monthName variable to 'unknown'.

function getMonthName(mo) {

mo--; // Adjust month number for array index (so that 0 = Jan, 11 = Dec)

const months = [

'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',

'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec',

];

if (months[mo]) {

return months[mo];

} else {

throw new Error('InvalidMonthNo'); // throw keyword is used here

}

}

try { // statements to try

monthName = getMonthName(myMonth); // function could throw exception

} catch (e) {

monthName = 'unknown';

logMyErrors(e); // pass exception object to error handler (i.e. your own function)

}

Copy to Clipboard

**The catch block**

You can use a catch block to handle all exceptions that may be generated in the try block.

catch (catchID) {

statements

}

Copy to Clipboard

The catch block specifies an identifier (catchID in the preceding syntax) that holds the value specified by the throw statement. You can use this identifier to get information about the exception that was thrown.

JavaScript creates this identifier when the catch block is entered. The identifier lasts only for the duration of the catch block. Once the catch block finishes executing, the identifier no longer exists.

For example, the following code throws an exception. When the exception occurs, control transfers to the catch block.

try {

throw 'myException'; // generates an exception

} catch (err) {

// statements to handle any exceptions

logMyErrors(err); // pass exception object to error handler

}

Copy to Clipboard

**Note:** When logging errors to the console inside a catch block, using console.error() rather than console.log() is advised for debugging. It formats the message as an error, and adds it to the list of error messages generated by the page.

**The finally block**

The finally block contains statements to be executed *after* the try and catch blocks execute. Additionally, the finally block executes *before* the code that follows the try…catch…finally statement.

It is also important to note that the finally block will execute *whether or not* an exception is thrown. If an exception is thrown, however, the statements in the finally block execute even if no catch block handles the exception that was thrown.

You can use the finally block to make your script fail gracefully when an exception occurs. For example, you may need to release a resource that your script has tied up.

The following example opens a file and then executes statements that use the file. (Server-side JavaScript allows you to access files.) If an exception is thrown while the file is open, the finally block closes the file before the script fails. Using finally here *ensures* that the file is never left open, even if an error occurs.

openMyFile();

try {

writeMyFile(theData); // This may throw an error

} catch (e) {

handleError(e); // If an error occurred, handle it

} finally {

closeMyFile(); // Always close the resource

}

Copy to Clipboard

If the finally block returns a value, this value becomes the return value of the entire try…catch…finally production, regardless of any return statements in the try and catch blocks:

function f() {

try {

console.log(0);

throw 'bogus';

} catch (e) {

console.log(1);

return true; // this return statement is suspended

// until finally block has completed

console.log(2); // not reachable

} finally {

console.log(3);

return false; // overwrites the previous "return"

console.log(4); // not reachable

}

// "return false" is executed now

console.log(5); // not reachable

}

console.log(f()); // 0, 1, 3, false

Copy to Clipboard

Overwriting of return values by the finally block also applies to exceptions thrown or re-thrown inside of the catch block:

function f() {

try {

throw 'bogus';

} catch (e) {

console.log('caught inner "bogus"');

throw e; // this throw statement is suspended until

// finally block has completed

} finally {

return false; // overwrites the previous "throw"

}

// "return false" is executed now

}

try {

console.log(f());

} catch (e) {

// this is never reached!

// while f() executes, the `finally` block returns false,

// which overwrites the `throw` inside the above `catch`

console.log('caught outer "bogus"');

}

// Logs:

// caught inner "bogus"

// false

Copy to Clipboard

**Nesting try...catch statements**

You can nest one or more try...catch statements.

If an inner try block does *not* have a corresponding catch block:

1. it *must* contain a finally block, and
2. the enclosing try...catch statement's catch block is checked for a match.

For more information, see [nested try-blocks](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch#nested_try-blocks) on the [try...catch](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch) reference page.

[**Utilizing Error objects**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#utilizing_error_objects)

Depending on the type of error, you may be able to use the name and message properties to get a more refined message.

The name property provides the general class of Error (such as DOMException or Error), while message generally provides a more succinct message than one would get by converting the error object to a string.

If you are throwing your own exceptions, in order to take advantage of these properties (such as if your catch block doesn't discriminate between your own exceptions and system ones), you can use the Error constructor.

For example:

function doSomethingErrorProne() {

if (ourCodeMakesAMistake()) {

throw new Error('The message');

} else {

doSomethingToGetAJavaScriptError();

}

}

try {

doSomethingErrorProne();

} catch (e) {

// Now, we actually use `console.error()`

console.error(e.name); // 'Error'

console.error(e.message); // 'The message', or a JavaScript error message

}

Copy to Clipboard

* [« Previous](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types)