Control flow and error handling

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 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 for complete information about expressions.

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:

```
{
  statement_1;
  statement_2;
  ⋮
  statement_n;
}
```

Example

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

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

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

Note: JavaScript before ECMAScript2015 (6th edition) **does not** have block scope! In older JavaScript, variables introduced within a block are scoped to the containing function or script,

and the effects of setting them persist beyond the block itself. In other words, *block statements do not define a scope*.

"Standalone" blocks in JavaScript can produce completely different results from what they would produce in C or Java. For example:

```
var x = 1;
{
  var x = 2;
}
console.log(x); // outputs 2
```

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 outputted `1`.)

Since ECMAScript2015, the `let` and `const` variable declarations are block-scoped. See the `let` and `const` reference pages for more information.

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:

```
if (condition) {
  statement_1;
} else {
  statement_2;
}
```

Here, the `condition` can be any expression that evaluates to `true` or `false`. (See Boolean 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 (condition_1) {
  statement_1;
} else if (condition_2) {
  statement_2;
} else if (condition_n) {
  statement_n;
} else {
  statement_last;
}
```

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) {
  statement_1_runs_if_condition_is_true;
  statement_2_runs_if_condition_is_true;
} else {
  statement_3_runs_if_condition_is_false;
  statement_4_runs_if_condition_is_false;
}
```

It's unwise to use simple assignments in a conditional expression, because the assignment can be confused with equality when glancing over the code.

For example, do *not* write code like this:

```
// Prone to being misread as "x == y"
if (x = y) {
  /* statements here */
}
```

If you need to use an assignment in a conditional expression, a common practice is to put additional parentheses around the assignment, like this:

```
if ((x = y)) {
  /* statements here */
}
```

Falsy values

The following values evaluate to `false` (also known as 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` object!

For example:

```
var b = new Boolean(false);
if (b)         // this condition evaluates to true
if (b == true) // this condition evaluates to 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`.

```
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:

```
switch (expression) {
  case label_1:
    statements_1
    [break;]
  case label_2:
    statements_2
    [break;]
    …
  default:
    statements_def
    [break;]
}
```

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?");
```

Exception handling statements

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

- `throw` statement
- `try...catch` statement

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
- `DOMException` and `DOMError`

`throw` statement

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

```
throw expression;
```

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: function() { return "I'm an object!"; } };
```

Note: You can specify an object when you throw an exception. You can then reference the object's properties in the `catch` block.

object's properties in the catch block.

```
// Create an object type UserException
function UserException(message) {
  this.message = message;
  this.name = 'UserException';
}

// Make the exception convert to a pretty string when used as a string
// (e.g., by the error console)
UserException.prototype.toString = function() {
  return `${this.name}: "${this.message}"`;
}

// Create an instance of the object type and throw it
throw new UserException('Value too high');
```

`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 = mo - 1; // Adjust month number for array index (1 = Jan, 12 = Dec)
  let months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul',
                'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
  if (months[mo]) {

    return months[mo];
  } else {
    throw '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 fun
}
```

The `catch` block

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

```
catch (catchID) {
  statements
}
```

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
}
```

> **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
}
```

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:

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
```

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"');
}

// OUTPUT


// caught inner "bogus"
// false
```

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 on the `try...catch` reference page.

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);    // logs 'Error'
  console.error(e.message); // logs 'The message', or a JavaScript error messa
}
```

Last modified: Jul 21, 2021, by MDN contributors