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.

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;
}
```

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

```
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**:**

```
if (condition) {
  statement_1;
} else {
  Statement_2;
}
```

If the `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;
}
```

To execute multiple statements, group them within a block statement (`{ /* … */ }`).

It's **not good** practice to have `if...else` with an assignment like `x = y` as a condition:

```
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.

Do not confuse the primitive boolean values true and false with the true and false values of the Boolean object!

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

```
switch (expression) {
  case label1:
    statements1;
    break;
  case label2:
    statements2;
    break;
  // …
  default:
    statementsDefault;
}
```

- 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 block** once the matched statement is executed.
If break is omitted, the program continues execution inside the switch statement (and will evaluate the next case, and so on).

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 "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;
  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.
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!";
  },
};
```

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.

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)
}
```

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.

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
}
```

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.

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.

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.

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.

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.

```
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
}
```