let

The **`let`** declaration declares re-assignable, block-scoped local variables, optionally initializing each to a value.

Try it

JavaScript Demo: Statement - Let

```
let x = 1;

if (x === 1) {
  let x = 2;

  console.log(x);
  // Expected output: 2
}

console.log(x);
// Expected output: 1
```

Run ›

Reset

Syntax

JS

```
let name1;
let name1 = value1;
let name1 = value1, name2 = value2;
let name1, name2 = value2;
let name1 = value1, name2, /* …, */ nameN = valueN;
```

Parameters

`nameN`

The name of the variable to declare. Each must be a legal JavaScript identifier or a destructuring binding pattern.

`valueN` Optional

Initial value of the variable. It can be any legal expression. Default value is `undefined`.

Description

The scope of a variable declared with `let` is one of the following curly-brace-enclosed syntaxes that most closely contains the `let` declaration:

- Block statement
- `switch` statement

- `try...catch` statement
- Body of one of the `for` statements, if the `let` is in the header of the statement
- Function body
- Static initialization block

Or if none of the above applies:

- The current module, for code running in module mode
- The global scope, for code running in script mode.

Compared with `var`, `let` declarations have the following differences:

- `let` declarations are scoped to blocks as well as functions.
- `let` declarations can only be accessed after the place of declaration is reached (see temporal dead zone). For this reason, `let` declarations are commonly regarded as non-hoisted.
- `let` declarations do not create properties on `globalThis` when declared at the top level of a script.
- `let` declarations cannot be redeclared by any other declaration in the same scope.
- `let` begins *declarations, not statements*. That means you cannot use a lone `let` declaration as the body of a block (which makes sense, since there's no way to access the variable).

```
JS
```

```
if (true) let a = 1; // SyntaxError: Lexical declaration cannot appear in a single-statement context
```

Note that `let` is allowed as an identifier name when declared with `var` or `function` in non-strict mode, but you should avoid using `let` as an identifier name to prevent unexpected syntax ambiguities.

Many style guides (including MDN's) recommend using `const` over `let` whenever a variable is not reassigned in its scope. This makes the intent clear that a variable's type (or value, in the case of a primitive) can never change. Others may prefer `let` for non-primitives that are mutated.

The list that follows the `let` keyword is called a *binding list* and is separated by commas, where the commas are *not* comma operators and the `=` signs are *not* assignment operators. Initializers of later variables can refer to earlier variables in the list.

Temporal dead zone (TDZ)

A variable declared with `let`, `const`, or `class` is said to be in a "temporal dead zone" (TDZ) from the start of the block until code execution reaches the place where the variable is declared and initialized.

While inside the TDZ, the variable has not been initialized with a value, and any attempt to access it will result in a `ReferenceError`. The variable is initialized with a value when execution reaches the place in the code where it was declared. If no initial value was specified with the variable declaration, it will be initialized with a value of `undefined`.

This differs from `var` variables, which will return a value of `undefined` if they are accessed before they are declared. The code below demonstrates the different result when `let` and `var` are accessed in code before the place where they are declared.

```
JS
```

```
{
  // TDZ starts at beginning of scope
  console.log(bar); // "undefined"
  console.log(foo); // ReferenceError: Cannot access 'foo' before initialization
  var bar = 1;
```

```
  let foo = 2; // End of TDZ (for foo)
}
```

The term "temporal" is used because the zone depends on the order of execution (time) rather than the order in which the code is written (position). For example, the code below works because, even though the function that uses the `let` variable appears before the variable is declared, the function is *called* outside the TDZ.

JS

```
{
  // TDZ starts at beginning of scope
  const func = () => console.log(letVar); // OK

  // Within the TDZ letVar access throws `ReferenceError`

  let letVar = 3; // End of TDZ (for letVar)
  func(); // Called outside TDZ!
}
```

Using the `typeof` operator for a `let` variable in its TDZ will throw a `ReferenceError`:

JS

```
typeof i; // ReferenceError: Cannot access 'i' before initialization
let i = 10;
```

This differs from using `typeof` for undeclared variables, and variables that hold a value of `undefined`:

JS

```
console.log(typeof undeclaredVariable); // "undefined"
```

> **Note:** `let` and `const` declarations are only processed when the current script gets processed. If you have two `<script>` elements running in script mode within one HTML, the first script is not subject to the TDZ restrictions for top-level `let` or `const` variables declared in the second script, although if you declare a `let` or `const` variable in the first script, declaring it again in the second script will cause a redeclaration error.

Redeclarations

`let` declarations cannot be in the same scope as any other declaration, including `let`, `const`, `class`, `function`, `var`, and `import` declaration.

JS

```
{
  let foo;
  let foo; // SyntaxError: Identifier 'foo' has already been declared
}
```

A `let` declaration within a function's body cannot have the same name as a parameter. A `let` declaration within a `catch` block cannot have the same name as the `catch`-bound identifier.

JS

```
function foo(a) {
  let a = 1; // SyntaxError: Identifier 'a' has already been declared
}
try {
```

```
} catch (e) {
  let e; // SyntaxError: Identifier 'e' has already been declared
}
```

If you're experimenting in a REPL, such as the Firefox web console (**Tools** > **Web Developer** > **Web Console**), and you run two `let` declarations with the same name in two separate inputs, you may get the same re-declaration error. See further discussion of this issue in Firefox bug 1580891 . The Chrome console allows `let` re-declarations between different REPL inputs.

You may encounter errors in `switch` statements because there is only one block.

JS

```
let x = 1;

switch (x) {
  case 0:
    let foo;
    break;
  case 1:
    let foo; // SyntaxError: Identifier 'foo' has already been declared
    break;
}
```

To avoid the error, wrap each `case` in a new block statement.

JS

```
let x = 1;

switch (x) {
  case 0: {
    let foo;
    break;
  }
  case 1: {
    let foo;
    break;
  }
}
```

Examples

Scoping rules

Variables declared by `let` have their scope in the block for which they are declared, as well as in any contained sub-blocks. In this way, `let` works very much like `var` . The main difference is that the scope of a `var` variable is the entire enclosing function:

JS

```
function varTest() {
  var x = 1;
  {
    var x = 2; // same variable!
    console.log(x); // 2
  }
  console.log(x); // 2
}

function letTest() {
  let x = 1;
  {
    let x = 2; // different variable
```

```
    console.log(x); // 2
  }
  console.log(x); // 1
}
```

At the top level of programs and functions, `let`, unlike `var`, does not create a property on the global object. For example:

JS

```
var x = "global";
let y = "global";
console.log(this.x); // "global"
console.log(this.y); // undefined
```

TDZ combined with lexical scoping

The following code results in a `ReferenceError` at the line shown:

JS

```
function test() {
  var foo = 33;
  if (foo) {
    let foo = foo + 55; // ReferenceError
  }
}
test();
```

The `if` block is evaluated because the outer `var foo` has a value. However due to lexical scoping this value is not available inside the block: the identifier `foo` *inside* the `if` block is the `let foo`. The expression `foo + 55` throws a `ReferenceError` because initialization of `let foo` has not completed — it is still in the temporal dead zone.

This phenomenon can be confusing in a situation like the following. The instruction `let n of n.a` is already inside the scope of the `for...of` loop's block. So, the identifier `n.a` is resolved to the property `a` of the `n` object located in the first part of the instruction itself (`let n`). This is still in the temporal dead zone as its declaration statement has not been reached and terminated.

JS

```
function go(n) {
  // n here is defined!
  console.log(n); // { a: [1, 2, 3] }

  for (let n of n.a) {
    //          ^ ReferenceError
    console.log(n);
  }
}

go({ a: [1, 2, 3] });
```

Other situations

When used inside a block, `let` limits the variable's scope to that block. Note the difference between `var`, whose scope is inside the function where it is declared.

JS

```
var a = 1;
var b = 2;

{
  var a = 11; // the scope is global
```

```
  let b = 22; // the scope is inside the block

  console.log(a); // 11
  console.log(b); // 22
}

console.log(a); // 11
console.log(b); // 2
```

However, this combination of `var` and `let` declarations below is a `SyntaxError` because `var` not being block-scoped, leading to them being in the same scope. This results in an implicit re-declaration of the variable.

JS

```
let x = 1;

{
  var x = 2; // SyntaxError for re-declaration
}
```

Declaration with destructuring

The left-hand side of each `=` can also be a binding pattern. This allows creating multiple variables at once.

JS

```
const result = /(a+)(b+)(c+)/.exec("aaabcc");
let [, a, b, c] = result;
console.log(a, b, c); // "aaa" "b" "cc"
```

For more information, see Destructuring assignment.

Specifications

Specification
ECMAScript Language Specification # sec-let-and-const-declarations

Browser compatibility

Report problems with this compatibility data on GitHub

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android
`let`	Chrome 49	Edge 14	Firefox 44	Opera 17	Safari 10	Chrome 49 Android	Firefox 44 for Android	Opera Androi

Tip: you can click/tap on a cell for more information.

Full support Partial support See implementation notes. Has more compatibility info.

See also

- `var`
- `const`
- Hoisting
- ES6 In Depth: `let` and `const` on hacks.mozilla.org (2015)
- Breaking changes in `let` and `const` in Firefox 44 on blog.mozilla.org (2015)
- You Don't Know JS: Scope & Closures, Ch.3: Function vs. Block Scope by Kyle Simpson
- What is the Temporal Dead Zone? on Stack Overflow
- What is the difference between using `let` and `var`? on Stack Overflow
- Why was the name 'let' chosen for block-scoped variable declarations in JavaScript? on Stack Overflow

return

The **`return`** statement ends function execution and specifies a value to be returned to the function caller.

Try it

JavaScript Demo: Statement - Return

```
function getRectArea(width, height) {
  if (width > 0 && height > 0) {
    return width * height;
  }
  return 0;
}

console.log(getRectArea(3, 4));
// Expected output: 12

console.log(getRectArea(-3, 4));
// Expected output: 0

```

Run ›

Reset

Syntax

JS

```
return;
return expression;
```

`expression` Optional

The expression whose value is to be returned. If omitted, `undefined` is returned.

Description

The `return` statement can only be used within function bodies. When a `return` statement is used in a function body, the execution of the function is stopped. The `return` statement has different effects when placed in different functions:

- In a plain function, the call to that function evaluates to the return value.
- In an async function, the produced promise is resolved with the returned value.
- In a generator function, the produced generator object's `next()` method returns `{ done: true, value: returnedValue }`.
- In an async generator function, the produced async generator object's `next()` method returns a promise fulfilled with `{ done: true, value: returnedValue }`.

If a `return` statement is executed within a `try` block, its `finally` block, if present, is first executed, before the value is actually returned.

Automatic semicolon insertion

The syntax forbids line terminators between the `return` keyword and the expression to be returned.

JS

```
return
a + b;
```

The code above is transformed by automatic semicolon insertion (ASI) into:

JS

```
return;
a + b;
```

This makes the function return `undefined` and the `a + b` expression is never evaluated. This may generate a warning in the console.

To avoid this problem (to prevent ASI), you could use parentheses:

JS

```
return (
  a + b
);
```

Examples

Interrupt a function

A function immediately stops at the point where `return` is called.

JS

```
function counter() {
  // Infinite loop
  for (let count = 1; ; count++) {
    console.log(`${count}A`); // Until 5
    if (count === 5) {
      return;
    }
    console.log(`${count}B`); // Until 4
  }
  console.log(`${count}C`); // Never appears
}

counter();

// Logs:
// 1A
// 1B
// 2A
// 2B
// 3A
// 3B
// 4A
// 4B
// 5A
```

Returning a function

See also the article about Closures.

JS

```
function magic() {
  return function calc(x) {
    return x * 42;
  };
}

const answer = magic();
answer(1337); // 56154
```

Specifications

Specification
ECMAScript Language Specification # sec-return-statement

Browser compatibility

Report problems with this compatibility data on GitHub

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS
`return`	Chrome 1	Edge 12	Firefox 1	Opera 3	Safari 1	Chrome 18 Android	Firefox 4 for Android	Opera 10.1 Android	Safari 1 on iOS

Tip: you can click/tap on a cell for more information.

Full support

See also

- Functions
- Closures

Help improve MDN

Was this page helpful to you?

Yes No

Learn how to contribute.

This page was last modified on Aug 11, 2023 by MDN contributors.

var

The **`var`** statement declares function-scoped or globally-scoped variables, optionally initializing each to a value.

Try it

JavaScript Demo: Statement - Var

```
var x = 1;

if (x === 1) {
  var x = 2;

  console.log(x);
  // Expected output: 2
}

console.log(x);
// Expected output: 2
```

Run ›

Reset

Syntax

JS

```
var name1;
var name1 = value1;
var name1 = value1, name2 = value2;
var name1, name2 = value2;
var name1 = value1, name2, /* …, */ nameN = valueN;
```

`nameN`

The name of the variable to declare. Each must be a legal JavaScript identifier or a destructuring binding pattern.

`valueN` Optional

Initial value of the variable. It can be any legal expression. Default value is `undefined`.

Description

The scope of a variable declared with `var` is one of the following curly-brace-enclosed syntaxes that most closely contains the `var` statement:

- Function body
- Static initialization block

Or if none of the above applies:

- The current module, for code running in module mode
- The global scope, for code running in script mode.

JS

```
function foo() {
  var x = 1;
  function bar() {
    var y = 2;
    console.log(x); // 1 (function `bar` closes over `x`)
    console.log(y); // 2 (`y` is in scope)
  }
  bar();
  console.log(x); // 1 (`x` is in scope)
  console.log(y); // ReferenceError, `y` is scoped to `bar`
}

foo();
```

Importantly, other block constructs, including block statements, `try...catch`, `switch`, headers of one of the `for` statements, do not create scopes for `var`, and variables declared with `var` inside such a block can continue to be referenced outside the block.

JS

```
for (var a of [1, 2, 3]);
console.log(a); // 3
```

In a script, a variable declared using `var` is added as a non-configurable property of the global object. This means its property descriptor cannot be changed and it cannot be deleted using `delete`. JavaScript has automatic memory management, and it would make no sense to be able to use the `delete` operator on a global variable.

JS

```
"use strict";
var x = 1;
Object.hasOwn(globalThis, "x"); // true
delete globalThis.x; // TypeError in strict mode. Fails silently otherwise.
delete x; // SyntaxError in strict mode. Fails silently otherwise.
```

In both NodeJS CommonJS modules and native ECMAScript modules, top-level variable declarations are scoped to the module, and are not added as properties to the global object.

The list that follows the `var` keyword is called a *binding list* and is separated by commas, where the commas are *not* comma operators and the `=` signs are *not* assignment operators. Initializers of later variables can refer to earlier variables in the list and get the initialized value.

Hoisting

`var` declarations, wherever they occur in a script, are processed before any code within the script is executed. Declaring a variable anywhere in the code is equivalent to declaring it at the top. This also means that a variable can appear to be used before it's declared. This behavior is called *hoisting*, as it appears that the variable declaration is moved to the top of the function, static initialization block, or script source in which it occurs.

> **Note:** `var` declarations are only hoisted to the top of the current script. If you have two `<script>` elements within one HTML, the first script cannot access variables declared by the second before the second script has been processed and executed.

JS

```
bla = 2;
var bla;
```

This is implicitly understood as:

```JS
var bla;
bla = 2;
```

For that reason, it is recommended to always declare variables at the top of their scope (the top of global code and the top of function code) so it's clear which variables are scoped to the current function.

Only a variable's declaration is hoisted, not its initialization. The initialization happens only when the assignment statement is reached. Until then the variable remains `undefined` (but declared):

```JS
function doSomething() {
  console.log(bar); // undefined
  var bar = 111;
  console.log(bar); // 111
}
```

This is implicitly understood as:

```JS
function doSomething() {
  var bar;
  console.log(bar); // undefined
  bar = 111;
  console.log(bar); // 111
}
```

Redeclarations

Duplicate variable declarations using `var` will not trigger an error, even in strict mode, and the variable will not lose its value, unless the declaration has an initializer.

```JS
var a = 1;
var a = 2;
console.log(a); // 2
var a;
console.log(a); // 2; not undefined
```

`var` declarations can also be in the same scope as a `function` declaration. In this case, the `var` declaration's initializer always overrides the function's value, regardless of their relative position. This is because function declarations are hoisted before any initializer gets evaluated, so the initializer comes later and overrides the value.

```JS
var a = 1;
function a() {}
console.log(a); // 1
```

`var` declarations cannot be in the same scope as a `let`, `const`, `class`, or `import` declaration.

```JS
var a = 1;
let a = 2; // SyntaxError: Identifier 'a' has already been declared
```

Because `var` declarations are not scoped to blocks, this also applies to the following case:

JS

```
let a = 1;
{
  var a = 1; // SyntaxError: Identifier 'a' has already been declared
}
```

It does not apply to the following case, where `let` is in a child scope of `var`, not the same scope:

JS

```
var a = 1;
{
  let a = 2;
}
```

A `var` declaration within a function's body can have the same name as a parameter.

JS

```
function foo(a) {
  var a = 1;
  console.log(a);
}

foo(2); // Logs 1
```

A `var` declaration within a `catch` block can have the same name as the `catch`-bound identifier, but only if the `catch` binding is a simple identifier, not a destructuring pattern. This is a deprecated syntax and you should not rely on it. In this case, the declaration is hoisted to outside the `catch` block, but any value assigned within the `catch` block is not visible outside.

JS

```
try {
  throw 1;
} catch (e) {
  var e = 2; // Works
}
console.log(e); // undefined
```

Examples

Declaring and initializing two variables

JS

```
var a = 0,
  b = 0;
```

Assigning two variables with single string value

JS

```
var a = "A";
var b = a;
```

This is equivalent to:

JS

```
var a, b = a = "A";
```

Be mindful of the order:

```
JS
var x = y,
  y = "A";
console.log(x, y); // undefined A
```

Here, `x` and `y` are declared before any code is executed, but the assignments occur later. At the time `x = y` is evaluated, `y` exists so no `ReferenceError` is thrown and its value is `undefined`. So, `x` is assigned the undefined value. Then, `y` is assigned the value `"A"`.

Initialization of several variables

Be careful of the `var x = y = 1` syntax — `y` is not actually declared as a variable, so `y = 1` is an unqualified identifier assignment, which creates a global variable in non-strict mode.

```
JS
var x = 0;
function f() {
  var x = y = 1; // Declares x locally; declares y globally.
}
f();

console.log(x, y); // 0 1

// In non-strict mode:
// x is the global one as expected;
// y is leaked outside of the function, though!
```

The same example as above but with a strict mode:

```
JS
"use strict";

var x = 0;
function f() {
  var x = y = 1; // ReferenceError: y is not defined
}
f();

console.log(x, y);
```

Implicit globals and outer function scope

Variables that appear to be implicit globals may be references to variables in an outer function scope:

```
JS
var x = 0; // Declares x within file scope, then assigns it a value of 0.

console.log(typeof z); // "undefined", since z doesn't exist yet

function a() {
  var y = 2; // Declares y within scope of function a, then assigns it a value of 2.

  console.log(x, y); // 0 2

  function b() {
    x = 3; // Assigns 3 to existing file scoped x.
    y = 4; // Assigns 4 to existing outer y.
    z = 5; // Creates a new global variable z, and assigns it a value of 5.
```

```
    // (Throws a ReferenceError in strict mode.)
  }

  b(); // Creates z as a global variable.
  console.log(x, y, z); // 3 4 5
}

a(); // Also calls b.
console.log(x, z); // 3 5
console.log(typeof y); // "undefined", as y is local to function a
```

Declaration with destructuring

The left-hand side of each `=` can also be a binding pattern. This allows creating multiple variables at once.

JS

```
const result = /(a+)(b+)(c+)/.exec("aaabcc");
var [, a, b, c] = result;
console.log(a, b, c); // "aaa" "b" "cc"
```

For more information, see Destructuring assignment.

Specifications

Specification
ECMAScript Language Specification # sec-variable-statement

Browser compatibility

Report problems with this compatibility data on GitHub

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	
var	Chrome 1	Edge 12	Firefox 1	Opera 3	Safari 1	Chrome 18 Android	Firefox 4 for Android	Opera 10.1 Android	Safari 1 on iOS	

Tip: you can click/tap on a cell for more information.

Full support

See also

- `let`
- `const`

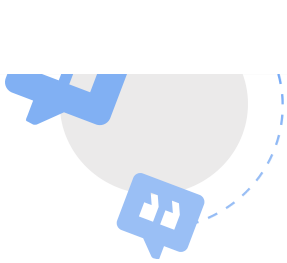
Help improve MDN

Was this page helpful to you?

Yes No

Learn how to contribute.

This page was last modified on Sep 12, 2023 by MDN contributors.



JavaScript first steps

In our first JavaScript module, we first answer some fundamental questions such as "what is JavaScript?", "what does it look like?", and "what can it do?", before moving on to taking you through your first practical experience of writing JavaScript. After that, we discuss some key building blocks in detail, such as variables, strings, numbers and arrays.

Prerequisites

Before starting this module, you don't need any previous JavaScript knowledge, but you should have some familiarity with HTML and CSS. You are advised to work through the following modules before starting on JavaScript:

- Getting started with the Web (which includes a really basic JavaScript introduction).
- Introduction to HTML.
- Introduction to CSS.

> **Note:** If you are working on a computer/tablet/other device where you don't have the ability to create your own files, you could try out (most of) the code examples in an online coding program such as JSBin or Glitch .

Guides

What is JavaScript?

Welcome to the MDN beginner's JavaScript course! In this first article we will look at JavaScript from a high level, answering questions such as "what is it?", and "what is it doing?", and making sure you are comfortable with JavaScript's purpose.

A first splash into JavaScript

Now you've learned something about the theory of JavaScript, and what you can do with it, we are going to give you a crash course on the basic features of JavaScript via a completely practical tutorial. Here you'll build up a simple "Guess the number" game, step by step.

What went wrong? Troubleshooting JavaScript

When you built up the "Guess the number" game in the previous article, you may have found that it didn't work. Never fear — this article aims to save you from tearing your hair out over such problems by providing you with some simple tips on how to find and fix errors in JavaScript programs.

Storing the information you need — Variables

After reading the last couple of articles you should now know what JavaScript is, what it can do for you, how you use it alongside other web technologies, and what its main features look like from a high level. In this article, we will get down to the real basics, looking at how to work with the most basic building blocks of JavaScript — Variables.

Basic math in JavaScript — numbers and operators

At this point in the course, we discuss maths in JavaScript — how we can combine operators and other features to successfully manipulate numbers to do our bidding.

Handling text — strings in JavaScript

Next, we'll turn our attention to strings — this is what pieces of text are called in programming. In this article, we'll look at all the common things that you really ought to know about strings when learning JavaScript, such as creating strings, escaping quotes in strings, and joining them together.

Useful string methods

Now we've looked at the very basics of strings, let's move up a gear and start thinking about what useful operations we can do on strings with built-in methods, such as finding the length of a text string, joining and splitting strings, substituting one character in a string for another, and more.

Arrays

In the final article of this module, we'll look at arrays — a neat way of storing a list of data items under a single variable name. Here we look at why this is useful, then explore how to create an array, retrieve, add, and remove items stored in an array, and more besides.

Assessments

The following assessment will test your understanding of the JavaScript basics covered in the guides above.

Silly story generator

In this assessment, you'll be tasked with taking some of the knowledge you've picked up in this module's articles and applying it to creating a fun app that generates random silly stories. Have fun!

See also

Learn JavaScript

An excellent resource for aspiring web developers — Learn JavaScript in an interactive environment, with short lessons and interactive tests, guided by automated assessment. The first 40 lessons are free, and the complete course is available for a small one-time payment.

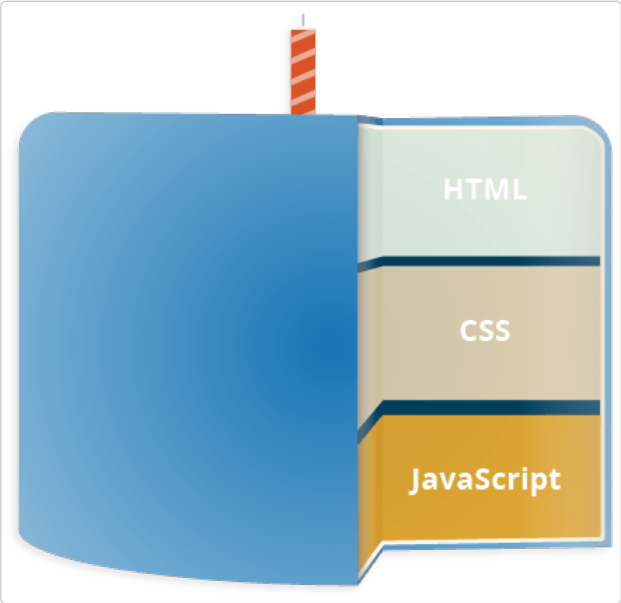
What is JavaScript?

Welcome to the MDN beginner's JavaScript course! In this article we will look at JavaScript from a high level, answering questions such as "What is it?" and "What can you do with it?", and making sure you are comfortable with JavaScript's purpose.

Prerequisites:	A basic understanding of HTML and CSS.
Objective:	To gain familiarity with what JavaScript is, what it can do, and how it fits into a website.

A high-level definition

JavaScript is a scripting or programming language that allows you to implement complex features on web pages — every time a web page does more than just sit there and display static information for you to look at — displaying timely content updates, interactive maps, animated 2D/3D graphics, scrolling video jukeboxes, etc. — you can bet that JavaScript is probably involved. It is the third layer of the layer cake of standard web technologies, two of which (HTML and CSS) we have covered in much more detail in other parts of the Learning Area.

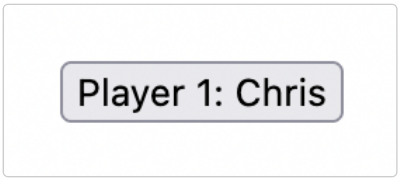


- HTML is the markup language that we use to structure and give meaning to our web content, for example defining paragraphs, headings, and data tables, or embedding images and videos in the page.
- CSS is a language of style rules that we use to apply styling to our HTML content, for example setting background colors and fonts, and laying out our content in multiple columns.
- JavaScript is a scripting language that enables you to create dynamically updating content, control multimedia, animate images, and pretty much everything else. (Okay, not everything, but it is amazing what you can achieve with a few lines of JavaScript code.)

The three layers build on top of one another nicely. Let's take a button as an example. We can mark it up using HTML to give it structure and purpose:

HTML Play

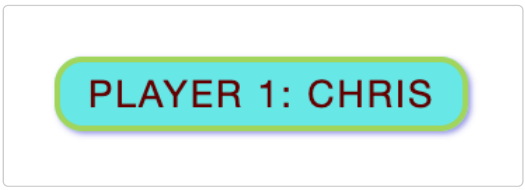
```
<button type="button">Player 1: Chris</button>
```



Then we can add some CSS into the mix to get it looking nice:

CSS Play

```css
button {
  font-family: "helvetica neue", helvetica, sans-serif;
  letter-spacing: 1px;
  text-transform: uppercase;
  border: 2px solid rgb(200 200 0 / 60%);
  background-color: rgb(0 217 217 / 60%);
  color: rgb(100 0 0 / 100%);
  box-shadow: 1px 1px 2px rgb(0 0 200 / 40%);
  border-radius: 10px;
  padding: 3px 10px;
  cursor: pointer;
}
```



And finally, we can add some JavaScript to implement dynamic behavior:

JS Play

```js
const button = document.querySelector("button");

button.addEventListener("click", updateName);

function updateName() {
  const name = prompt("Enter a new name");
  button.textContent = `Player 1: ${name}`;
}
```

Play

Try clicking on this last version of the text label to see what happens (note also that you can find this demo on GitHub — see the source code , or run it live)!

JavaScript can do a lot more than that — let's explore what in more detail.

So what can it really do?

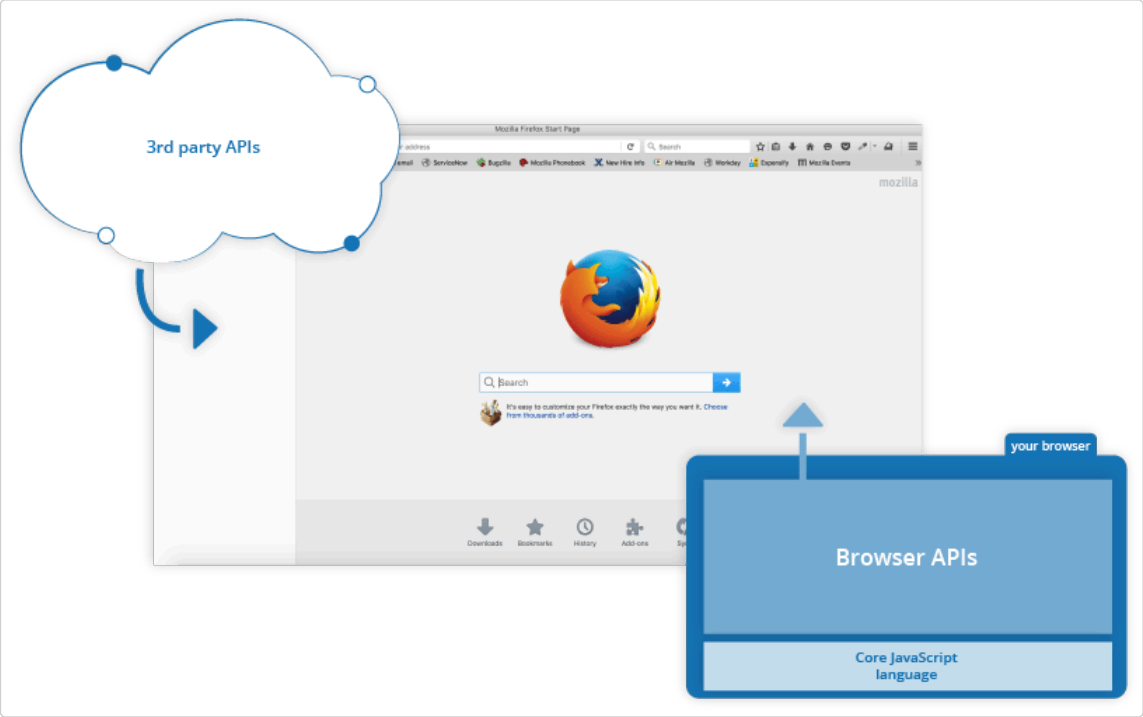
The core client-side JavaScript language consists of some common programming features that allow you to do things like:

- Store useful values inside variables. In the above example for instance, we ask for a new name to be entered then store that name in a variable called `name`.
- Operations on pieces of text (known as "strings" in programming). In the above example we take the string "Player 1: " and join it to the `name` variable to create the complete text label, e.g. "Player 1: Chris".
- Running code in response to certain events occurring on a web page. We used a `click` event in our example above to detect when the label is clicked and then run the code that updates the text label.
- And much more!

What is even more exciting however is the functionality built on top of the client-side JavaScript language. So-called **Application Programming Interfaces** (**APIs**) provide you with extra superpowers to use in your JavaScript code.

APIs are ready-made sets of code building blocks that allow a developer to implement programs that would otherwise be hard or impossible to implement. They do the same thing for programming that ready-made furniture kits do for home building — it is much easier to take ready-cut panels and screw them together to make a bookshelf than it is to work out the design yourself, go and find the correct wood, cut all the panels to the right size and shape, find the correct-sized screws, and *then* put them together to make a bookshelf.

They generally fall into two categories.



Browser APIs are built into your web browser, and are able to expose data from the surrounding computer environment, or do useful complex things. For example:

- The `DOM (Document Object Model) API` allows you to manipulate HTML and CSS, creating, removing and changing HTML, dynamically applying new styles to your page, etc. Every time you see a popup window appear on a page, or some new content displayed (as we saw above in our simple demo) for example, that's the DOM in action.
- The `Geolocation API` retrieves geographical information. This is how Google Maps is able to find your location and plot it on a map.
- The `Canvas` and `WebGL` APIs allow you to create animated 2D and 3D graphics. People are doing some amazing things using these web technologies — see Chrome Experiments and webglsamples.
- Audio and Video APIs like `HTMLMediaElement` and `WebRTC` allow you to do really interesting things with multimedia, such as play audio and video right in a web page, or grab video from your web camera and display it on someone else's computer (try our

simple Snapshot demo to get the idea).

> **Note:** Many of the above demos won't work in an older browser — when experimenting, it's a good idea to use a modern browser like Firefox, Chrome, Edge or Opera to run your code in. You will need to consider cross browser testing in more detail when you get closer to delivering production code (i.e. real code that real customers will use).

Third party APIs are not built into the browser by default, and you generally have to grab their code and information from somewhere on the Web. For example:

- The Twitter API allows you to do things like displaying your latest tweets on your website.
- The Google Maps API and OpenStreetMap API allows you to embed custom maps into your website, and other such functionality.

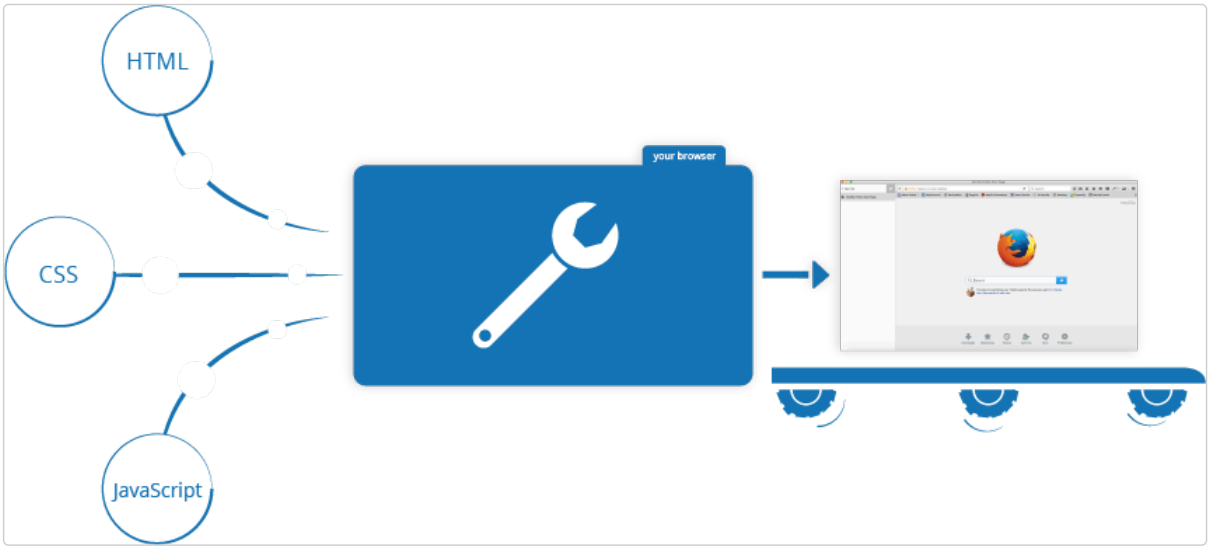
> **Note:** These APIs are advanced, and we'll not be covering any of these in this module. You can find out much more about these in our Client-side web APIs module.

There's a lot more available, too! However, don't get over excited just yet. You won't be able to build the next Facebook, Google Maps, or Instagram after studying JavaScript for 24 hours — there are a lot of basics to cover first. And that's why you're here — let's move on!

What is JavaScript doing on your page?

Here we'll actually start looking at some code, and while doing so, explore what actually happens when you run some JavaScript in your page.

Let's briefly recap the story of what happens when you load a web page in a browser (first talked about in our How CSS works article). When you load a web page in your browser, you are running your code (the HTML, CSS, and JavaScript) inside an execution environment (the browser tab). This is like a factory that takes in raw materials (the code) and outputs a product (the web page).



A very common use of JavaScript is to dynamically modify HTML and CSS to update a user interface, via the Document Object Model API (as mentioned above). Note that the code in your web documents is generally loaded and executed in the order it appears on the page. Errors may occur if JavaScript is loaded and run before the HTML and CSS that it is intended to modify. You will learn ways around this later in the article, in the Script loading strategies section.

Browser security

Each browser tab has its own separate bucket for running code in (these buckets are called "execution environments" in technical terms) — this means that in most cases the code in each tab is run completely separately, and the code in one tab cannot directly affect the code in another tab — or on another website. This is a good security measure — if this were not the case, then pirates could start writing code to steal information from other websites, and other such bad things.

> **Note:** There are ways to send code and data between different websites/tabs in a safe manner, but these are advanced techniques that we won't cover in this course.

JavaScript running order

When the browser encounters a block of JavaScript, it generally runs it in order, from top to bottom. This means that you need to be careful what order you put things in. For example, let's return to the block of JavaScript we saw in our first example:

```
JS
const button = document.querySelector("button");

button.addEventListener("click", updateName);

function updateName() {
  const name = prompt("Enter a new name");
  button.textContent = `Player 1: ${name}`;
}
```

Here we are selecting a button (line 1), then attaching an event listener to it (line 3) so that when the button is clicked, the `updateName()` code block (lines 5–8) is run. The `updateName()` code block (these types of reusable code blocks are called "functions") asks the user for a new name, and then inserts that name into the button text to update the display.

If you swapped the order of the first two lines of code, it would no longer work — instead, you'd get an error returned in the browser developer console — `Uncaught ReferenceError: Cannot access 'button' before initialization`. This means that the `button` object has not been initialized yet, so we can't add an event listener to it.

> **Note:** This is a very common error — you need to be careful that the objects referenced in your code exist before you try to do stuff to them.

Interpreted versus compiled code

You might hear the terms **interpreted** and **compiled** in the context of programming. In interpreted languages, the code is run from top to bottom and the result of running the code is immediately returned. You don't have to transform the code into a different form before the browser runs it. The code is received in its programmer-friendly text form and processed directly from that.

Compiled languages on the other hand are transformed (compiled) into another form before they are run by the computer. For example, C/C++ are compiled into machine code that is then run by the computer. The program is executed from a binary format, which was generated from the original program source code.

JavaScript is a lightweight interpreted programming language. The web browser receives the JavaScript code in its original text form and runs the script from that. From a technical standpoint, most modern JavaScript interpreters actually use a technique called **just-in-time compiling** to improve performance; the JavaScript source code gets compiled into a faster, binary format while the script is being used, so that it can be run as quickly as possible. However, JavaScript is still considered an interpreted language, since the compilation is handled at run time, rather than ahead of time.

There are advantages to both types of language, but we won't discuss them right now.

Server-side versus client-side code

You might also hear the terms **server-side** and **client-side** code, especially in the context of web development. Client-side code is code that is run on the user's computer — when a web page is viewed, the page's client-side code is downloaded, then run and displayed by the browser. In this module we are explicitly talking about **client-side JavaScript**.

Server-side code on the other hand is run on the server, then its results are downloaded and displayed in the browser. Examples of popular server-side web languages include PHP, Python, Ruby, ASP.NET, and even JavaScript! JavaScript can also be used as a server-side language, for example in the popular Node.js environment — you can find out more about server-side JavaScript in our Dynamic Websites – Server-side programming topic.

Dynamic versus static code

The word **dynamic** is used to describe both client-side JavaScript, and server-side languages — it refers to the ability to update the display of a web page/app to show different things in different circumstances, generating new content as required. Server-side code dynamically generates new content on the server, e.g. pulling data from a database, whereas client-side JavaScript dynamically generates new content inside the browser on the client, e.g. creating a new HTML table, filling it with data requested from the server, then displaying the table in a web page shown to the user. The meaning is slightly different in the two contexts, but related, and both approaches (server-side and client-side) usually work together.

A web page with no dynamically updating content is referred to as **static** — it just shows the same content all the time.

How do you add JavaScript to your page?

JavaScript is applied to your HTML page in a similar manner to CSS. Whereas CSS uses `<link>` elements to apply external stylesheets and `<style>` elements to apply internal stylesheets to HTML, JavaScript only needs one friend in the world of HTML — the `<script>` element. Let's learn how this works.

Internal JavaScript

1. First of all, make a local copy of our example file apply-javascript.html . Save it in a directory somewhere sensible.
2. Open the file in your web browser and in your text editor. You'll see that the HTML creates a simple web page containing a clickable button.
3. Next, go to your text editor and add the following in your head — just before your closing `</head>` tag:

```HTML
<script>
  // JavaScript goes here
</script>
```

4. Now we'll add some JavaScript inside our `<script>` element to make the page do something more interesting — add the following code just below the "// JavaScript goes here" line:

```JS
document.addEventListener("DOMContentLoaded", () => {
  function createParagraph() {
    const para = document.createElement("p");
    para.textContent = "You clicked the button!";
    document.body.appendChild(para);
  }

  const buttons = document.querySelectorAll("button");

  for (const button of buttons) {
    button.addEventListener("click", createParagraph);
```

```
  }
});
```

5. Save your file and refresh the browser — now you should see that when you click the button, a new paragraph is generated and placed below.

> **Note:** If your example doesn't seem to work, go through the steps again and check that you did everything right. Did you save your local copy of the starting code as a `.html` file? Did you add your `<script>` element just before the `</head>` tag? Did you enter the JavaScript exactly as shown? **JavaScript is case sensitive, and very fussy, so you need to enter the syntax exactly as shown, otherwise it may not work.**

> **Note:** You can see this version on GitHub as apply-javascript-internal.html (see it live too).

External JavaScript

This works great, but what if we wanted to put our JavaScript in an external file? Let's explore this now.

1. First, create a new file in the same directory as your sample HTML file. Call it `script.js` — make sure it has that .js filename extension, as that's how it is recognized as JavaScript.
2. Replace your current `<script>` element with the following:

```
HTML
<script src="script.js" defer></script>
```

3. Inside `script.js`, add the following script:

```
JS
function createParagraph() {
  const para = document.createElement("p");
  para.textContent = "You clicked the button!";
  document.body.appendChild(para);
}

const buttons = document.querySelectorAll("button");

for (const button of buttons) {
  button.addEventListener("click", createParagraph);
}
```

4. Save and refresh your browser, and you should see the same thing! It works just the same, but now we've got our JavaScript in an external file. This is generally a good thing in terms of organizing your code and making it reusable across multiple HTML files. Plus, the HTML is easier to read without huge chunks of script dumped in it.

> **Note:** You can see this version on GitHub as apply-javascript-external.html and script.js (see it live too).

Inline JavaScript handlers

Note that sometimes you'll come across bits of actual JavaScript code living inside HTML. It might look something like this:

```
JS                                                                    Play
function createParagraph() {
  const para = document.createElement("p");
  para.textContent = "You clicked the button!";
```

```
  document.body.appendChild(para);
}
```

HTML Play

```
<button onclick="createParagraph()">Click me!</button>
```

You can try this version of our demo below.

Play

This demo has exactly the same functionality as in the previous two sections, except that the `<button>` element includes an inline `onclick` handler to make the function run when the button is pressed.

Please don't do this, however. It is bad practice to pollute your HTML with JavaScript, and it is inefficient — you'd have to include the `onclick="createParagraph()"` attribute on every button you want the JavaScript to apply to.

Using addEventListener instead

Instead of including JavaScript in your HTML, use a pure JavaScript construct. The `querySelectorAll()` function allows you to select all the buttons on a page. You can then loop through the buttons, assigning a handler for each using `addEventListener()`. The code for this is shown below:

JS

```
const buttons = document.querySelectorAll("button");

for (const button of buttons) {
  button.addEventListener("click", createParagraph);
}
```

This might be a bit longer than the `onclick` attribute, but it will work for all buttons — no matter how many are on the page, nor how many are added or removed. The JavaScript does not need to be changed.

> **Note:** Try editing your version of `apply-javascript.html` and add a few more buttons into the file. When you reload, you should find that all of the buttons when clicked will create a paragraph. Neat, huh?

Script loading strategies

There are a number of issues involved with getting scripts to load at the right time. Nothing is as simple as it seems! A common problem is that all the HTML on a page is loaded in the order in which it appears. If you are using JavaScript to manipulate elements on the page (or more accurately, the Document Object Model), your code won't work if the JavaScript is loaded and parsed before the HTML you are trying to do something to.

In the above code examples, in the internal and external examples the JavaScript is loaded and run in the head of the document, before the HTML body is parsed. This could cause an error, so we've used some constructs to get around it.

In the internal example, you can see this structure around the code:

```
JS
document.addEventListener("DOMContentLoaded", () => {
  // …
});
```

This is an event listener, which listens for the browser's `DOMContentLoaded` event, which signifies that the HTML body is completely loaded and parsed. The JavaScript inside this block will not run until after that event is fired, therefore the error is avoided (you'll learn about events later in the course).

In the external example, we use a more modern JavaScript feature to solve the problem, the `defer` attribute, which tells the browser to continue downloading the HTML content once the `<script>` tag element has been reached.

```
HTML
<script src="script.js" defer></script>
```

In this case both the script and the HTML will load simultaneously and the code will work.

> **Note:** In the external case, we did not need to use the `DOMContentLoaded` event because the `defer` attribute solved the problem for us. We didn't use the `defer` solution for the internal JavaScript example because `defer` only works for external scripts.

An old-fashioned solution to this problem used to be to put your script element right at the bottom of the body (e.g. just before the `</body>` tag), so that it would load after all the HTML has been parsed. The problem with this solution is that loading/parsing of the script is completely blocked until the HTML DOM has been loaded. On larger sites with lots of JavaScript, this can cause a major performance issue, slowing down your site.

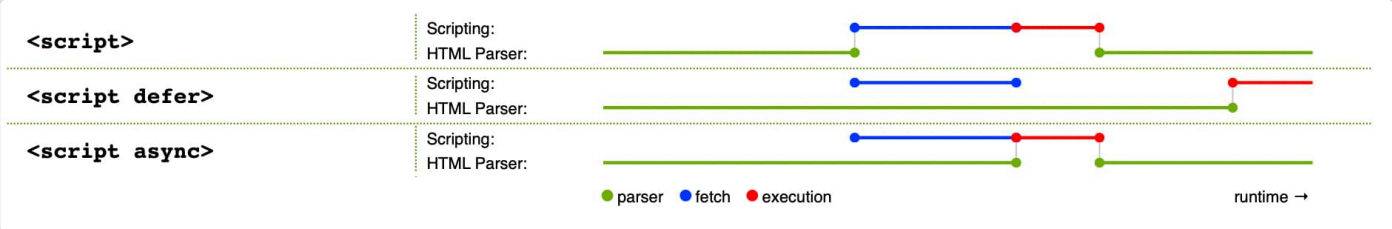
async and defer

There are actually two modern features we can use to bypass the problem of the blocking script — `async` and `defer` (which we saw above). Let's look at the difference between these two.

Scripts loaded using the `async` attribute will download the script without blocking the page while the script is being fetched. However, once the download is complete, the script will execute, which blocks the page from rendering. This means that the rest of the content on the web page is prevented from being processed and displayed to the user until the script finishes executing. You get no guarantee that scripts will run in any specific order. It is best to use `async` when the scripts in the page run independently from each other and depend on no other script on the page.

Scripts loaded with the `defer` attribute will load in the order they appear on the page. They won't run until the page content has all loaded, which is useful if your scripts depend on the DOM being in place (e.g. they modify one or more elements on the page).

Here is a visual representation of the different script loading methods and what that means for your page:



This image is from the HTML spec , copied and cropped to a reduced version, under CC BY 4.0 license terms.

For example, if you have the following script elements:

```HTML
<script async src="js/vendor/jquery.js"></script>

<script async src="js/script2.js"></script>

<script async src="js/script3.js"></script>
```

You can't rely on the order the scripts will load in. `jquery.js` may load before or after `script2.js` and `script3.js` and if this is the case, any functions in those scripts depending on `jquery` will produce an error because `jquery` will not be defined at the time the script runs.

`async` should be used when you have a bunch of background scripts to load in, and you just want to get them in place as soon as possible. For example, maybe you have some game data files to load, which will be needed when the game actually begins, but for now you just want to get on with showing the game intro, titles, and lobby, without them being blocked by script loading.

Scripts loaded using the `defer` attribute (see below) will run in the order they appear in the page and execute them as soon as the script and content are downloaded:

```HTML
<script defer src="js/vendor/jquery.js"></script>

<script defer src="js/script2.js"></script>

<script defer src="js/script3.js"></script>
```

In the second example, we can be sure that `jquery.js` will load before `script2.js` and `script3.js` and that `script2.js` will load before `script3.js`. They won't run until the page content has all loaded, which is useful if your scripts depend on the DOM being in place (e.g. they modify one or more elements on the page).

To summarize:

- `async` and `defer` both instruct the browser to download the script(s) in a separate thread, while the rest of the page (the DOM, etc.) is downloading, so the page loading is not blocked during the fetch process.
- scripts with an `async` attribute will execute as soon as the download is complete. This blocks the page and does not guarantee any specific execution order.
- scripts with a `defer` attribute will load in the order they are in and will only execute once everything has finished loading.
- If your scripts should be run immediately and they don't have any dependencies, then use `async`.
- If your scripts need to wait for parsing and depend on other scripts and/or the DOM being in place, load them using `defer` and put their corresponding `<script>` elements in the order you want the browser to execute them.

Comments

As with HTML and CSS, it is possible to write comments into your JavaScript code that will be ignored by the browser, and exist to provide instructions to your fellow developers on how the code works (and you, if you come back to your code after six months and can't remember what you did). Comments are very useful, and you should use them often, particularly for larger applications. There are two types:

- A single line comment is written after a double forward slash (//), e.g.

  JS

  ```
  // I am a comment
  ```

- A multi-line comment is written between the strings /* and */, e.g.

  JS

  ```
  /*
    I am also
    a comment
  */
  ```

So for example, we could annotate our last demo's JavaScript with comments like so:

JS

```
// Function: creates a new paragraph and appends it to the bottom of the HTML body.

function createParagraph() {
  const para = document.createElement("p");
  para.textContent = "You clicked the button!";
  document.body.appendChild(para);
}

/*
  1. Get references to all the buttons on the page in an array format.
  2. Loop through all the buttons and add a click event listener to each one.

  When any button is pressed, the createParagraph() function will be run.
*/

const buttons = document.querySelectorAll("button");

for (const button of buttons) {
  button.addEventListener("click", createParagraph);
}
```

> **Note:** In general more comments are usually better than less, but you should be careful if you find yourself adding lots of comments to explain what variables are (your variable names perhaps should be more intuitive), or to explain very simple operations (maybe your code is overcomplicated).

Summary

So there you go, your first step into the world of JavaScript. We've begun with just theory, to start getting you used to why you'd use JavaScript and what kind of things you can do with it. Along the way, you saw a few code examples and learned how JavaScript fits in with the rest of the code on your website, amongst other things.

JavaScript may seem a bit daunting right now, but don't worry — in this course, we will take you through it in simple steps that will make sense going forward. In the next article, we will plunge straight into the practical, getting you to jump straight in and build your own JavaScript examples.

A first splash into JavaScript

Now you've learned something about the theory of JavaScript and what you can do with it, we are going to give you an idea of what the process of creating a simple JavaScript program is like, by guiding you through a practical tutorial. Here you'll build up a simple "Guess the number" game, step by step.

Prerequisites:	A basic understanding of HTML and CSS, an understanding of what JavaScript is.
Objective:	To have a first bit of experience at writing some JavaScript, and gain at least a basic understanding of what writing a JavaScript program involves.

We want to set really clear expectations here: You won't be expected to learn JavaScript by the end of this article, or even understand all the code we are asking you to write. Instead, we want to give you an idea of how JavaScript's features work together, and what writing JavaScript feels like. In subsequent articles you'll revisit all the features shown here in a lot more detail, so don't worry if you don't understand it all immediately!

> **Note:** Many of the code features you'll see in JavaScript are the same as in other programming languages — functions, loops, etc. The code syntax looks different, but the concepts are still largely the same.

Thinking like a programmer

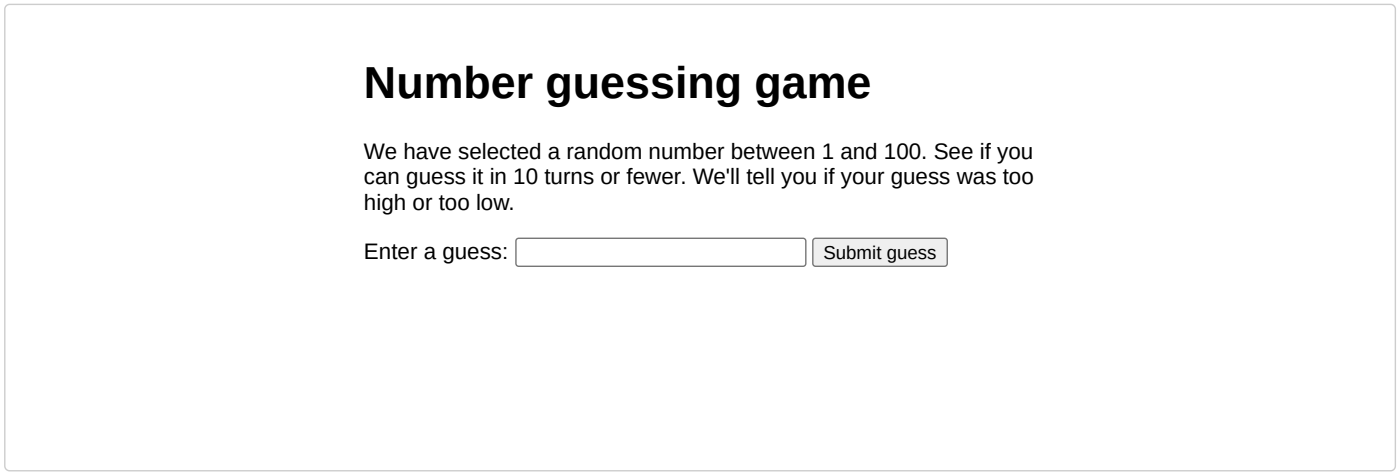
One of the hardest things to learn in programming is not the syntax you need to learn, but how to apply it to solve real-world problems. You need to start thinking like a programmer — this generally involves looking at descriptions of what your program needs to do, working out what code features are needed to achieve those things, and how to make them work together.

This requires a mixture of hard work, experience with the programming syntax, and practice — plus a bit of creativity. The more you code, the better you'll get at it. We can't promise that you'll develop "programmer brain" in five minutes, but we will give you plenty of opportunities to practice thinking like a programmer throughout the course.

With that in mind, let's look at the example we'll be building up in this article, and review the general process of dissecting it into tangible tasks.

Example — Guess the number game

In this article we'll show you how to build up the simple game you can see below:



Have a go at playing it — familiarize yourself with the game before you move on.

Let's imagine your boss has given you the following brief for creating this game:

> I want you to create a simple guess the number type game. It should choose a random number between 1 and 100, then challenge the player to guess the number in 10 turns. After each turn, the player should be told if they are right or wrong, and if they are wrong, whether the guess was too low or too high. It should also tell the player what numbers they previously guessed. The game will end once the player guesses correctly, or once they run out of turns. When the game ends, the player should be given an option to start playing again.

Upon looking at this brief, the first thing we can do is to start breaking it down into simple actionable tasks, in as much of a programmer mindset as possible:

1. Generate a random number between 1 and 100.
2. Record the turn number the player is on. Start it on 1.
3. Provide the player with a way to guess what the number is.
4. Once a guess has been submitted first record it somewhere so the user can see their previous guesses.
5. Next, check whether it is the correct number.
6. If it is correct:
    i. Display congratulations message.
    ii. Stop the player from being able to enter more guesses (this would mess the game up).
    iii. Display control allowing the player to restart the game.
7. If it is wrong and the player has turns left:
    i. Tell the player they are wrong and whether their guess was too high or too low.
    ii. Allow them to enter another guess.
    iii. Increment the turn number by 1.
8. If it is wrong and the player has no turns left:
    i. Tell the player it is game over.
    ii. Stop the player from being able to enter more guesses (this would mess the game up).
    iii. Display control allowing the player to restart the game.
9. Once the game restarts, make sure the game logic and UI are completely reset, then go back to step 1.

Let's now move forward, looking at how we can turn these steps into code, building up the example, and exploring JavaScript features as we go.

Initial setup

To begin this tutorial, we'd like you to make a local copy of the number-guessing-game-start.html file (see it live here). Open it in both your text editor and your web browser. At the moment you'll see a simple heading, paragraph of instructions and form for entering a guess, but the form won't currently do anything.

The place where we'll be adding all our code is inside the `<script>` element at the bottom of the HTML:

HTML

```
<script>
  // Your JavaScript goes here
</script>
```

Adding variables to store our data

Let's get started. First of all, add the following lines inside your `<script>` element:

JS

```
let randomNumber = Math.floor(Math.random() * 100) + 1;

const guesses = document.querySelector(".guesses");
const lastResult = document.querySelector(".lastResult");
const lowOrHi = document.querySelector(".lowOrHi");

const guessSubmit = document.querySelector(".guessSubmit");
const guessField = document.querySelector(".guessField");

let guessCount = 1;
let resetButton;
```

This section of the code sets up the variables and constants we need to store the data our program will use.

Variables are basically names for values (such as numbers, or strings of text). You create a variable with the keyword `let` followed by a name for your variable.

Constants are also used to name values, but unlike variables, you can't change the value once set. In this case, we are using constants to store references to parts of our user interface. The text inside some of these elements might change, but each constant always references the same HTML element that it was initialized with. You create a constant with the keyword `const` followed by a name for the constant.

You can assign a value to your variable or constant with an equals sign (`=`) followed by the value you want to give it.

In our example:

- The first variable — `randomNumber` — is assigned a random number between 1 and 100, calculated using a mathematical algorithm.
- The first three constants are each made to store a reference to the results paragraphs in our HTML, and are used to insert values into the paragraphs later on in the code (note how they are inside a `<div>` element, which is itself used to select all three later on for resetting, when we restart the game):

  HTML

```
<div class="resultParas">
  <p class="guesses"></p>
  <p class="lastResult"></p>
  <p class="lowOrHi"></p>
</div>
```

- The next two constants store references to the form text input and submit button and are used to control submitting the guess later on.

```HTML
<label for="guessField">Enter a guess: </label>
<input type="number" id="guessField" class="guessField" />
<input type="submit" value="Submit guess" class="guessSubmit" />
```

- Our final two variables store a guess count of 1 (used to keep track of how many guesses the player has had), and a reference to a reset button that doesn't exist yet (but will later).

> **Note:** You'll learn a lot more about variables and constants later on in the course, starting with the article Storing the information you need — Variables.

Functions

Next, add the following below your previous JavaScript:

```JS
function checkGuess() {
  alert("I am a placeholder");
}
```

Functions are reusable blocks of code that you can write once and run again and again, saving the need to keep repeating code all the time. This is really useful. There are a number of ways to define functions, but for now we'll concentrate on one simple type. Here we have defined a function by using the keyword `function`, followed by a name, with parentheses put after it. After that, we put two curly braces (`{ }`). Inside the curly braces goes all the code that we want to run whenever we call the function.

When we want to run the code, we type the name of the function followed by the parentheses.

Let's try that now. Save your code and refresh the page in your browser. Then go into the developer tools JavaScript console, and enter the following line:

```JS
checkGuess();
```

After pressing `Return`/`Enter`, you should see an alert come up that says `I am a placeholder`; we have defined a function in our code that creates an alert whenever we call it.

> **Note:** You'll learn a lot more about functions later on in the article Functions — reusable blocks of code.

Operators

JavaScript operators allow us to perform tests, do math, join strings together, and other such things.

If you haven't already done so, save your code, refresh the page in your browser, and open the developer tools JavaScript console. Then we can try typing in the examples shown below — type in each one from the "Example" columns exactly as shown, pressing

`Return`/`Enter` after each one, and see what results they return.

First let's look at arithmetic operators, for example:

Operator	Name	Example
`+`	Addition	`6 + 9`
`-`	Subtraction	`20 - 15`
`*`	Multiplication	`3 * 7`
`/`	Division	`10 / 5`

There are also some shortcut operators available, called compound assignment operators. For example, if you want to add a new number to an existing one and return the result, you could do this:

```
JS
let number1 = 1;
number1 += 2;
```

This is equivalent to

```
JS
let number2 = 1;
number2 = number2 + 2;
```

When we are running true/false tests (for example inside conditionals — see below) we use comparison operators. For example:

Operator	Name	Example

`===`	Strict equality (is it exactly the same?)	`JS`
		`5 === 2 + 4 // false`
		`'Chris' === 'Bob' // false`
		`5 === 2 + 3 // true`
		`2 === '2' // false; number versus string`

`!==`	Non-equality (is it not the same?)	`JS`
		`5 !== 2 + 4 // true`
		`'Chris' !== 'Bob' // true`
		`5 !== 2 + 3 // false`
		`2 !== '2' // true; number versus string`

`<`	Less than	`JS`
		`6 < 10 // true`
		`20 < 10 // false`

`>`	Greater than	`JS`
		`6 > 10 // false`
		`20 > 10 // true`

Text strings

Strings are used for representing text. We've already seen a string variable: in the following code, `"I am a placeholder"` is a string:

```
JS
function checkGuess() {
  alert("I am a placeholder");
}
```

You can declare strings using double quotes (`"`) or single quotes (`'`), but you must use the same form for the start and end of a single string declaration: you can't write `"I am a placeholder'`.

You can also declare strings using backticks (`` ` ``). Strings declared like this are called *template literals* and have some special properties. In particular, you can embed other variables or even expressions in them:

```
JS
const name = "Mahalia";

const greeting = `Hello ${name}`;
```

This gives you a mechanism to join strings together.

Conditionals

Returning to our `checkGuess()` function, I think it's safe to say that we don't want it to just spit out a placeholder message. We want it to check whether a player's guess is correct or not, and respond appropriately.

At this point, replace your current `checkGuess()` function with this version instead:

```
JS
function checkGuess() {
  const userGuess = Number(guessField.value);
  if (guessCount === 1) {
    guesses.textContent = "Previous guesses:";
  }
  guesses.textContent = `${guesses.textContent} ${userGuess}`;

  if (userGuess === randomNumber) {
    lastResult.textContent = "Congratulations! You got it right!";
    lastResult.style.backgroundColor = "green";
    lowOrHi.textContent = "";
    setGameOver();
  } else if (guessCount === 10) {
    lastResult.textContent = "!!!GAME OVER!!!";
    lowOrHi.textContent = "";
    setGameOver();
  } else {
    lastResult.textContent = "Wrong!";
    lastResult.style.backgroundColor = "red";
    if (userGuess < randomNumber) {
      lowOrHi.textContent = "Last guess was too low!";
    } else if (userGuess > randomNumber) {
      lowOrHi.textContent = "Last guess was too high!";
    }
  }

  guessCount++;
  guessField.value = "";
```

```
  guessField.focus();
}
```

This is a lot of code — phew! Let's go through each section and explain what it does.

- The first line declares a variable called `userGuess` and sets its value to the current value entered inside the text field. We also run this value through the built-in `Number()` constructor, just to make sure the value is definitely a number. Since we're not changing this variable, we'll declare it using `const`.
- Next, we encounter our first conditional code block. A conditional code block allows you to run code selectively, depending on whether a certain condition is true or not. It looks a bit like a function, but it isn't. The simplest form of conditional block starts with the keyword `if`, then some parentheses, then some curly braces. Inside the parentheses, we include a test. If the test returns `true`, we run the code inside the curly braces. If not, we don't, and move on to the next bit of code. In this case, the test is testing whether the `guessCount` variable is equal to `1` (i.e. whether this is the player's first go or not):

  ```
  JS
  guessCount === 1;
  ```

  If it is, we make the guesses paragraph's text content equal to `Previous guesses:`. If not, we don't.
- Next, we use a template literal to append the current `userGuess` value onto the end of the `guesses` paragraph, with a blank space in between.
- The next block does a few checks:
  - The first `if (){ }` checks whether the user's guess is equal to the `randomNumber` set at the top of our JavaScript. If it is, the player has guessed correctly and the game is won, so we show the player a congratulations message with a nice green color, clear the contents of the Low/High guess information box, and run a function called `setGameOver()`, which we'll discuss later.
  - Now we've chained another test onto the end of the last one using an `else if (){ }` structure. This one checks whether this turn is the user's last turn. If it is, the program does the same thing as in the previous block, except with a game over message instead of a congratulations message.
  - The final block chained onto the end of this code (the `else { }`) contains code that is only run if neither of the other two tests returns true (i.e. the player didn't guess right, but they have more guesses left). In this case we tell them they are wrong, then we perform another conditional test to check whether the guess was higher or lower than the answer, displaying a further message as appropriate to tell them higher or lower.
- The last three lines in the function (lines 26–28 above) get us ready for the next guess to be submitted. We add 1 to the `guessCount` variable so the player uses up their turn (`++` is an incrementation operation — increment by 1), and empty the value out of the form text field and focus it again, ready for the next guess to be entered.

Events

At this point, we have a nicely implemented `checkGuess()` function, but it won't do anything because we haven't called it yet. Ideally, we want to call it when the "Submit guess" button is pressed, and to do this we need to use an **event**. Events are things that happen in the browser — a button being clicked, a page loading, a video playing, etc. — in response to which we can run blocks of code. **Event listeners** observe specific events and call **event handlers**, which are blocks of code that run in response to an event firing.

Add the following line below your `checkGuess()` function:

```
  JS
guessSubmit.addEventListener("click", checkGuess);
```

Here we are adding an event listener to the `guessSubmit` button. This is a method that takes two input values (called *arguments*) — the type of event we are listening out for (in this case `click`) as a string, and the code we want to run when the event occurs (in

this case the `checkGuess()` function). Note that we don't need to specify the parentheses when writing it inside `addEventListener()`.

Try saving and refreshing your code now, and your example should work — to a point. The only problem now is that if you guess the correct answer or run out of guesses, the game will break because we've not yet defined the `setGameOver()` function that is supposed to be run once the game is over. Let's add our missing code now and complete the example functionality.

Finishing the game functionality

Let's add that `setGameOver()` function to the bottom of our code and then walk through it. Add this now, below the rest of your JavaScript:

```
JS
function setGameOver() {
  guessField.disabled = true;
  guessSubmit.disabled = true;
  resetButton = document.createElement("button");
  resetButton.textContent = "Start new game";
  document.body.append(resetButton);
  resetButton.addEventListener("click", resetGame);
}
```

- The first two lines disable the form text input and button by setting their disabled properties to `true`. This is necessary, because if we didn't, the user could submit more guesses after the game is over, which would mess things up.
- The next three lines generate a new `<button>` element, set its text label to "Start new game", and add it to the bottom of our existing HTML.
- The final line sets an event listener on our new button so that when it is clicked, a function called `resetGame()` is run.

Now we need to define this function too! Add the following code, again to the bottom of your JavaScript:

```
JS
function resetGame() {
  guessCount = 1;

  const resetParas = document.querySelectorAll(".resultParas p");
  for (const resetPara of resetParas) {
    resetPara.textContent = "";
  }

  resetButton.parentNode.removeChild(resetButton);

  guessField.disabled = false;
  guessSubmit.disabled = false;
  guessField.value = "";
  guessField.focus();

  lastResult.style.backgroundColor = "white";

  randomNumber = Math.floor(Math.random() * 100) + 1;
}
```

This rather long block of code completely resets everything to how it was at the start of the game, so the player can have another go. It:

- Puts the `guessCount` back down to 1.

- Empties all the text out of the information paragraphs. We select all paragraphs inside `<div class="resultParas"></div>`, then loop through each one, setting their `textContent` to `''` (an empty string).
- Removes the reset button from our code.
- Enables the form elements, and empties and focuses the text field, ready for a new guess to be entered.
- Removes the background color from the `lastResult` paragraph.
- Generates a new random number so that you are not just guessing the same number again!

At this point, you should have a fully working (simple) game — congratulations!

All we have left to do now in this article is to talk about a few other important code features that you've already seen, although you may have not realized it.

Loops

One part of the above code that we need to take a more detailed look at is the for...of loop. Loops are a very important concept in programming, which allow you to keep running a piece of code over and over again, until a certain condition is met.

To start with, go to your browser developer tools JavaScript console again, and enter the following:

```
JS
const fruits = ["apples", "bananas", "cherries"];
for (const fruit of fruits) {
  console.log(fruit);
}
```

What happened? The strings `'apples', 'bananas', 'cherries'` were printed out in your console.

This is because of the loop. The line `const fruits = ['apples', 'bananas', 'cherries'];` creates an array. We will work through a complete Arrays guide later in this module, but for now: an array is a collection of items (in this case strings).

A `for...of` loop gives you a way to get each item in the array and run some JavaScript on it. The line `for (const fruit of fruits)` says:

1. Get the first item in `fruits`.
2. Set the `fruit` variable to that item, then run the code between the `{}` curly braces.
3. Get the next item in `fruits`, and repeat 2, until you reach the end of `fruits`.

In this case, the code inside the curly braces is writing out `fruit` to the console.

Now let's look at the loop in our number guessing game — the following can be found inside the `resetGame()` function:

```
JS
const resetParas = document.querySelectorAll(".resultParas p");
for (const resetPara of resetParas) {
  resetPara.textContent = "";
}
```

This code creates a variable containing a list of all the paragraphs inside `<div class="resultParas">` using the `querySelectorAll()` method, then it loops through each one, removing the text content of each.

Note that even though `resetPara` is a constant, we can change its internal properties like `textContent`.

A small discussion on objects

Let's add one more final improvement before we get to this discussion. Add the following line just below the `let resetButton;` line near the top of your JavaScript, then save your file:

```
JS
guessField.focus();
```

This line uses the `focus()` method to automatically put the text cursor into the `<input>` text field as soon as the page loads, meaning that the user can start typing their first guess right away, without having to click the form field first. It's only a small addition, but it improves usability — giving the user a good visual clue as to what they've got to do to play the game.

Let's analyze what's going on here in a bit more detail. In JavaScript, most of the items you will manipulate in your code are objects. An object is a collection of related functionality stored in a single grouping. You can create your own objects, but that is quite advanced and we won't be covering it until much later in the course. For now, we'll just briefly discuss the built-in objects that your browser contains, which allow you to do lots of useful things.

In this particular case, we first created a `guessField` constant that stores a reference to the text input form field in our HTML — the following line can be found amongst our declarations near the top of the code:

```
JS
const guessField = document.querySelector(".guessField");
```

To get this reference, we used the `querySelector()` method of the `document` object. `querySelector()` takes one piece of information — a CSS selector that selects the element you want a reference to.

Because `guessField` now contains a reference to an `<input>` element, it now has access to a number of properties (basically variables stored inside objects, some of which can't have their values changed) and methods (basically functions stored inside objects). One method available to input elements is `focus()`, so we can now use this line to focus the text input:

```
JS
guessField.focus();
```

Variables that don't contain references to form elements won't have `focus()` available to them. For example, the `guesses` constant contains a reference to a `<p>` element, and the `guessCount` variable contains a number.

Playing with browser objects

Let's play with some browser objects a bit.

1. First of all, open up your program in a browser.
2. Next, open your browser developer tools, and make sure the JavaScript console tab is open.
3. Type `guessField` into the console and the console shows you that the variable contains an `<input>` element. You'll also notice that the console autocompletes the names of objects that exist inside the execution environment, including your variables!
4. Now type in the following:

   ```
   JS
   guessField.value = 2;
   ```

   The `value` property represents the current value entered into the text field. You'll see that by entering this command, we've changed the text in the text field!

5. Now try typing `guesses` into the console and pressing `Enter` (or `Return`, depending on your keyboard). The console shows you that the variable contains a `<p>` element.
6. Now try entering the following line:

   JS
   ```
   guesses.value;
   ```

   The browser returns `undefined`, because paragraphs don't have the `value` property.
7. To change the text inside a paragraph, you need the `textContent` property instead. Try this:

   JS
   ```
   guesses.textContent = "Where is my paragraph?";
   ```

8. Now for some fun stuff. Try entering the below lines, one by one:

   JS
   ```
   guesses.style.backgroundColor = "yellow";
   guesses.style.fontSize = "200%";
   guesses.style.padding = "10px";
   guesses.style.boxShadow = "3px 3px 6px black";
   ```

   Every element on a page has a `style` property, which itself contains an object whose properties contain all the inline CSS styles applied to that element. This allows us to dynamically set new CSS styles on elements using JavaScript.

Finished for now...

So that's it for building the example. You got to the end — well done! Try your final code out, or play with our finished version here. If you can't get the example to work, check it against the source code.

What went wrong? Troubleshooting JavaScript

When you built up the "Guess the number" game in the previous article, you may have found that it didn't work. Never fear — this article aims to save you from tearing your hair out over such problems by providing you with some tips on how to find and fix errors in JavaScript programs.

Prerequisites:	A basic understanding of HTML and CSS, an understanding of what JavaScript is.
Objective:	To gain the ability and confidence to start fixing problems in your own code.

Types of error

Generally speaking, when you do something wrong in code, there are two main types of error that you'll come across:

- **Syntax errors**: These are spelling errors in your code that actually cause the program not to run at all, or stop working part way through — you will usually be provided with some error messages too. These are usually okay to fix, as long as you are familiar with the right tools and know what the error messages mean!
- **Logic errors**: These are errors where the syntax is actually correct but the code is not what you intended it to be, meaning that program runs successfully but gives incorrect results. These are often harder to fix than syntax errors, as there usually isn't an error message to direct you to the source of the error.

Okay, so it's not quite *that* simple — there are some other differentiators as you drill down deeper. But the above classifications will do at this early stage in your career. We'll look at both of these types going forward.

An erroneous example

To get started, let's return to our number guessing game — except this time we'll be exploring a version that has some deliberate errors introduced. Go to GitHub and make yourself a local copy of number-game-errors.html (see it running live here).

1. To get started, open the local copy inside your favorite text editor, and your browser.
2. Try playing the game — you'll notice that when you press the "Submit guess" button, it doesn't work!

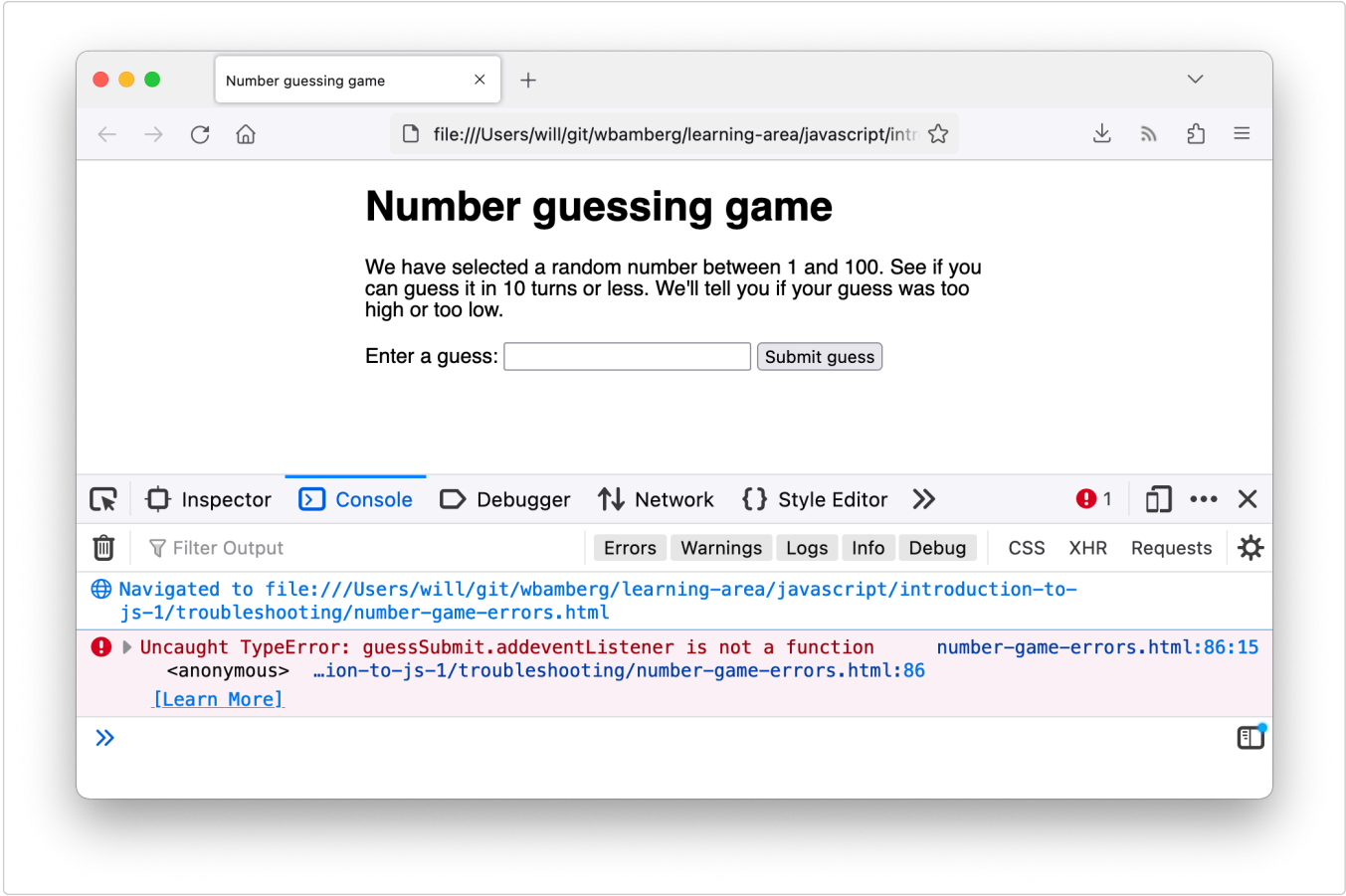
> **Note:** You might well have your own version of the game example that doesn't work, which you might want to fix! We'd still like you to work through the article with our version, so that you can learn the techniques we are teaching here. Then you can go back and try to fix your example.

At this point, let's consult the developer console to see if it reports any syntax errors, then try to fix them. You'll learn how below.

Fixing syntax errors

Earlier on in the course we got you to type some simple JavaScript commands into the developer tools JavaScript console (if you can't remember how to open this in your browser, follow the previous link to find out how). What's even more useful is that the console gives you error messages whenever a syntax error exists inside the JavaScript being fed into the browser's JavaScript engine. Now let's go hunting.

1. Go to the tab that you've got `number-game-errors.html` open in, and open your JavaScript console. You should see an error message along the following lines:



2. The first line of the error message is:

```
Uncaught TypeError: guessSubmit.addeventListener is not a function
number-game-errors.html:86:15
```

   - The first part, `Uncaught TypeError: guessSubmit.addeventListener is not a function`, is telling us something about what went wrong.
   - The second part, `number-game-errors.html:86:15`, is telling us where in the code the error came from: line 86, character 15 of the file "number-game-errors.html".

3. If we look at line 86 in our code editor, we'll find this line:

> **Warning:** Error message may not be on line 86.
>
> If you are using any code editor with an extension that launches a live server on your local machine, this will cause extra code to be injected. Because of this, the developer tools will list the error as occurring on a line that is not 86.

JS

```
guessSubmit.addeventListener("click", checkGuess);
```

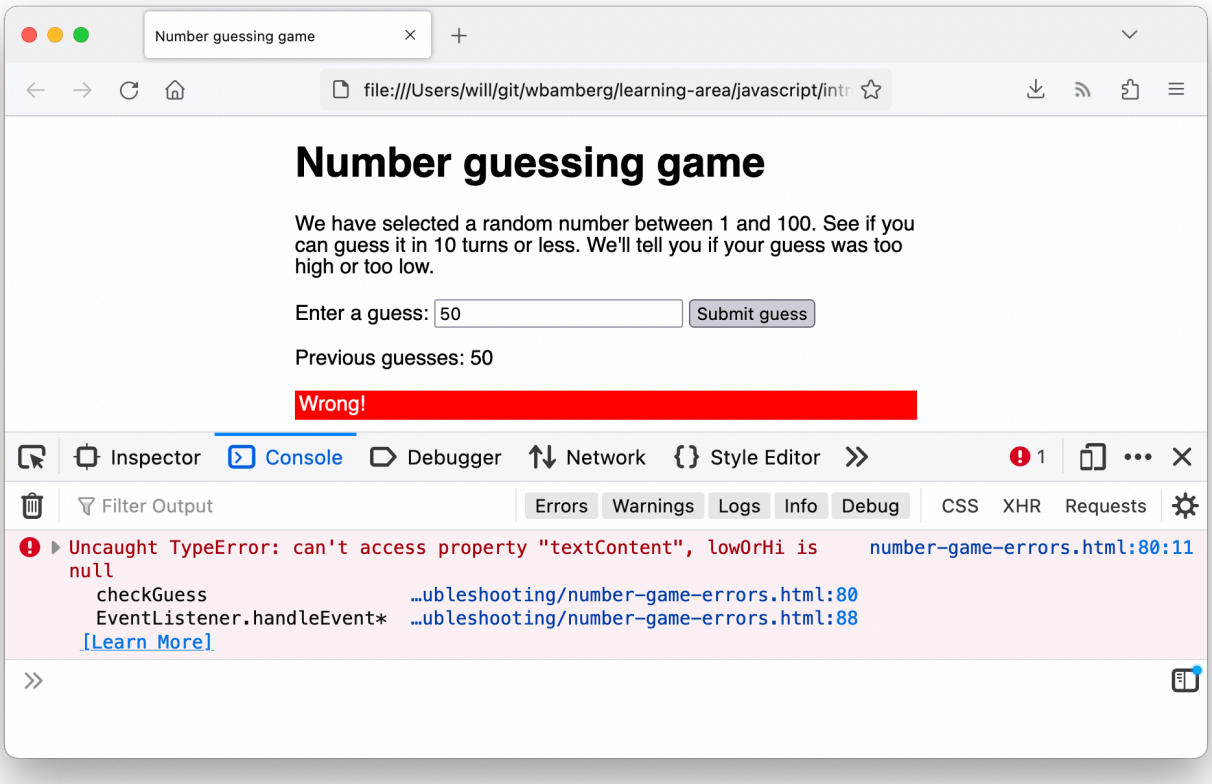
4. The error message says "guessSubmit.addeventListener is not a function", which means that the function we're calling is not recognized by the JavaScript interpreter. Often, this error message actually means that we've spelled something wrong. If you are not sure of the correct spelling of a piece of syntax, it is often good to look up the feature on MDN. The best way to do this currently is to search for "mdn *name-of-feature*" with your favorite search engine. Here's a shortcut to save you some time in this instance: `addEventListener()`.

5. So, looking at this page, the error appears to be that we've spelled the function name wrong! Remember that JavaScript is case-sensitive, so any slight difference in spelling or casing will cause an error. Changing `addeventListener` to `addEventListener` should fix this. Do this now.

> **Note:** See our TypeError: "x" is not a function reference page for more details about this error.

Syntax errors round two

1. Save your page and refresh, and you should see the error has gone.
2. Now if you try to enter a guess and press the Submit guess button, you'll see another error!



3. This time the error being reported is:

```
Uncaught TypeError: can't access property "textContent", lowOrHi is null
```

Depending on the browser you are using, you might see a different message here. The message above is what Firefox will show you, but Chrome, for example, will show you this:

```
Uncaught TypeError: Cannot set properties of null (setting 'textContent')
```

It's the same error, but different browsers describe it in a different way.

> **Note:** This error didn't come up as soon as the page was loaded because this error occurred inside a function (inside the `checkGuess() { }` block). As you'll learn in more detail in our later functions article, code inside functions runs in a separate scope than code outside functions. In this case, the code was not run and the error was not thrown until the `checkGuess()` function was run by line 86.

4. The line number given in the error is 80. Have a look at line 80, and you'll see the following code:

   JS

```
lowOrHi.textContent = "Last guess was too high!";
```

5. This line is trying to set the `textContent` property of the `lowOrHi` variable to a text string, but it's not working because `lowOrHi` does not contain what it's supposed to. Let's see why this is — try searching for other instances of `lowOrHi` in the code. The earliest instance you'll find is on line 49:

   JS

```
const lowOrHi = document.querySelector("lowOrHi");
```

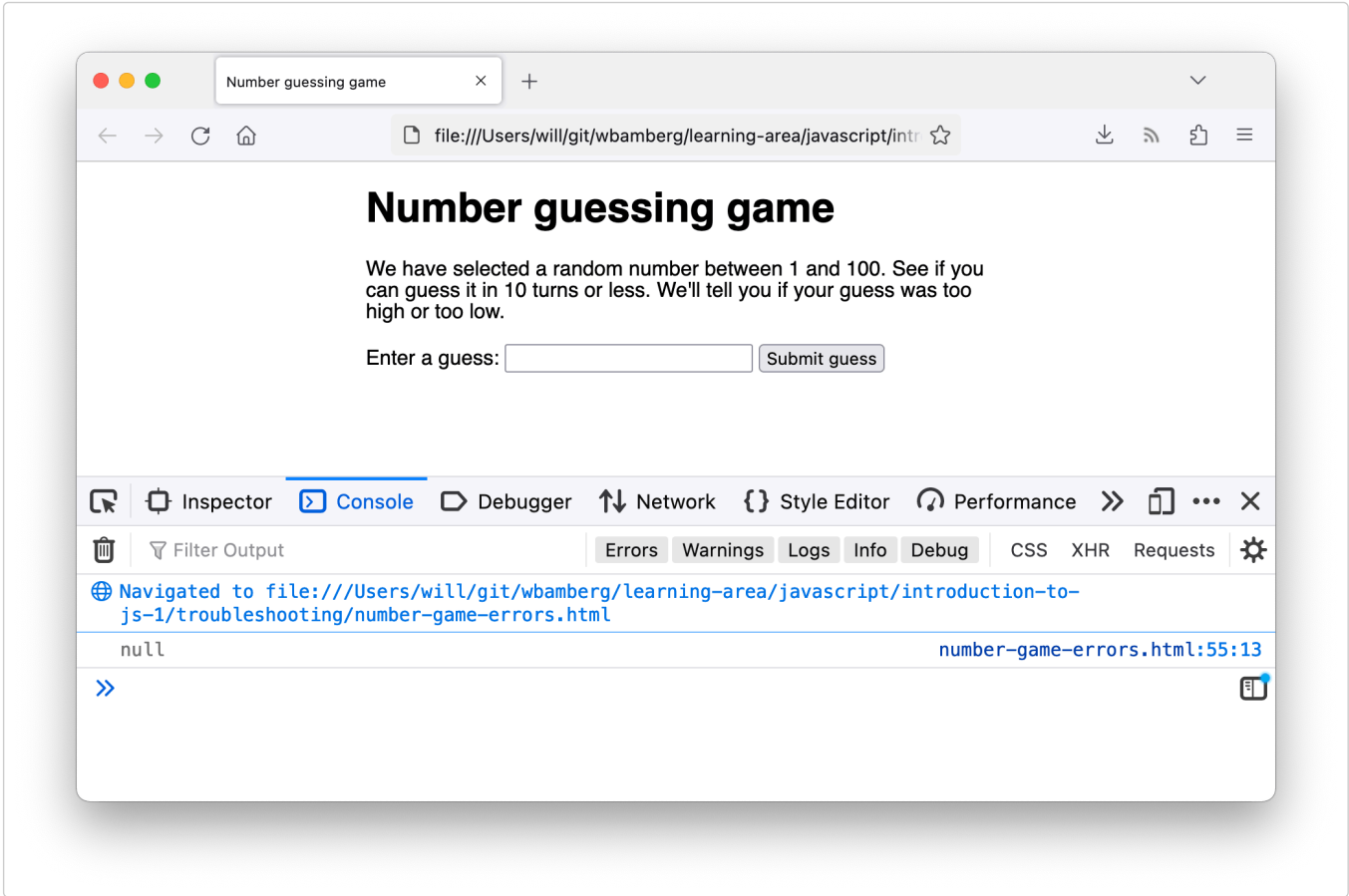
6. At this point we are trying to make the variable contain a reference to an element in the document's HTML. Let's see what the variable contains after this line has been run. Add the following code on line 50:

   JS

```
console.log(lowOrHi);
```

   This code will print the value of `lowOrHi` to the console after we tried to set it in line 49. See `console.log()` for more information.

7. Save and refresh, and you should now see the `console.log()` result in your console.



   Sure enough, `lowOrHi` 's value is `null` at this point, and this matches up with the Firefox error message `lowOrHi is null`. So there is definitely a problem with line 49. The `null` value means "nothing", or "no value". So our code to set `lowOrHi` to an element is going wrong.

8. Let's think about what the problem could be. Line 49 is using a `document.querySelector()` method to get a reference to an element by selecting it with a CSS selector. Looking further up our file, we can find the paragraph in question:

   HTML

```
<p class="lowOrHi"></p>
```

9. So we need a class selector here, which begins with a dot (.), but the selector being passed into the `querySelector()` method in line 49 has no dot. This could be the problem! Try changing `lowOrHi` to `.lowOrHi` in line 49.
10. Try saving and refreshing again, and your `console.log()` statement should return the `<p>` element we want. Phew! Another error fixed! You can delete your `console.log()` line now, or keep it to reference later on — your choice.

> **Note:** See our TypeError: "x" is (not) "y" reference page for more details about this error.

Syntax errors round three

1. Now if you try playing the game through again, you should get more success — the game should play through absolutely fine, until you end the game, either by guessing the right number, or by running out of guesses.
2. At that point, the game fails again, and the same error is spat out that we got at the beginning — "TypeError: resetButton.addeventListener is not a function"! However, this time it's listed as coming from line 94.
3. Looking at line number 94, it is easy to see that we've made the same mistake here. We again just need to change `addeventListener` to `addEventListener`. Do this now.

A logic error

At this point, the game should play through fine, however after playing through a few times you'll undoubtedly notice that the game always chooses 1 as the "random" number you've got to guess. Definitely not quite how we want the game to play out!

There's definitely a problem in the game logic somewhere — the game is not returning an error; it just isn't playing right.

1. Search for the `randomNumber` variable, and the lines where the random number is first set. The instance that stores the random number that we want to guess at the start of the game should be around line number 45:

```
JS
let randomNumber = Math.floor(Math.random()) + 1;
```

2. And the one that generates the random number before each subsequent game is around line 113:

```
JS
randomNumber = Math.floor(Math.random()) + 1;
```

3. To check whether these lines are indeed the problem, let's turn to our friend `console.log()` again — insert the following line directly below each of the above two lines:

```
JS
console.log(randomNumber);
```

4. Save and refresh, then play a few games — you'll see that `randomNumber` is equal to 1 at each point where it is logged to the console.

Working through the logic

To fix this, let's consider how this line is working. First, we invoke `Math.random()`, which generates a random decimal number between 0 and 1, e.g. 0.5675493843.

```
JS
Math.random();
```

Next, we pass the result of invoking `Math.random()` through `Math.floor()`, which rounds the number passed to it down to the nearest whole number. We then add 1 to that result:

JS

```
Math.floor(Math.random()) + 1;
```

Rounding a random decimal number between 0 and 1 down will always return 0, so adding 1 to it will always return 1. We need to multiply the random number by 100 before we round it down. The following would give us a random number between 0 and 99:

JS

```
Math.floor(Math.random() * 100);
```

Hence us wanting to add 1, to give us a random number between 1 and 100:

JS

```
Math.floor(Math.random() * 100) + 1;
```

Try updating both lines like this, then save and refresh — the game should now play like we are intending it to!

Other common errors

There are other common errors you'll come across in your code. This section highlights most of them.

SyntaxError: missing ; before statement

This error generally means that you have missed a semicolon at the end of one of your lines of code, but it can sometimes be more cryptic. For example, if we change this line inside the `checkGuess()` function:

JS

```
const userGuess = Number(guessField.value);
```

to

JS

```
  const userGuess === Number(guessField.value);
```

It throws this error because it thinks you are trying to do something different. You should make sure that you don't mix up the assignment operator (`=`) — which sets a variable to be equal to a value — with the strict equality operator (`===`), which tests whether one value is equal to another, and returns a `true` / `false` result.

> **Note:** See our SyntaxError: missing ; before statement reference page for more details about this error.

The program always says you've won, regardless of the guess you enter

This could be another symptom of mixing up the assignment and strict equality operators. For example, if we were to change this line inside `checkGuess()` :

JS

```
if (userGuess === randomNumber) {
```

to

JS

```
if (userGuess = randomNumber) {
```

the test would always return `true`, causing the program to report that the game has been won. Be careful!

SyntaxError: missing) after argument list

This one is pretty simple — it generally means that you've missed the closing parenthesis at the end of a function/method call.

> **Note:** See our SyntaxError: missing) after argument list reference page for more details about this error.

SyntaxError: missing : after property id

This error usually relates to an incorrectly formed JavaScript object, but in this case we managed to get it by changing

JS

```
function checkGuess() {
```

to

JS

```
function checkGuess( {
```

This has caused the browser to think that we are trying to pass the contents of the function into the function as an argument. Be careful with those parentheses!

SyntaxError: missing } after function body

This is easy — it generally means that you've missed one of your curly braces from a function or conditional structure. We got this error by deleting one of the closing curly braces near the bottom of the `checkGuess()` function.

SyntaxError: expected expression, got '*string*' or SyntaxError: unterminated string literal

These errors generally mean that you've left off a string value's opening or closing quote mark. In the first error above, *string* would be replaced with the unexpected character(s) that the browser found instead of a quote mark at the start of a string. The second error means that the string has not been ended with a quote mark.

For all of these errors, think about how we tackled the examples we looked at in the walkthrough. When an error arises, look at the line number you are given, go to that line and see if you can spot what's wrong. Bear in mind that the error is not necessarily going to be on that line, and also that the error might not be caused by the exact same problem we cited above!

> **Note:** See our SyntaxError: Unexpected token and SyntaxError: unterminated string literal reference pages for more details about these errors.

Summary

So there we have it, the basics of figuring out errors in simple JavaScript programs. It won't always be that simple to work out what's wrong in your code, but at least this will save you a few hours of sleep and allow you to progress a bit faster when things don't turn out right, especially in the earlier stages of your learning journey.

See also

- There are many other types of errors that aren't listed here; we are compiling a reference that explains what they mean in detail — see the JavaScript error reference.
- If you come across any errors in your code that you aren't sure how to fix after reading this article, you can get help! Ask for help on the communication channels. Tell us what your error is, and we'll try to help you. A listing of your code would be useful as well.

Storing the information you need — Variables

After reading the last couple of articles you should now know what JavaScript is, what it can do for you, how you use it alongside other web technologies, and what its main features look like from a high level. In this article, we will get down to the real basics, looking at how to work with the most basic building blocks of JavaScript — Variables.

Prerequisites:	A basic understanding of HTML and CSS, an understanding of what JavaScript is.
Objective:	To gain familiarity with the basics of JavaScript variables.

Tools you need

Throughout this article, you'll be asked to type in lines of code to test your understanding of the content. If you are using a desktop browser, the best place to type your sample code is your browser's JavaScript console (see What are browser developer tools for more information on how to access this tool).

What is a variable?

A variable is a container for a value, like a number we might use in a sum, or a string that we might use as part of a sentence.

Variable example

Let's look at a simple example:

HTML Play

```
<button id="button_A">Press me</button>
<h3 id="heading_A"></h3>
```

JS Play

```
const buttonA = document.querySelector("#button_A");
const headingA = document.querySelector("#heading_A");

buttonA.onclick = () => {
  const name = prompt("What is your name?");
  alert(`Hello ${name}, nice to see you!`);
  headingA.textContent = `Welcome ${name}`;
};
```

Play

In this example pressing the button runs some code. The first line pops a box up on the screen that asks the reader to enter their name, and then stores the value in a variable. The second line displays a welcome message that includes their name, taken from the variable value and the third line displays that name on the page.

Without a variable

To understand why this is so useful, let's think about how we'd write this example without using a variable. It would end up looking something like this:

HTML Play

```
<button id="button_B">Press me</button>
<h3 id="heading_B"></h3>
```

JS Play

```
const buttonB = document.querySelector("#button_B");
const headingB = document.querySelector("#heading_B");

buttonB.onclick = () => {
  alert(`Hello ${prompt("What is your name?")}, nice to see you!`);
  headingB.textContent = `Welcome ${prompt("What is your name?")}`;
};
```

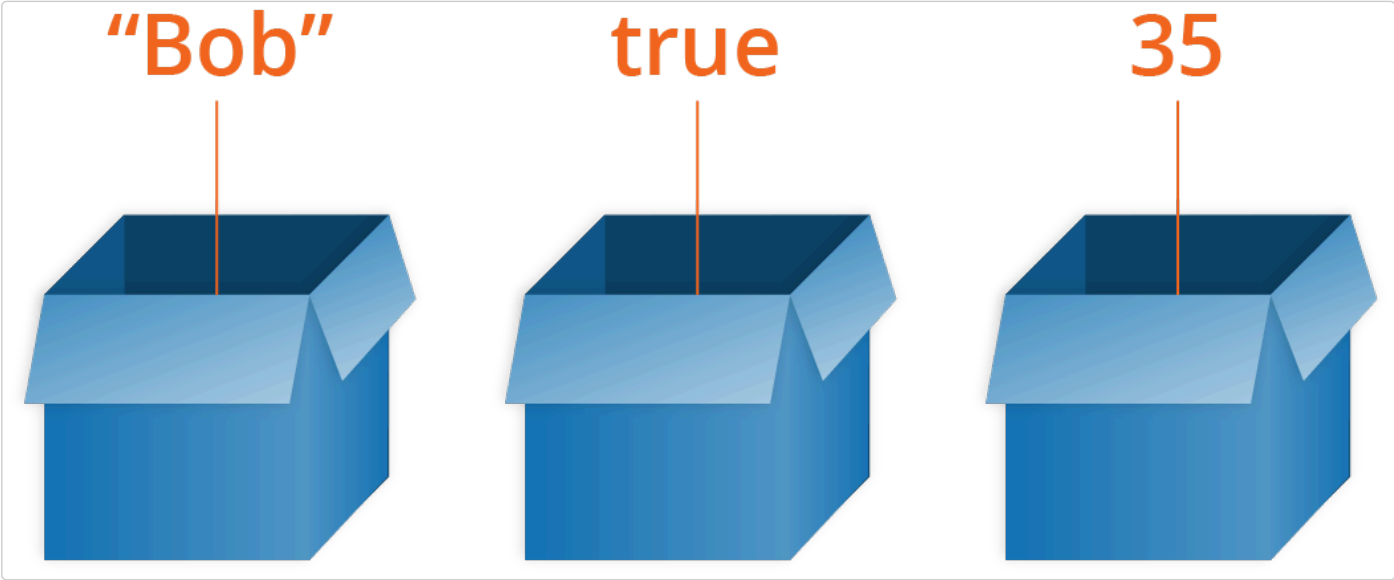
Play

You may not fully understand the syntax we are using (yet!), but you should be able to get the idea. If we didn't have variables available, we'd have to ask the reader for their name every time we needed to use it!

Variables just make sense, and as you learn more about JavaScript they will start to become second nature.

One special thing about variables is that they can contain just about anything — not just strings and numbers. Variables can also contain complex data and even entire functions to do amazing things. You'll learn more about this as you go along.

> **Note:** We say variables contain values. This is an important distinction to make. Variables aren't the values themselves; they are containers for values. You can think of them being like little cardboard boxes that you can store things in.



Declaring a variable

To use a variable, you've first got to create it — more accurately, we call this declaring the variable. To do this, we type the keyword `let` followed by the name you want to call your variable:

```
JS
let myName;
let myAge;
```

Here we're creating two variables called `myName` and `myAge`. Try typing these lines into your web browser's console. After that, try creating a variable (or two) with your own name choices.

> **Note:** In JavaScript, all code instructions should end with a semicolon (;) — your code may work correctly for single lines, but probably won't when you are writing multiple lines of code together. Try to get into the habit of including it.

You can test whether these values now exist in the execution environment by typing just the variable's name, e.g.

```
JS
myName;
myAge;
```

They currently have no value; they are empty containers. When you enter the variable names, you should get a value of `undefined` returned. If they don't exist, you'll get an error message — try typing in

```
JS
scoobyDoo;
```

> **Note:** Don't confuse a variable that exists but has no defined value with a variable that doesn't exist at all — they are very different things. In the box analogy you saw above, not existing would mean there's no box (variable) for a value to go in. No value defined would mean that there is a box, but it has no value inside it.

Initializing a variable

Once you've declared a variable, you can initialize it with a value. You do this by typing the variable name, followed by an equals sign (=), followed by the value you want to give it. For example:

```
JS
myName = "Chris";
myAge = 37;
```

Try going back to the console now and typing in these lines. You should see the value you've assigned to the variable returned in the console to confirm it, in each case. Again, you can return your variable values by typing their name into the console — try these again:

```
JS
myName;
myAge;
```

You can declare and initialize a variable at the same time, like this:

```
JS
let myDog = "Rover";
```

This is probably what you'll do most of the time, as it is quicker than doing the two actions on two separate lines.

A note about var

You'll probably also see a different way to declare variables, using the `var` keyword:

```
JS
var myName;
var myAge;
```

Back when JavaScript was first created, this was the only way to declare variables. The design of `var` is confusing and error-prone. So `let` was created in modern versions of JavaScript, a new keyword for creating variables that works somewhat differently to `var`, fixing its issues in the process.

A couple of simple differences are explained below. We won't go into all the differences now, but you'll start to discover them as you learn more about JavaScript (if you really want to read about them now, feel free to check out our let reference page).

For a start, if you write a multiline JavaScript program that declares and initializes a variable, you can actually declare a variable with `var` after you initialize it and it will still work. For example:

```
JS
myName = "Chris";

function logName() {
  console.log(myName);
}

logName();

var myName;
```

> **Note:** This won't work when typing individual lines into a JavaScript console, just when running multiple lines of JavaScript in a web document.

This works because of **hoisting** — read var hoisting for more detail on the subject.

Hoisting no longer works with `let`. If we changed `var` to `let` in the above example, it would fail with an error. This is a good thing — declaring a variable after you initialize it results in confusing, harder to understand code.

Secondly, when you use `var`, you can declare the same variable as many times as you like, but with `let` you can't. The following would work:

```
JS
var myName = "Chris";
var myName = "Bob";
```

But the following would throw an error on the second line:

```
JS
let myName = "Chris";
let myName = "Bob";
```

You'd have to do this instead:

```
JS
let myName = "Chris";
myName = "Bob";
```

Again, this is a sensible language decision. There is no reason to redeclare variables — it just makes things more confusing.

For these reasons and more, we recommend that you use `let` in your code, rather than `var`. Unless you are explicitly writing support for ancient browsers, there is no longer any reason to use `var` as all modern browsers have supported `let` since 2015.

> **Note:** If you are trying this code in your browser's console, prefer to copy & paste each of the code blocks here as a whole. There's a feature in Chrome's console where variable re-declarations with `let` and `const` are allowed:
>
> ```
> > let myName = "Chris";
>   let myName = "Bob";
> // As one input: SyntaxError: Identifier 'myName' has already been declared
>
> > let myName = "Chris";
> > let myName = "Bob";
> // As two inputs: both succeed
> ```

Updating a variable

Once a variable has been initialized with a value, you can change (or update) that value by giving it a different value. Try entering the following lines into your console:

```
JS
myName = "Bob";
myAge = 40;
```

An aside on variable naming rules

You can call a variable pretty much anything you like, but there are limitations. Generally, you should stick to just using Latin characters (0-9, a-z, A-Z) and the underscore character.

- You shouldn't use other characters because they may cause errors or be hard to understand for an international audience.
- Don't use underscores at the start of variable names — this is used in certain JavaScript constructs to mean specific things, so may get confusing.
- Don't use numbers at the start of variables. This isn't allowed and causes an error.
- A safe convention to stick to is lower camel case, where you stick together multiple words, using lower case for the whole first word and then capitalize subsequent words. We've been using this for our variable names in the article so far.
- Make variable names intuitive, so they describe the data they contain. Don't just use single letters/numbers, or big long phrases.
- Variables are case sensitive — so `myage` is a different variable from `myAge`.
- One last point: you also need to avoid using JavaScript reserved words as your variable names — by this, we mean the words that make up the actual syntax of JavaScript! So, you can't use words like `var`, `function`, `let`, and `for` as variable names. Browsers recognize them as different code items, and so you'll get errors.

> **Note:** You can find a fairly complete list of reserved keywords to avoid at Lexical grammar — keywords.

Good name examples:

```
age
myAge
init
initialColor
finalOutputValue
audio1
audio2
```

Bad name examples:

```
1
a
_12
myage
MYAGE
var
Document
skjfndskjfnbdskjfb
thisisareallylongvariablenameman
```

Try creating a few more variables now, with the above guidance in mind.

Variable types

There are a few different types of data we can store in variables. In this section we'll describe these in brief, then in future articles, you'll learn about them in more detail.

Numbers

You can store numbers in variables, either whole numbers like 30 (also called integers) or decimal numbers like 2.456 (also called floats or floating point numbers). You don't need to declare variable types in JavaScript, unlike some other programming languages. When you give a variable a number value, you don't include quotes:

JS

```
let myAge = 17;
```

Strings

Strings are pieces of text. When you give a variable a string value, you need to wrap it in single or double quote marks; otherwise, JavaScript tries to interpret it as another variable name.

JS

```
let dolphinGoodbye = "So long and thanks for all the fish";
```

Booleans

Booleans are true/false values — they can have two values, `true` or `false`. These are generally used to test a condition, after which code is run as appropriate. So for example, a simple case would be:

JS

```
let iAmAlive = true;
```

Whereas in reality it would be used more like this:

JS

```
let test = 6 < 3;
```

This is using the "less than" operator (`<`) to test whether 6 is less than 3. As you might expect, it returns `false`, because 6 is not less than 3! You will learn a lot more about such operators later on in the course.

Arrays

An array is a single object that contains multiple values enclosed in square brackets and separated by commas. Try entering the following lines into your console:

JS

```
let myNameArray = ["Chris", "Bob", "Jim"];
let myNumberArray = [10, 15, 40];
```

Once these arrays are defined, you can access each value by their location within the array. Try these lines:

JS

```
myNameArray[0]; // should return 'Chris'
myNumberArray[2]; // should return 40
```

The square brackets specify an index value corresponding to the position of the value you want returned. You might have noticed that arrays in JavaScript are zero-indexed: the first element is at index 0.

To learn more, see our article on Arrays.

Objects

In programming, an object is a structure of code that models a real-life object. You can have a simple object that represents a box and contains information about its width, length, and height, or you could have an object that represents a person, and contains data about their name, height, weight, what language they speak, how to say hello to them, and more.

Try entering the following line into your console:

```
JS
let dog = { name: "Spot", breed: "Dalmatian" };
```

To retrieve the information stored in the object, you can use the following syntax:

```
JS
dog.name;
```

For more on this topic, see the Introducing JavaScript objects module.

Dynamic typing

JavaScript is a "dynamically typed language", which means that, unlike some other languages, you don't need to specify what data type a variable will contain (numbers, strings, arrays, etc.).

For example, if you declare a variable and give it a value enclosed in quotes, the browser treats the variable as a string:

```
JS
let myString = "Hello";
```

Even if the value enclosed in quotes is just digits, it is still a string — not a number — so be careful:

```
JS
let myNumber = "500"; // oops, this is still a string
typeof myNumber;
myNumber = 500; // much better — now this is a number
typeof myNumber;
```

Try entering the four lines above into your console one by one, and see what the results are. You'll notice that we are using a special operator called `typeof` — this returns the data type of the variable you type after it. The first time it is called, it should return `string`, as at that point the `myNumber` variable contains a string, `'500'`. Have a look and see what it returns the second time you call it.

Constants in JavaScript

As well as variables, you can declare constants. These are like variables, except that:

- you must initialize them when you declare them
- you can't assign them a new value after you've initialized them.

For example, using `let` you can declare a variable without initializing it:

```
JS
let count;
```

If you try to do this using `const` you will see an error:

```
JS
const count;
```

Similarly, with `let` you can initialize a variable, and then assign it a new value (this is also called *reassigning* the variable):

```
JS
let count = 1;
count = 2;
```

If you try to do this using `const` you will see an error:

```
JS
const count = 1;
count = 2;
```

Note that although a constant in JavaScript must always name the same value, you can change the content of the value that it names. This isn't a useful distinction for simple types like numbers or booleans, but consider an object:

```
JS
const bird = { species: "Kestrel" };
console.log(bird.species); // "Kestrel"
```

You can update, add, or remove properties of an object declared using `const`, because even though the content of the object has changed, the constant is still pointing to the same object:

```
JS
bird.species = "Striated Caracara";
console.log(bird.species); // "Striated Caracara"
```

When to use const and when to use let

If you can't do as much with `const` as you can with `let`, why would you prefer to use it rather than `let`? In fact `const` is very useful. If you use `const` to name a value, it tells anyone looking at your code that this name will never be assigned to a different value. Any time they see this name, they will know what it refers to.

In this course, we adopt the following principle about when to use `let` and when to use `const`:

Use `const` when you can, and use `let` when you have to.

This means that if you can initialize a variable when you declare it, and don't need to reassign it later, make it a constant.

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: variables.

Summary

By now you should know a reasonable amount about JavaScript variables and how to create them. In the next article, we'll focus on numbers in more detail, looking at how to do basic math in JavaScript.

Yes No

Learn how to contribute.

This page was last modified on Mar 6, 2024 by MDN contributors.

Basic math in JavaScript — numbers and operators

At this point in the course, we discuss math in JavaScript — how we can use operators and other features to successfully manipulate numbers to do our bidding.

Prerequisites:	A basic understanding of HTML and CSS, an understanding of what JavaScript is.
Objective:	To gain familiarity with the basics of math in JavaScript.

Everybody loves math

Okay, maybe not. Some of us like math, some of us have hated math ever since we had to learn multiplication tables and long division in school, and some of us sit somewhere in between the two. But none of us can deny that math is a fundamental part of life that we can't get very far without. This is especially true when we are learning to program JavaScript (or any other language for that matter) — so much of what we do relies on processing numerical data, calculating new values, and so on, that you won't be surprised to learn that JavaScript has a full-featured set of math functions available.

This article discusses only the basic parts that you need to know now.

Types of numbers

In programming, even the humble decimal number system that we all know so well is more complicated than you might think. We use different terms to describe different types of decimal numbers, for example:

- **Integers** are floating-point numbers without a fraction. They can either be positive or negative, e.g. 10, 400, or -5.

  (meaning that they are accurate to a greater number of decimal places).

We even have different types of number systems! Decimal is base 10 (meaning it uses 0–9 in each column), but we also have things like:

- **Binary** — The lowest level language of computers; 0s and 1s.
- **Octal** — Base 8, uses 0–7 in each column.
- **Hexadecimal** — Base 16, uses 0–9 and then a–f in each column. You may have encountered these numbers before when setting colors in CSS.

Before you start to get worried about your brain melting, stop right there! For a start, we are just going to stick to decimal numbers throughout this course; you'll rarely come across a need to start thinking about other types, if ever.

The second bit of good news is that unlike some other programming languages, JavaScript only has one data type for numbers, both integers and decimals — you guessed it, `Number`. This means that whatever type of numbers you are dealing with in JavaScript, you handle them in exactly the same way.

> **Note:** Actually, JavaScript has a second number type, BigInt, used for very, very large integers. But for the purposes of

> this course, we'll just worry about `Number` values.

It's all numbers to me

Let's quickly play with some numbers to reacquaint ourselves with the basic syntax we need. Enter the commands listed below into your developer tools JavaScript console.

1. First of all, let's declare a couple of variables and initialize them with an integer and a float, respectively, then type the variable names back in to check that everything is in order:

   JS
   ```
   const myInt = 5;
   const myFloat = 6.667;
   myInt;
   myFloat;
   ```

2. Number values are typed in without quote marks — try declaring and initializing a couple more variables containing numbers before you move on.
3. Now let's check that both our original variables are of the same datatype. There is an operator called `typeof` in JavaScript that does this. Enter the below two lines as shown:

   JS
   ```
   typeof myInt;
   typeof myFloat;
   ```

   You should get `"number"` returned in both cases — this makes things a lot easier for us than if different numbers had different data types, and we had to deal with them in different ways. Phew!

Useful Number methods

The `Number` object, an instance of which represents all standard numbers you'll use in your JavaScript, has a number of useful methods available on it for you to manipulate numbers. We don't cover these in detail in this article because we wanted to keep it as a simple introduction and only cover the real basic essentials for now; however, once you've read through this module a couple of times it is worth going to the object reference pages and learning more about what's available.

For example, to round your number to a fixed number of decimal places, use the `toFixed()` method. Type the following lines into your browser's console :

JS
```
const lotsOfDecimal = 1.766584958675746364;
lotsOfDecimal;
const twoDecimalPlaces = lotsOfDecimal.toFixed(2);
twoDecimalPlaces;
```

Converting to number data types

Sometimes you might end up with a number that is stored as a string type, which makes it difficult to perform calculations with it. This most commonly happens when data is entered into a form input, and the input type is text. There is a way to solve this problem — passing the string value into the `Number()` constructor to return a number version of the same value.

For example, try typing these lines into your console:

JS
```
let myNumber = "74";
myNumber += 3;
```

You end up with the result 743, not 77, because `myNumber` is actually defined as a string. You can test this by typing in the following:

```
JS
typeof myNumber;
```

To fix the calculation, you can do this:

```
JS
let myNumber = "74";
myNumber = Number(myNumber) + 3;
```

The result is then 77, as initially expected.

Arithmetic operators

Arithmetic operators are used for performing mathematical calculations in JavaScript:

Operator	Name	Purpose	Example
`+`	Addition	Adds two numbers together.	`6 + 9`
`-`	Subtraction	Subtracts the right number from the left.	`20 - 15`
`*`	Multiplication	Multiplies two numbers together.	`3 * 7`
`/`	Division	Divides the left number by the right.	`10 / 5`
`%`	Remainder (sometimes called modulo)	Returns the remainder left over after you've divided the left number into a number of integer portions equal to the right number.	`8 % 3` (returns 2, as three goes into 8 twice, leaving 2 left over).
`**`	Exponent	Raises a `base` number to the `exponent` power, that is, the `base` number multiplied by itself, `exponent` times.	`5 ** 2` (returns `25`, which is the same as `5 * 5`).

> **Note:** You'll sometimes see numbers involved in arithmetic referred to as operands.

> **Note:** You may sometimes see exponents expressed using the older `Math.pow()` method, which works in a very similar way. For example, in `Math.pow(7, 3)`, `7` is the base and `3` is the exponent, so the result of the expression is `343`. `Math.pow(7, 3)` is equivalent to `7**3`.

We probably don't need to teach you how to do basic math, but we would like to test your understanding of the syntax involved. Try entering the examples below into your developer tools JavaScript console to familiarize yourself with the syntax.

1. First try entering some simple examples of your own, such as

   ```
   JS
   10 + 7;
   9 * 8;
   60 % 3;
   ```

2. You can also try declaring and initializing some numbers inside variables, and try using those in the sums — the variables will behave exactly like the values they hold for the purposes of the sum. For example:

```
JS
const num1 = 10;
const num2 = 50;
9 * num1;
num1 ** 3;
num2 / num1;
```

3. Last for this section, try entering some more complex expressions, such as:

```
JS
5 + 10 * 3;
(num2 % 9) * num1;
num2 + num1 / 8 + 2;
```

Parts of this last set of calculations might not give you quite the result you were expecting; the section below might well give the answer as to why.

Operator precedence

Let's look at the last example from above, assuming that `num2` holds the value 50 and `num1` holds the value 10 (as originally stated above):

```
JS
num2 + num1 / 8 + 2;
```

As a human being, you may read this as *"50 plus 10 equals 60"*, then *"8 plus 2 equals 10"*, and finally *"60 divided by 10 equals 6"*.

But the browser does *"10 divided by 8 equals 1.25"*, then *"50 plus 1.25 plus 2 equals 53.25"*.

This is because of **operator precedence** — some operators are applied before others when calculating the result of a calculation (referred to as an *expression*, in programming). Operator precedence in JavaScript is the same as is taught in math classes in school — multiply and divide are always done first, then add and subtract (the calculation is always evaluated from left to right).

If you want to override operator precedence, you can put parentheses around the parts that you want to be explicitly dealt with first. So to get a result of 6, we could do this:

```
JS
(num2 + num1) / (8 + 2);
```

Try it and see.

> **Note:** A full list of all JavaScript operators and their precedence can be found in Operator precedence.

Increment and decrement operators

Sometimes you'll want to repeatedly add or subtract one to or from a numeric variable value. This can be conveniently done using the increment (`++`) and decrement (`--`) operators. We used `++` in our "Guess the number" game back in our first splash into JavaScript article, when we added 1 to our `guessCount` variable to keep track of how many guesses the user has left after each turn.

```
JS
guessCount++;
```

Let's try playing with these in your console. For a start, note that you can't apply these directly to a number, which might seem strange, but we are assigning a variable a new updated value, not operating on the value itself. The following will return an error:

```
JS
3++;
```

So, you can only increment an existing variable. Try this:

```
JS
let num1 = 4;
num1++;
```

Okay, strangeness number 2! When you do this, you'll see a value of 4 returned — this is because the browser returns the current value, *then* increments the variable. You can see that it's been incremented if you return the variable value again:

```
JS
num1;
```

The same is true of `--` : try the following

```
JS
let num2 = 6;
num2--;
num2;
```

> **Note:** You can make the browser do it the other way round — increment/decrement the variable *then* return the value — by putting the operator at the start of the variable instead of the end. Try the above examples again, but this time use `++num1` and `--num2`.

Assignment operators

Assignment operators are operators that assign a value to a variable. We have already used the most basic one, `=`, loads of times — it assigns the variable on the left the value stated on the right:

```
JS
let x = 3; // x contains the value 3
let y = 4; // y contains the value 4
x = y; // x now contains the same value y contains, 4
```

But there are some more complex types, which provide useful shortcuts to keep your code neater and more efficient. The most common are listed below:

Operator	Name	Purpose	Example	Shortcut for
`+=`	Addition assignment	Adds the value on the right to the variable value on the left, then returns the new variable value	`x += 4;`	`x = x + 4;`
`-=`	Subtraction assignment	Subtracts the value on the right from the variable value on the left, and returns the new variable value	`x -= 3;`	`x = x - 3;`

Operator	Name	Purpose	Example	Shortcut for
`*=`	Multiplication assignment	Multiplies the variable value on the left by the value on the right, and returns the new variable value	`x *= 3;`	`x = x * 3;`
`/=`	Division assignment	Divides the variable value on the left by the value on the right, and returns the new variable value	`x /= 5;`	`x = x / 5;`

Try typing some of the above examples into your console, to get an idea of how they work. In each case, see if you can guess what the value is before you type in the second line.

Note that you can quite happily use other variables on the right-hand side of each expression, for example:

JS

```
let x = 3; // x contains the value 3
let y = 4; // y contains the value 4
x *= y; // x now contains the value 12
```

> **Note:** There are lots of other assignment operators available, but these are the basic ones you should learn now.

Active learning: sizing a canvas box

In this exercise, you will manipulate some numbers and operators to change the size of a box. The box is drawn using a browser API called the Canvas API. There is no need to worry about how this works — just concentrate on the math for now. The width and height of the box (in pixels) are defined by the variables `x` and `y`, which are initially both given a value of 50.

Live output

The rectangle is 50px wide and 50px high.

Editable code

Press Esc to move focus away from the code area (Tab inserts a tab character).

```
let x = 50; let y = 50;

// Edit the two lines below here ONLY
x = 50;
y = 50;

ctx.fillStyle = 'green';
ctx.fillRect(10, 10, x, y);
```

Reset

Let's try the following:

Open in new window

In the editable code box above, there are two lines marked with a comment that we'd like you to update to make the box grow/shrink to certain sizes, using certain operators and/or values in each case. Let's try the following:

- Change the line that calculates x so the box is still 50px wide, but the 50 is calculated using the numbers 43 and 7 and an arithmetic operator.
- Change the line that calculates y so the box is 75px high, but the 75 is calculated using the numbers 25 and 3 and an arithmetic operator.
- Change the line that calculates x so the box is 250px wide, but the 250 is calculated using two numbers and the remainder (modulo) operator.
- Change the line that calculates y so the box is 150px high, but the 150 is calculated using three numbers and the subtraction and division operators.
- Change the line that calculates x so the box is 200px wide, but the 200 is calculated using the number 4 and an assignment operator.
- Change the line that calculates y so the box is 200px high, but the 200 is calculated using the numbers 50 and 3, the multiplication operator, and the addition assignment operator.

Don't worry if you totally mess the code up. You can always press the Reset button to get things working again. After you've answered all the above questions correctly, feel free to play with the code some more or create your own challenges.

Comparison operators

Sometimes we will want to run true/false tests, then act accordingly depending on the result of that test — to do this we use **comparison operators**.

Operator	Name	Purpose	Example
`===`	Strict equality	Tests whether the left and right values are identical to one another	`5 === 2 + 4`
`!==`	Strict-non-equality	Tests whether the left and right values are **not** identical to one another	`5 !== 2 + 3`
`<`	Less than	Tests whether the left value is smaller than the right one.	`10 < 6`
`>`	Greater than	Tests whether the left value is greater than the right one.	`10 > 20`
`<=`	Less than or equal to	Tests whether the left value is smaller than or equal to the right one.	`3 <= 2`
`>=`	Greater than or equal to	Tests whether the left value is greater than or equal to the right one.	`5 >= 4`

> **Note:** You may see some people using `==` and `!=` in their tests for equality and non-equality. These are valid operators in JavaScript, but they differ from `===` / `!==`. The former versions test whether the values are the same but not whether the values' datatypes are the same. The latter, strict versions test the equality of both the values and their datatypes. The strict versions tend to result in fewer errors, so we recommend you use them.

If you try entering some of these values in a console, you'll see that they all return `true` / `false` values — those booleans we mentioned in the last article. These are very useful, as they allow us to make decisions in our code, and they are used every time we want to make a choice of some kind. For example, booleans can be used to:

- Display the correct text label on a button depending on whether a feature is turned on or off
- Display a game over message if a game is over or a victory message if the game has been won
- Display the correct seasonal greeting depending on what holiday season it is
- Zoom a map in or out depending on what zoom level is selected

We'll look at how to code such logic when we look at conditional statements in a future article. For now, let's look at a quick example:

HTML

```
<button>Start machine</button>
<p>The machine is stopped.</p>
```

JS

```
const btn = document.querySelector("button");
const txt = document.querySelector("p");

btn.addEventListener("click", updateBtn);

function updateBtn() {
  if (btn.textContent === "Start machine") {
    btn.textContent = "Stop machine";
    txt.textContent = "The machine has started!";
  } else {
    btn.textContent = "Start machine";
    txt.textContent = "The machine is stopped.";
  }
}
```

Open in new window

You can see the equality operator being used just inside the `updateBtn()` function. In this case, we are not testing if two mathematical expressions have the same value — we are testing whether the text content of a button contains a certain string — but it is still the same principle at work. If the button is currently saying "Start machine" when it is pressed, we change its label to "Stop machine", and update the label as appropriate. If the button is currently saying "Stop machine" when it is pressed, we swap the display back again.

> **Note:** Such a control that swaps between two states is generally referred to as a **toggle**. It toggles between one state and another — light on, light off, etc.

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: Math.

Summary

In this article, we have covered the fundamental information you need to know about numbers in JavaScript, for now. You'll see numbers used again and again, all the way through your JavaScript learning, so it's a good idea to get this out of the way now. If you are one of those people that doesn't enjoy math, you can take comfort in the fact that this chapter was pretty short.

In the next article, we'll explore text and how JavaScript allows us to manipulate it.

> **Note:** If you do enjoy math and want to read more about how it is implemented in JavaScript, you can find a lot more detail in MDN's main JavaScript section. Great places to start are our Numbers and dates and Expressions and operators articles.

Help improve MDN

Was this page helpful to you?

Yes No

Learn how to contribute.

This page was last modified on Jan 1, 2024 by MDN contributors.

Handling text — strings in JavaScript

Next, we'll turn our attention to strings — this is what pieces of text are called in programming. In this article, we'll look at all the common things that you really ought to know about strings when learning JavaScript, such as creating strings, escaping quotes in strings, and joining strings together.

Prerequisites:	A basic understanding of HTML and CSS, an understanding of what JavaScript is.
Objective:	To gain familiarity with the basics of strings in JavaScript.

The power of words

Words are very important to humans — they are a large part of how we communicate. Since the web is a largely text-based medium designed to allow humans to communicate and share information, it is useful for us to have control over the words that appear on it. HTML provides structure and meaning to text, CSS allows us to precisely style it, and JavaScript offers many features for manipulating strings. These include creating custom welcome messages and prompts, showing the right text labels when needed, sorting terms into the desired order, and much more.

Pretty much all of the programs we've shown you so far in the course have involved some string manipulation.

Declaring strings

Strings are dealt with similarly to numbers at first glance, but when you dig deeper you'll start to see some notable differences. Let's start by entering some basic lines into the browser developer console to familiarize ourselves.

To start with, enter the following lines:

```
JS
const string = "The revolution will not be televised.";
console.log(string);
```

Just like we did with numbers, we are declaring a variable, initializing it with a string value, and then returning the value. The only difference here is that when writing a string, you need to surround the value with quotes.

If you don't do this, or miss one of the quotes, you'll get an error. Try entering the following lines:

```
JS
const badString1 = This is a test;
const badString2 = 'This is a test;
const badString3 = This is a test';
```

These lines don't work because any text without quotes around it is interpreted as a variable name, property name, reserved word, or similar. If the browser doesn't recognize the unquoted text, then an error is raised (e.g., "missing; before statement"). If the browser can detect where a string starts but not its end (owing to the missing second quote), it reports an "unterminated string literal" error. If your program is raising such errors, then go back and check all your strings to make sure you have no missing quotation marks.

The following will work if you previously defined the variable `string` — try it now:

```
JS
const badString = string;
console.log(badString);
```

`badString` is now set to have the same value as `string`.

Single quotes, double quotes, and backticks

In JavaScript, you can choose single quotes (`'`), double quotes (`"`), or backticks (`` ` ``) to wrap your strings in. All of the following will work:

```
JS
const single = 'Single quotes';
const double = "Double quotes";
const backtick = `Backtick`;

console.log(single);
console.log(double);
console.log(backtick);
```

You must use the same character for the start and end of a string, or you will get an error:

```
JS
const badQuotes = 'This is not allowed!";
```

Strings declared using single quotes and strings declared using double quotes are the same, and which you use is down to personal preference — although it is good practice to choose one style and use it consistently in your code.

Strings declared using backticks are a special kind of string called a *template literal*. In most ways, template literals are like normal strings, but they have some special properties:

- you can embed JavaScript in them
- you can declare template literals over multiple lines

Embedding JavaScript

Inside a template literal, you can wrap JavaScript variables or expressions inside `${ }`, and the result will be included in the string:

```
JS
const name = "Chris";
const greeting = `Hello, ${name}`;
console.log(greeting); // "Hello, Chris"
```

You can use the same technique to join together two variables:

```
JS
const one = "Hello, ";
const two = "how are you?";
const joined = `${one}${two}`;
console.log(joined); // "Hello, how are you?"
```

Joining strings together like this is called *concatenation*.

Concatenation in context

Let's have a look at concatenation being used in action:

HTML Play

```
<button>Press me</button>
<div id="greeting"></div>
```

JS Play

```
const button = document.querySelector("button");

function greet() {
  const name = prompt("What is your name?");
  const greeting = document.querySelector("#greeting");
  greeting.textContent = `Hello ${name}, nice to see you!`;
}

button.addEventListener("click", greet);
```

Play

Here, we are using the `window.prompt()` function, which prompts the user to answer a question via a popup dialog box and then stores the text they enter inside a given variable — in this case `name`. We then display a string that inserts the name into a generic greeting message.

Concatenation using "+"

You can use `${}` only with template literals, not normal strings. You can concatenate normal strings using the `+` operator:

JS

```
const greeting = "Hello";
const name = "Chris";
console.log(greeting + ", " + name); // "Hello, Chris"
```

However, template literals usually give you more readable code:

JS

```
const greeting = "Hello";
const name = "Chris";
console.log(`${greeting}, ${name}`); // "Hello, Chris"
```

Including expressions in strings

You can include JavaScript expressions in template literals, as well as just variables, and the results will be included in the result:

JS

```
const song = "Fight the Youth";
const score = 9;
const highestScore = 10;
const output = `I like the song ${song}. I gave it a score of ${
```

```
  (score / highestScore) * 100
}%.`;
console.log(output); // "I like the song Fight the Youth. I gave it a score of 90%."
```

Multiline strings

Template literals respect the line breaks in the source code, so you can write strings that span multiple lines like this:

JS

```
const newline = `One day you finally knew
what you had to do, and began,`;
console.log(newline);

/*
One day you finally knew
what you had to do, and began,
*/
```

To have the equivalent output using a normal string you'd have to include line break characters (`\n`) in the string:

JS

```
const newline = "One day you finally knew\nwhat you had to do, and began,";
console.log(newline);

/*
One day you finally knew
what you had to do, and began,
*/
```

See our Template literals reference page for more examples and details of advanced features.

Including quotes in strings

Since we use quotes to indicate the start and end of strings, how can we include actual quotes in strings? We know that this won't work:

JS

```
const badQuotes = "She said "I think so!"";
```

One common option is to use one of the other characters to declare the string:

JS

```
const goodQuotes1 = 'She said "I think so!"';
const goodQuotes2 = `She said "I'm not going in there!"`;
```

Another option is to *escape* the problem quotation mark. Escaping characters means that we do something to them to make sure they are recognized as text, not part of the code. In JavaScript, we do this by putting a backslash just before the character. Try this:

JS

```
const bigmouth = 'I\'ve got no right to take my place…';
console.log(bigmouth);
```

You can use the same technique to insert other special characters. See Escape sequences for more details.

Numbers vs. strings

What happens when we try to concatenate a string and a number? Let's try it in our console:

```
JS
const name = "Front ";
const number = 242;
console.log(name + number); // "Front 242"
```

You might expect this to return an error, but it works just fine. How numbers should be displayed as strings is fairly well-defined, so the browser automatically converts the number to a string and concatenates the two strings.

If you have a numeric variable that you want to convert to a string or a string variable that you want to convert to a number, you can use the following two constructs:

- The `Number()` function converts anything passed to it into a number if it can. Try the following:

  ```
  JS
  const myString = "123";
  const myNum = Number(myString);
  console.log(typeof myNum);
  // number
  ```

- Conversely, the `String()` function converts its argument to a string. Try this:

  ```
  JS
  const myNum2 = 123;
  const myString2 = String(myNum2);
  console.log(typeof myString2);
  // string
  ```

These constructs can be really useful in some situations. For example, if a user enters a number into a form's text field, it's a string. However, if you want to add this number to something, you'll need it to be a number, so you could pass it through `Number()` to handle this. We did exactly this in our Number Guessing Game, in line 59 .

Conclusion

So that's the very basics of strings covered in JavaScript. In the next article, we'll build on this, looking at some of the built-in methods available to strings in JavaScript and how we can use them to manipulate our strings into just the form we want.

Useful string methods

Now that we've looked at the very basics of strings, let's move up a gear and start thinking about what useful operations we can do on strings with built-in methods, such as finding the length of a text string, joining and splitting strings, substituting one character in a string for another, and more.

Prerequisites:	A basic understanding of HTML and CSS, an understanding of what JavaScript is.
Objective:	To understand that strings are objects, and learn how to use some of the basic methods available on those objects to manipulate strings.

Strings as objects

Most things are objects in JavaScript. When you create a string, for example by using

```
JS
const string = "This is my string";
```

your variable becomes a string object instance, and as a result has a large number of properties and methods available to it. You can see this if you go to the `String` object page and look down the list on the side of the page!

Now, before your brain starts melting, don't worry! You really don't need to know about most of these early on in your learning journey. But there are a few that you'll potentially use quite often that we'll look at here.

Let's enter some examples into the browser developer console.

Finding the length of a string

This is easy — you use the `length` property. Try entering the following lines:

```
JS
const browserType = "mozilla";
browserType.length;
```

This should return the number 7, because "mozilla" is 7 characters long. This is useful for many reasons; for example, you might want to find the lengths of a series of names so you can display them in order of length, or let a user know that a username they have entered into a form field is too long if it is over a certain length.

Retrieving a specific string character

On a related note, you can return any character inside a string by using **square bracket notation** — this means you include square brackets (`[]`) on the end of your variable name. Inside the square brackets, you include the number of the character you want to return, so for example to retrieve the first letter you'd do this:

```
JS
browserType[0];
```

Remember: computers count from 0, not 1!

To retrieve the last character of *any* string, we could use the following line, combining this technique with the `length` property we looked at above:

JS

```
browserType[browserType.length - 1];
```

The length of the string "mozilla" is 7, but because the count starts at 0, the last character's position is 6; using `length-1` gets us the last character.

Testing if a string contains a substring

Sometimes you'll want to find if a smaller string is present inside a larger one (we generally say *if a substring is present inside a string*). This can be done using the `includes()` method, which takes a single parameter — the substring you want to search for.

It returns `true` if the string contains the substring, and `false` otherwise.

JS

```
const browserType = "mozilla";

if (browserType.includes("zilla")) {
  console.log("Found zilla!");
} else {
  console.log("No zilla here!");
}
```

Often you'll want to know if a string starts or ends with a particular substring. This is a common enough need that there are two special methods for this: `startsWith()` and `endsWith()`:

JS

```
const browserType = "mozilla";

if (browserType.startsWith("zilla")) {
  console.log("Found zilla!");
} else {
  console.log("No zilla here!");
}
```

JS

```
const browserType = "mozilla";

if (browserType.endsWith("zilla")) {
  console.log("Found zilla!");
} else {
  console.log("No zilla here!");
}
```

Finding the position of a substring in a string

You can find the position of a substring inside a larger string using the `indexOf()` method. This method takes two parameters – the substring that you want to search for, and an optional parameter that specifies the starting point of the search.

If the string contains the substring, `indexOf()` returns the index of the first occurrence of the substring. If the string does not contain the substring, `indexOf()` returns `-1`.

JS

```
const tagline = "MDN - Resources for developers, by developers";
console.log(tagline.indexOf("developers")); // 20
```

Starting at `0`, if you count the number of characters (including the whitespace) from the beginning of the string, the first occurrence of the substring `"developers"` is at index `20`.

JS

```
console.log(tagline.indexOf("x")); // -1
```

This, on the other hand, returns `-1` because the character `x` is not present in the string.

So now that you know how to find the first occurrence of a substring, how do you go about finding subsequent occurrences? You can do that by passing in a value that's greater than the index of the previous occurrence as the second parameter to the method.

JS

```
const firstOccurrence = tagline.indexOf("developers");
const secondOccurrence = tagline.indexOf("developers", firstOccurrence + 1);

console.log(firstOccurrence); // 20
console.log(secondOccurrence); // 35
```

Here we're telling the method to search for the substring `"developers"` starting at index `21` (`firstOccurrence + 1`), and it returns the index `35`.

Extracting a substring from a string

You can extract a substring from a string using the `slice()` method. You pass it:

- the index at which to start extracting
- the index at which to stop extracting. This is exclusive, meaning that the character at this index is not included in the extracted substring.

For example:

JS

```
const browserType = "mozilla";
console.log(browserType.slice(1, 4)); // "ozi"
```

The character at index `1` is `"o"`, and the character at index 4 is `"l"`. So we extract all characters starting at `"o"` and ending just before `"l"`, giving us `"ozi"`.

If you know that you want to extract all of the remaining characters in a string after a certain character, you don't have to include the second parameter. Instead, you only need to include the character position from where you want to extract the remaining characters in a string. Try the following:

JS

```
browserType.slice(2); // "zilla"
```

This returns `"zilla"` — this is because the character position of 2 is the letter `"z"`, and because you didn't include a second parameter, the substring that was returned was all of the remaining characters in the string.

> **Note:** `slice()` has other options too; study the `slice()` page to see what else you can find out.

Changing case

The string methods `toLowerCase()` and `toUpperCase()` take a string and convert all the characters to lower- or uppercase, respectively. This can be useful for example if you want to normalize all user-entered data before storing it in a database.

Let's try entering the following lines to see what happens:

```
JS
const radData = "My NaMe Is MuD";
console.log(radData.toLowerCase());
console.log(radData.toUpperCase());
```

Updating parts of a string

You can replace one substring inside a string with another substring using the `replace()` method.

In this example, we're providing two parameters — the string we want to replace, and the string we want to replace it with:

```
JS
const browserType = "mozilla";
const updated = browserType.replace("moz", "van");

console.log(updated); // "vanilla"
console.log(browserType); // "mozilla"
```

Note that `replace()`, like many string methods, doesn't change the string it was called on, but returns a new string. If you want to update the original `browserType` variable, you would have to do something like this:

```
JS
let browserType = "mozilla";
browserType = browserType.replace("moz", "van");

console.log(browserType); // "vanilla"
```

Also note that we now have to declare `browserType` using `let`, not `const`, because we are reassigning it.

Be aware that `replace()` in this form only changes the first occurrence of the substring. If you want to change all occurrences, you can use `replaceAll()`:

```
JS
let quote = "To be or not to be";
quote = quote.replaceAll("be", "code");

console.log(quote); // "To code or not to code"
```

Active learning examples

In this section, we'll get you to try your hand at writing some string manipulation code. In each exercise below, we have an array of strings, and a loop that processes each value in the array and displays it in a bulleted list. You don't need to understand arrays or loops right now — these will be explained in future articles. All you need to do in each case is write the code that will output the strings in the format that we want them in.

Each example comes with a "Reset" button, which you can use to reset the code if you make a mistake and can't get it working again, and a "Show solution" button you can press to see a potential answer if you get really stuck.

Filtering greeting messages

In the first exercise, we'll start you off simple — we have an array of greeting card messages, but we want to sort them to list just the Christmas messages. We want you to fill in a conditional test inside the `if ()` structure to test each string and only print it in the list if it is a Christmas message.

Think about how you could test whether the message in each case is a Christmas message. What string is present in all of those messages, and what method could you use to test whether it is present?

Play

Fixing capitalization

In this exercise, we have the names of cities in the United Kingdom, but the capitalization is all messed up. We want you to change them so that they are all lowercase, except for a capital first letter. A good way to do this is to:

1. Convert the whole of the string contained in the `city` variable to lowercase and store it in a new variable.
2. Grab the first letter of the string in this new variable and store it in another variable.
3. Using this latest variable as a substring, replace the first letter of the lowercase string with the first letter of the lowercase string changed to upper case. Store the result of this replacement procedure in another new variable.
4. Change the value of the `result` variable to equal to the final result, not the `city`.

> **Note:** A hint — the parameters of the string methods don't have to be string literals; they can also be variables, or even variables with a method being invoked on them.

Play

Making new strings from old parts

In this last exercise, the array contains a bunch of strings containing information about train stations in the North of England. The strings are data items that contain the three-letter station code, followed by some machine-readable data, followed by a semicolon, followed by the human-readable station name. For example:

```
MAN675847583748sjt567654;Manchester Piccadilly
```

We want to extract the station code and name, and put them together in a string with the following structure:

```
MAN: Manchester Piccadilly
```

We'd recommend doing it like this:

1. Extract the three-letter station code and store it in a new variable.
2. Find the character index number of the semicolon.
3. Extract the human-readable station name using the semicolon character index number as a reference point, and store it in a new variable.
4. Concatenate the two new variables and a string literal to make the final string.
5. Change the value of the `result` variable to the final string, not the `station`.

Play

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: Strings.

Conclusion

You can't escape the fact that being able to handle words and sentences in programming is very important — particularly in JavaScript, as websites are all about communicating with people. This article has given you the basics that you need to know about manipulating strings for now. This should serve you well as you go into more complex topics in the future. Next, we're going to look at the last major type of data we need to focus on in the short term — arrays.

Arrays

In the final article of this module, we'll look at arrays — a neat way of storing a list of data items under a single variable name. Here we look at why this is useful, then explore how to create an array, retrieve, add, and remove items stored in an array, and more besides.

Prerequisites:	A basic understanding of HTML and CSS, an understanding of what JavaScript is.
Objective:	To understand what arrays are and how to manipulate them in JavaScript.

What is an array?

Arrays are generally described as "list-like objects"; they are basically single objects that contain multiple values stored in a list. Array objects can be stored in variables and dealt with in much the same way as any other type of value, the difference being that we can access each value inside the list individually, and do super useful and efficient things with the list, like loop through it and do the same thing to every value. Maybe we've got a series of product items and their prices stored in an array, and we want to loop through them all and print them out on an invoice, while totaling all the prices together and printing out the total price at the bottom.

If we didn't have arrays, we'd have to store every item in a separate variable, then call the code that does the printing and adding separately for each item. This would be much longer to write out, less efficient, and more error-prone. If we had 10 items to add to the invoice it would already be annoying, but what about 100 items, or 1000? We'll return to this example later on in the article.

As in previous articles, let's learn about the real basics of arrays by entering some examples into browser developer console.

Creating arrays

Arrays consist of square brackets and items that are separated by commas.

1. Suppose we want to store a shopping list in an array. Paste the following code into the console:

   JS

   ```
   const shopping = ["bread", "milk", "cheese", "hummus", "noodles"];
   console.log(shopping);
   ```

2. In the above example, each item is a string, but in an array we can store various data types — strings, numbers, objects, and even other arrays. We can also mix data types in a single array — we do not have to limit ourselves to storing only numbers in one array, and in another only strings. For example:

   JS

   ```
   const sequence = [1, 1, 2, 3, 5, 8, 13];
   const random = ["tree", 795, [0, 1, 2]];
   ```

3. Before proceeding, create a few example arrays.

Finding the length of an array

You can find out the length of an array (how many items are in it) in exactly the same way as you find out the length (in characters) of a string — by using the `length` property. Try the following:

```
JS
const shopping = ["bread", "milk", "cheese", "hummus", "noodles"];
console.log(shopping.length); // 5
```

Accessing and modifying array items

Items in an array are numbered, starting from zero. This number is called the item's *index*. So the first item has index 0, the second has index 1, and so on. You can access individual items in the array using bracket notation and supplying the item's index, in the same way that you accessed the letters in a string.

1. Enter the following into your console:

   ```
   JS
   const shopping = ["bread", "milk", "cheese", "hummus", "noodles"];
   console.log(shopping[0]);
   // returns "bread"
   ```

2. You can also modify an item in an array by giving a single array item a new value. Try this:

   ```
   JS
   const shopping = ["bread", "milk", "cheese", "hummus", "noodles"];
   shopping[0] = "tahini";
   console.log(shopping);
   // shopping will now return [ "tahini", "milk", "cheese", "hummus", "noodles" ]
   ```

   > **Note:** We've said it before, but just as a reminder — computers start counting from 0!

3. Note that an array inside an array is called a multidimensional array. You can access an item inside an array that is itself inside another array by chaining two sets of square brackets together. For example, to access one of the items inside the array that is the third item inside the `random` array (see previous section), we could do something like this:

   ```
   JS
   const random = ["tree", 795, [0, 1, 2]];
   random[2][2];
   ```

4. Try making some more modifications to your array examples before moving on. Play around a bit, and see what works and what doesn't.

Finding the index of items in an array

If you don't know the index of an item, you can use the `indexOf()` method. The `indexOf()` method takes an item as an argument and will either return the item's index or `-1` if the item is not in the array:

```
JS
const birds = ["Parrot", "Falcon", "Owl"];
console.log(birds.indexOf("Owl")); //  2
console.log(birds.indexOf("Rabbit")); // -1
```

Adding items

To add one or more items to the end of an array we can use `push()`. Note that you need to include one or more items that you want to add to the end of your array.

```
JS
const cities = ["Manchester", "Liverpool"];
cities.push("Cardiff");
```

```
console.log(cities); // [ "Manchester", "Liverpool", "Cardiff" ]
cities.push("Bradford", "Brighton");
console.log(cities); // [ "Manchester", "Liverpool", "Cardiff", "Bradford", "Brighton" ]
```

The new length of the array is returned when the method call completes. If you wanted to store the new array length in a variable, you could do something like this:

JS

```
const cities = ["Manchester", "Liverpool"];
const newLength = cities.push("Bristol");
console.log(cities); // [ "Manchester", "Liverpool", "Bristol" ]
console.log(newLength); // 3
```

To add an item to the start of the array, use `unshift()`:

JS

```
const cities = ["Manchester", "Liverpool"];
cities.unshift("Edinburgh");
console.log(cities); // [ "Edinburgh", "Manchester", "Liverpool" ]
```

Removing items

To remove the last item from the array, use `pop()`.

JS

```
const cities = ["Manchester", "Liverpool"];
cities.pop();
console.log(cities); // [ "Manchester" ]
```

The `pop()` method returns the item that was removed. To save that item in a new variable, you could do this:

JS

```
const cities = ["Manchester", "Liverpool"];
const removedCity = cities.pop();
console.log(removedCity); // "Liverpool"
```

To remove the first item from an array, use `shift()`:

JS

```
const cities = ["Manchester", "Liverpool"];
cities.shift();
console.log(cities); // [ "Liverpool" ]
```

If you know the index of an item, you can remove it from the array using `splice()`:

JS

```
const cities = ["Manchester", "Liverpool", "Edinburgh", "Carlisle"];
const index = cities.indexOf("Liverpool");
if (index !== -1) {
  cities.splice(index, 1);
}
console.log(cities); // [ "Manchester", "Edinburgh", "Carlisle" ]
```

In this call to `splice()`, the first argument says where to start removing items, and the second argument says how many items should be removed. So you can remove more than one item:

```
JS
const cities = ["Manchester", "Liverpool", "Edinburgh", "Carlisle"];
const index = cities.indexOf("Liverpool");
if (index !== -1) {
  cities.splice(index, 2);
}
console.log(cities); // [ "Manchester", "Carlisle" ]
```

Accessing every item

Very often you will want to access every item in the array. You can do this using the `for...of` statement:

```
JS
const birds = ["Parrot", "Falcon", "Owl"];

for (const bird of birds) {
  console.log(bird);
}
```

Sometimes you will want to do the same thing to each item in an array, leaving you with an array containing the changed items. You can do this using `map()`. The code below takes an array of numbers and doubles each number:

```
JS
function double(number) {
  return number * 2;
}
const numbers = [5, 2, 7, 6];
const doubled = numbers.map(double);
console.log(doubled); // [ 10, 4, 14, 12 ]
```

We give a function to the `map()`, and `map()` calls the function once for each item in the array, passing in the item. It then adds the return value from each function call to a new array, and finally returns the new array.

Sometimes you'll want to create a new array containing only the items in the original array that match some test. You can do that using `filter()`. The code below takes an array of strings and returns an array containing just the strings that are greater than 8 characters long:

```
JS
function isLong(city) {
  return city.length > 8;
}
const cities = ["London", "Liverpool", "Totnes", "Edinburgh"];
const longer = cities.filter(isLong);
console.log(longer); // [ "Liverpool", "Edinburgh" ]
```

Like `map()`, we give a function to the `filter()` method, and `filter()` calls this function for every item in the array, passing in the item. If the function returns `true`, then the item is added to a new array. Finally it returns the new array.

Converting between strings and arrays

Often you'll be presented with some raw data contained in a big long string, and you might want to separate the useful items out into a more useful form and then do things to them, like display them in a data table. To do this, we can use the `split()` method. In

its simplest form, this takes a single parameter, the character you want to separate the string at, and returns the substrings between the separator as items in an array.

> **Note:** Okay, this is technically a string method, not an array method, but we've put it in with arrays as it goes well here.

1. Let's play with this, to see how it works. First, create a string in your console:

```
JS
const data = "Manchester,London,Liverpool,Birmingham,Leeds,Carlisle";
```

2. Now let's split it at each comma:

```
JS
const cities = data.split(",");
cities;
```

3. Finally, try finding the length of your new array, and retrieving some items from it:

```
JS
cities.length;
cities[0]; // the first item in the array
cities[1]; // the second item in the array
cities[cities.length - 1]; // the last item in the array
```

4. You can also go the opposite way using the `join()` method. Try the following:

```
JS
const commaSeparated = cities.join(",");
commaSeparated;
```

5. Another way of converting an array to a string is to use the `toString()` method. `toString()` is arguably simpler than `join()` as it doesn't take a parameter, but more limiting. With `join()` you can specify different separators, whereas `toString()` always uses a comma. (Try running Step 4 with a different character than a comma.)

```
JS
const dogNames = ["Rocket", "Flash", "Bella", "Slugger"];
dogNames.toString(); // Rocket,Flash,Bella,Slugger
```

Active learning: Printing those products

Let's return to the example we described earlier — printing out product names and prices on an invoice, then totaling the prices and printing them at the bottom. In the editable example below there are comments containing numbers — each of these marks a place where you have to add something to the code. They are as follows:

1. Below the `// number 1` comment are a number of strings, each one containing a product name and price separated by a colon. We'd like you to turn this into an array and store it in an array called `products`.
2. Below the `// number 2` comment, start a `for...of()` loop to go through every item in the `products` array.
3. Below the `// number 3` comment we want you to write a line of code that splits the current array item (`name:price`) into two separate items, one containing just the name and one containing just the price. If you are not sure how to do this, consult the Useful string methods article for some help, or even better, look at the Converting between strings and arrays section of this article.
4. As part of the above line of code, you'll also want to convert the price from a string to a number. If you can't remember how to do this, check out the first strings article.
5. There is a variable called `total` that is created and given a value of 0 at the top of the code. Inside the loop (below `// number 4`) we want you to add a line that adds the current item price to that total in each iteration of the loop, so that at the end of the code the correct total is printed onto the invoice. You might need an assignment operator to do this.

6. We want you to change the line just below `// number 5` so that the `itemText` variable is made equal to "current item name — $current item price", for example "Shoes — $23.99" in each case, so the correct information for each item is printed on the invoice. This is just simple string concatenation, which should be familiar to you.
7. Finally, below the `// number 6` comment, you'll need to add a `}` to mark the end of the `for...of()` loop.

Play

Active learning: Top 5 searches

A good use for array methods like `push()` and `pop()` is when you are maintaining a record of currently active items in a web app. In an animated scene for example, you might have an array of objects representing the background graphics currently displayed, and you might only want 50 displayed at once, for performance or clutter reasons. As new objects are created and added to the array, older ones can be deleted from the array to maintain the desired number.

In this example we're going to show a much simpler use — here we're giving you a fake search site, with a search box. The idea is that when terms are entered in the search box, the top 5 previous search terms are displayed in the list. When the number of terms goes over 5, the last term starts being deleted each time a new term is added to the top, so the 5 previous terms are always displayed.

> **Note:** In a real search app, you'd probably be able to click the previous search terms to return to previous searches, and it would display actual search results! We are just keeping it simple for now.

To complete the app, we need you to:

1. Add a line below the `// number 1` comment that adds the current value entered into the search input to the start of the array. This can be retrieved using `searchInput.value`.
2. Add a line below the `// number 2` comment that removes the value currently at the end of the array.

Play

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: Arrays.

Conclusion

After reading through this article, we are sure you will agree that arrays seem pretty darn useful; you'll see them crop up everywhere in JavaScript, often in association with loops in order to do the same thing to every item in an array. We'll be teaching you all the useful basics there are to know about loops in the next module, but for now you should give yourself a clap and take a well-deserved break; you've worked through all the articles in this module!

The only thing left to do is work through this module's assessment, which will test your understanding of the articles that came before it.

See also

- Indexed collections — an advanced level guide to arrays and their cousins, typed arrays.

- `Array` — the `Array` object reference page — for a detailed reference guide to the features discussed in this page, and many more.

Silly story generator

In this assessment you'll be tasked with taking some of the knowledge you've picked up in this module's articles and applying it to creating a fun app that generates random silly stories. Have fun!

Prerequisites:	Before attempting this assessment you should have already worked through all the articles in this module.
Objective:	To test comprehension of JavaScript fundamentals, such as variables, numbers, operators, strings, and arrays.

Starting point

To get this assessment started, you should:

- Go and grab the HTML file for the example, save a local copy of it as `index.html` in a new directory somewhere on your computer, and do the assessment locally to begin with. This also has the CSS to style the example contained within it.
- Go to the page containing the raw text and keep this open in a separate browser tab somewhere. You'll need it later.

Alternatively, you could use an online editor such as CodePen, JSFiddle, or Glitch. You could paste the HTML, CSS and JavaScript into one of these online editors. If the online editor you are using doesn't have a separate JavaScript panel, feel free to put it inline in a `<script>` element inside the HTML page.

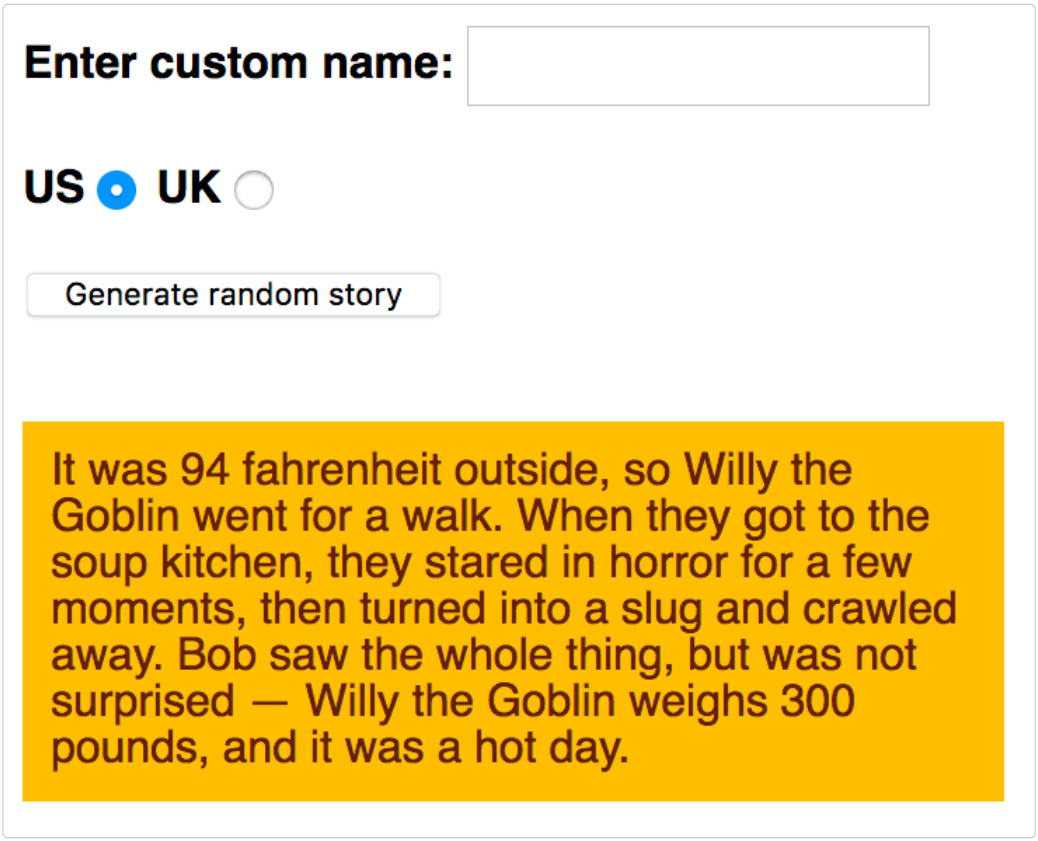
> **Note:** If you get stuck, you can reach out to us in one of our communication channels.

Project brief

You have been provided with some raw HTML/CSS and a few text strings and JavaScript functions; you need to write the necessary JavaScript to turn this into a working program, which does the following:

- Generates a silly story when the "Generate random story" button is pressed.
- Replaces the default name "Bob" in the story with a custom name, only if a custom name is entered into the "Enter custom name" text field before the generate button is pressed.
- Converts the default US weight and temperature quantities and units in the story into UK equivalents if the UK radio button is checked before the generate button is pressed.
- Generates a new random silly story every time the button is pressed.

The following screenshot shows an example of what the finished program should output:



To give you more of an idea, have a look at the finished example (no peeking at the source code!)

Steps to complete

The following sections describe what you need to do.

Basic setup:

1. Create a new file called `main.js`, in the same directory as your `index.html` file.
2. Apply the external JavaScript file to your HTML by inserting a `<script>` element into your HTML referencing `main.js`. Put it just before the closing `</body>` tag.

Initial variables and functions:

1. In the raw text file, copy all of the code underneath the heading "1. COMPLETE VARIABLE AND FUNCTION DEFINITIONS" and paste it into the top of the `main.js` file. This gives you three variables that store references to the "Enter custom name" text field (`customName`), the "Generate random story" button (`randomize`), and the `<p>` element at the bottom of the HTML body that the story will be copied into (`story`), respectively. In addition you've got a function called `randomValueFromArray()` that takes an array, and returns one of the items stored inside the array at random.
2. Now look at the second section of the raw text file — "2. RAW TEXT STRINGS". This contains text strings that will act as input into our program. We'd like you to contain these inside variables inside `main.js`:
    i. Store the first, big long, string of text inside a variable called `storyText`.
    ii. Store the first set of three strings inside an array called `insertX`.
    iii. Store the second set of three strings inside an array called `insertY`.
    iv. Store the third set of three strings inside an array called `insertZ`.

Placing the event handler and incomplete function:

1. Now return to the raw text file.
2. Copy the code found underneath the heading "3. EVENT LISTENER AND PARTIAL FUNCTION DEFINITION" and paste it into the bottom of your `main.js` file. This:
    - Adds a click event listener to the `randomize` variable so that when the button it represents is clicked, the `result()` function is run.
    - Adds a partially-completed `result()` function definition to your code. For the remainder of the assessment, you'll be filling in lines inside this function to complete it and make it work properly.

Completing the `result()` function:

1. Create a new variable called `newStory`, and set its value to equal `storyText`. This is needed so we can create a new random story each time the button is pressed and the function is run. If we made changes directly to `storyText`, we'd only be able to generate a new story once.
2. Create three new variables called `xItem`, `yItem`, and `zItem`, and make them equal to the result of calling `randomValueFromArray()` on your three arrays (the result in each case will be a random item out of each array it is called on). For example you can call the function and get it to return one random string out of `insertX` by writing `randomValueFromArray(insertX)`.
3. Next we want to replace the three placeholders in the `newStory` string — `:insertx:`, `:inserty:`, and `:insertz:` — with the strings stored in `xItem`, `yItem`, and `zItem`. There are two possible string methods that will help you here — in each case, make the call to the method equal to `newStory`, so each time it is called, `newStory` is made equal to itself, but with substitutions made. So each time the button is pressed, these placeholders are each replaced with a random silly string. As a further hint, depending on the method you choose, you might need to make one of the calls twice.
4. Inside the first `if` block, add another string replacement method call to replace the name 'Bob' found in the `newStory` string with the `name` variable. In this block we are saying "If a value has been entered into the `customName` text input, replace Bob in the story with that custom name."
5. Inside the second `if` block, we are checking to see if the `uk` radio button has been selected. If so, we want to convert the weight and temperature values in the story from pounds and Fahrenheit into stones and centigrade. What you need to do is as follows:
    i. Look up the formulas for converting pounds to stone, and Fahrenheit to centigrade.
    ii. Inside the line that defines the `weight` variable, replace 300 with a calculation that converts 300 pounds into stones. Concatenate `' stone'` onto the end of the result of the overall `Math.round()` call.
    iii. Inside the line that defines the `temperature` variable, replace 94 with a calculation that converts 94 Fahrenheit into centigrade. Concatenate `' centigrade'` onto the end of the result of the overall `Math.round()` call.
    iv. Just under the two variable definitions, add two more string replacement lines that replace '94 fahrenheit' with the contents of the `temperature` variable, and '300 pounds' with the contents of the `weight` variable.
6. Finally, in the second-to-last line of the function, make the `textContent` property of the `story` variable (which references the paragraph) equal to `newStory`.

Hints and tips

- You don't need to edit the HTML in any way, except to apply the JavaScript to your HTML.
- If you are unsure whether the JavaScript is applied to your HTML properly, try removing everything else from the JavaScript file temporarily, adding in a simple bit of JavaScript that you know will create an obvious effect, then saving and refreshing. The following for example turns the background of the `<html>` element red — so the entire browser window should go red if the JavaScript is applied properly:

```
JS
document.querySelector("html").style.backgroundColor = "red";
```

- `Math.round()` is a built-in JavaScript method that rounds the result of a calculation to the nearest whole number.
- There are three instances of strings that need to be replaced. You may repeat the `replace()` method multiple times, or you can use `replaceAll()`. Remember, Strings are immutable!

Asynchronous JavaScript

In this module, we take a look at asynchronous JavaScript, why it is important, and how it can be used to effectively handle potential blocking operations, such as fetching resources from a server.

Prerequisites

Asynchronous JavaScript is a fairly advanced topic, and you are advised to work through JavaScript first steps and JavaScript building blocks modules before attempting this.

> **Note:** If you are working on a computer/tablet/other device where you don't have the ability to create your own files, you can try out (most of) the code examples in an online coding program such as JSBin or Glitch.

Guides

Introducing asynchronous JavaScript

In this article, we'll learn about **synchronous** and **asynchronous** programming, why we often need to use asynchronous techniques, and the problems related to the way asynchronous functions have historically been implemented in JavaScript.

How to use promises

Here we'll introduce promises and show how to use promise-based APIs. We'll also introduce the `async` and `await` keywords.

Implementing a promise-based API

This article will outline how to implement your own promise-based API.

Introducing workers

Workers enable you to run certain tasks in a separate thread to keep your main code responsive. In this article, we'll rewrite a long-running synchronous function to use a worker.

Assessments

Sequencing animations

The assessment asks you to use promises to play a set of animations in a particular sequence.

See also

- Asynchronous Programming from the fantastic Eloquent JavaScript online book by Marijn Haverbeke.

This page was last modified on Mar 5, 2024 by MDN contributors.

Asynchronous JavaScript

In this module, we take a look at asynchronous JavaScript, why it is important, and how it can be used to effectively handle potential blocking operations, such as fetching resources from a server.

Prerequisites

Asynchronous JavaScript is a fairly advanced topic, and you are advised to work through JavaScript first steps and JavaScript building blocks modules before attempting this.

> **Note:** If you are working on a computer/tablet/other device where you don't have the ability to create your own files, you can try out (most of) the code examples in an online coding program such as JSBin or Glitch.

Guides

Introducing asynchronous JavaScript

In this article, we'll learn about **synchronous** and **asynchronous** programming, why we often need to use asynchronous techniques, and the problems related to the way asynchronous functions have historically been implemented in JavaScript.

How to use promises

Here we'll introduce promises and show how to use promise-based APIs. We'll also introduce the `async` and `await` keywords.

Implementing a promise-based API

This article will outline how to implement your own promise-based API.

Introducing workers

Workers enable you to run certain tasks in a separate thread to keep your main code responsive. In this article, we'll rewrite a long-running synchronous function to use a worker.

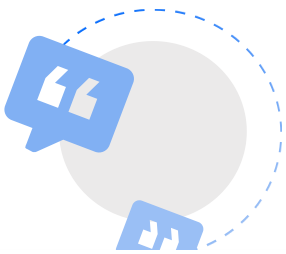
Assessments

Sequencing animations

The assessment asks you to use promises to play a set of animations in a particular sequence.

See also

- Asynchronous Programming from the fantastic Eloquent JavaScript online book by Marijn Haverbeke.



This page was last modified on Mar 5, 2024 by MDN contributors.

Introducing asynchronous JavaScript

In this article, we'll explain what asynchronous programming is, why we need it, and briefly discuss some of the ways asynchronous functions have historically been implemented in JavaScript.

Prerequisites:	A reasonable understanding of JavaScript fundamentals, including functions and event handlers.
Objective:	To gain familiarity with what asynchronous JavaScript is, how it differs from synchronous JavaScript, and why we need it.

Asynchronous programming is a technique that enables your program to start a potentially long-running task and still be able to be responsive to other events while that task runs, rather than having to wait until that task has finished. Once that task has finished, your program is presented with the result.

Many functions provided by browsers, especially the most interesting ones, can potentially take a long time, and therefore, are asynchronous. For example:

- Making HTTP requests using `fetch()`
- Accessing a user's camera or microphone using `getUserMedia()`
- Asking a user to select files using `showOpenFilePicker()`

So even though you may not have to *implement* your own asynchronous functions very often, you are very likely to need to *use* them correctly.

In this article, we'll start by looking at the problem with long-running synchronous functions, which make asynchronous programming a necessity.

Synchronous programming

Consider the following code:

```
JS
const name = "Miriam";
const greeting = `Hello, my name is ${name}!`;
console.log(greeting);
// "Hello, my name is Miriam!"
```

This code:

1. Declares a string called `name`.
2. Declares another string called `greeting`, which uses `name`.
3. Outputs the greeting to the JavaScript console.

We should note here that the browser effectively steps through the program one line at a time, in the order we wrote it. At each point, the browser waits for the line to finish its work before going on to the next line. It has to do this because each line depends on the work done in the preceding lines.

That makes this a **synchronous program**. It would still be synchronous even if we called a separate function, like this:

JS

```
function makeGreeting(name) {
  return `Hello, my name is ${name}!`;
}

const name = "Miriam";
const greeting = makeGreeting(name);
console.log(greeting);
// "Hello, my name is Miriam!"
```

Here, `makeGreeting()` is a **synchronous function** because the caller has to wait for the function to finish its work and return a value before the caller can continue.

A long-running synchronous function

What if the synchronous function takes a long time?

The program below uses a very inefficient algorithm to generate multiple large prime numbers when a user clicks the "Generate primes" button. The higher the number of primes a user specifies, the longer the operation will take.

HTML Play

```
<label for="quota">Number of primes:</label>
<input type="text" id="quota" name="quota" value="1000000" />

<button id="generate">Generate primes</button>
<button id="reload">Reload</button>

<div id="output"></div>
```

JS Play

```
const MAX_PRIME = 1000000;

function isPrime(n) {
  for (let i = 2; i <= Math.sqrt(n); i++) {
    if (n % i === 0) {
      return false;
    }
  }
  return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generatePrimes(quota) {
  const primes = [];
  while (primes.length < quota) {
    const candidate = random(MAX_PRIME);
    if (isPrime(candidate)) {
      primes.push(candidate);
    }
  }
  return primes;
}

const quota = document.querySelector("#quota");
const output = document.querySelector("#output");
```

```
document.querySelector("#generate").addEventListener("click", () => {
  const primes = generatePrimes(quota.value);
  output.textContent = `Finished generating ${quota.value} primes!`;
});

document.querySelector("#reload").addEventListener("click", () => {
  document.location.reload();
});
```

Play

Try clicking "Generate primes". Depending on how fast your computer is, it will probably take a few seconds before the program displays the "Finished!" message.

The trouble with long-running synchronous functions

The next example is just like the last one, except we added a text box for you to type in. This time, click "Generate primes", and try typing in the text box immediately after.

You'll find that while our `generatePrimes()` function is running, our program is completely unresponsive: you can't type anything, click anything, or do anything else.

Play

The reason for this is that this JavaScript program is *single-threaded*. A thread is a sequence of instructions that a program follows. Because the program consists of a single thread, it can only do one thing at a time: so if it is waiting for our long-running synchronous call to return, it can't do anything else.

What we need is a way for our program to:

1. Start a long-running operation by calling a function.
2. Have that function start the operation and return immediately, so that our program can still be responsive to other events.
3. Have the function execute the operation in a way that does not block the main thread, for example by starting a new thread.
4. Notify us with the result of the operation when it eventually completes.

That's precisely what asynchronous functions enable us to do. The rest of this module explains how they are implemented in JavaScript.

Event handlers

The description we just saw of asynchronous functions might remind you of event handlers, and if it does, you'd be right. Event handlers are really a form of asynchronous programming: you provide a function (the event handler) that will be called, not right away, but whenever the event happens. If "the event" is "the asynchronous operation has completed", then that event could be used to notify the caller about the result of an asynchronous function call.

Some early asynchronous APIs used events in just this way. The `XMLHttpRequest` API enables you to make HTTP requests to a remote server using JavaScript. Since this can take a long time, it's an asynchronous API, and you get notified about the progress and eventual completion of a request by attaching event listeners to the `XMLHttpRequest` object.

The following example shows this in action. Press "Click to start request" to send a request. We create a new `XMLHttpRequest` and listen for its `loadend` event. The handler logs a "Finished!" message along with the status code.

After adding the event listener we send the request. Note that after this, we can log "Started XHR request": that is, our program can continue to run while the request is going on, and our event handler will be called when the request is complete.

HTML Play

```
<button id="xhr">Click to start request</button>
<button id="reload">Reload</button>

<pre readonly class="event-log"></pre>
```

JS Play

```
const log = document.querySelector(".event-log");

document.querySelector("#xhr").addEventListener("click", () => {
  log.textContent = "";

  const xhr = new XMLHttpRequest();

  xhr.addEventListener("loadend", () => {
    log.textContent = `${log.textContent}Finished with status: ${xhr.status}`;
  });

  xhr.open(
    "GET",
    "https://raw.githubusercontent.com/mdn/content/main/files/en-us/_wikihistory.json",
  );
  xhr.send();
  log.textContent = `${log.textContent}Started XHR request\n`;
});

document.querySelector("#reload").addEventListener("click", () => {
  log.textContent = "";
  document.location.reload();
});
```

Play

This is just like the event handlers we've encountered in a previous module, except that instead of the event being a user action, such as the user clicking a button, the event is a change in the state of some object.

Callbacks

An event handler is a particular type of callback. A callback is just a function that's passed into another function, with the expectation that the callback will be called at the appropriate time. As we just saw, callbacks used to be the main way asynchronous functions were implemented in JavaScript.

However, callback-based code can get hard to understand when the callback itself has to call functions that accept a callback. This is a common situation if you need to perform some operation that breaks down into a series of asynchronous functions. For example, consider the following:

```
JS
function doStep1(init) {
  return init + 1;
}

function doStep2(init) {
  return init + 2;
}

function doStep3(init) {
  return init + 3;
}

function doOperation() {
  let result = 0;
  result = doStep1(result);
  result = doStep2(result);
  result = doStep3(result);
  console.log(`result: ${result}`);
}

doOperation();
```

Here we have a single operation that's split into three steps, where each step depends on the last step. In our example, the first step adds 1 to the input, the second adds 2, and the third adds 3. Starting with an input of 0, the end result is 6 (0 + 1 + 2 + 3). As a synchronous program, this is very straightforward. But what if we implemented the steps using callbacks?

```
JS
function doStep1(init, callback) {
  const result = init + 1;
  callback(result);
}

function doStep2(init, callback) {
  const result = init + 2;
  callback(result);
}

function doStep3(init, callback) {
  const result = init + 3;
  callback(result);
}

function doOperation() {
  doStep1(0, (result1) => {
    doStep2(result1, (result2) => {
```

```
      doStep3(result2, (result3) => {
        console.log(`result: ${result3}`);
      });
    });
  });
}

doOperation();
```

Because we have to call callbacks inside callbacks, we get a deeply nested `doOperation()` function, which is much harder to read and debug. This is sometimes called "callback hell" or the "pyramid of doom" (because the indentation looks like a pyramid on its side).

When we nest callbacks like this, it can also get very hard to handle errors: often you have to handle errors at each level of the "pyramid", instead of having error handling only once at the top level.

For these reasons, most modern asynchronous APIs don't use callbacks. Instead, the foundation of asynchronous programming in JavaScript is the `Promise`, and that's the subject of the next article.

How to use promises

Promises are the foundation of asynchronous programming in modern JavaScript. A promise is an object returned by an asynchronous function, which represents the current state of the operation. At the time the promise is returned to the caller, the operation often isn't finished, but the promise object provides methods to handle the eventual success or failure of the operation.

Prerequisites:	A reasonable understanding of JavaScript fundamentals, including event handling.
Objective:	To understand how to use promises in JavaScript.

In the previous article, we talked about the use of callbacks to implement asynchronous functions. With that design, you call the asynchronous function, passing in your callback function. The function returns immediately and calls your callback when the operation is finished.

With a promise-based API, the asynchronous function starts the operation and returns a `Promise` object. You can then attach handlers to this promise object, and these handlers will be executed when the operation has succeeded or failed.

Using the fetch() API

> **Note:** In this article, we will explore promises by copying code samples from the page into your browser's JavaScript console. To set this up:
>
> 1. open a browser tab and visit https://example.org
> 2. in that tab, open the JavaScript console in your browser's developer tools
> 3. when we show an example, copy it into the console. You will have to reload the page each time you enter a new example, or the console will complain that you have redeclared `fetchPromise`.

In this example, we'll download the JSON file from https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json , and log some information about it.

To do this, we'll make an **HTTP request** to the server. In an HTTP request, we send a request message to a remote server, and it sends us back a response. In this case, we'll send a request to get a JSON file from the server. Remember in the last article, where we made HTTP requests using the `XMLHttpRequest` API? Well, in this article, we'll use the `fetch()` API, which is the modern, promise-based replacement for `XMLHttpRequest`.

Copy this into your browser's JavaScript console:

JS

```
const fetchPromise = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);

console.log(fetchPromise);

fetchPromise.then((response) => {
  console.log(`Received response: ${response.status}`);
});
```

```
console.log("Started request…");
```

Here we are:

1. calling the `fetch()` API, and assigning the return value to the `fetchPromise` variable
2. immediately after, logging the `fetchPromise` variable. This should output something like: `Promise { <state>: "pending" }`, telling us that we have a `Promise` object, and it has a `state` whose value is `"pending"`. The `"pending"` state means that the fetch operation is still going on.
3. passing a handler function into the Promise's **`then()`** method. When (and if) the fetch operation succeeds, the promise will call our handler, passing in a `Response` object, which contains the server's response.
4. logging a message that we have started the request.

The complete output should be something like:

```
Promise { <state>: "pending" }
Started request…
Received response: 200
```

Note that `Started request…` is logged before we receive the response. Unlike a synchronous function, `fetch()` returns while the request is still going on, enabling our program to stay responsive. The response shows the `200` (OK) status code, meaning that our request succeeded.

This probably seems a lot like the example in the last article, where we added event handlers to the `XMLHttpRequest` object. Instead of that, we're passing a handler into the `then()` method of the returned promise.

Chaining promises

With the `fetch()` API, once you get a `Response` object, you need to call another function to get the response data. In this case, we want to get the response data as JSON, so we would call the `json()` method of the `Response` object. It turns out that `json()` is also asynchronous. So this is a case where we have to call two successive asynchronous functions.

Try this:

JS

```
const fetchPromise = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);

fetchPromise.then((response) => {
  const jsonPromise = response.json();
  jsonPromise.then((data) => {
    console.log(data[0].name);
  });
});
```

In this example, as before, we add a `then()` handler to the promise returned by `fetch()`. But this time, our handler calls `response.json()`, and then passes a new `then()` handler into the promise returned by `response.json()`.

This should log "baked beans" (the name of the first product listed in "products.json").

But wait! Remember the last article, where we said that by calling a callback inside another callback, we got successively more nested levels of code? And we said that this "callback hell" made our code hard to understand? Isn't this just the same, only with

`then()` calls?

It is, of course. But the elegant feature of promises is that *`then()` itself returns a promise, which will be completed with the result of the function passed to it*. This means that we can (and certainly should) rewrite the above code like this:

JS

```
const fetchPromise = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);

fetchPromise
  .then((response) => response.json())
  .then((data) => {
    console.log(data[0].name);
  });
```

Instead of calling the second `then()` inside the handler for the first `then()`, we can *return* the promise returned by `json()`, and call the second `then()` on that return value. This is called **promise chaining** and means we can avoid ever-increasing levels of indentation when we need to make consecutive asynchronous function calls.

Before we move on to the next step, there's one more piece to add. We need to check that the server accepted and was able to handle the request, before we try to read it. We'll do this by checking the status code in the response and throwing an error if it wasn't "OK":

JS

```
const fetchPromise = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);

fetchPromise
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    return response.json();
  })
  .then((data) => {
    console.log(data[0].name);
  });
```

Catching errors

This brings us to the last piece: how do we handle errors? The `fetch()` API can throw an error for many reasons (for example, because there was no network connectivity or the URL was malformed in some way) and we are throwing an error ourselves if the server returned an error.

In the last article, we saw that error handling can get very difficult with nested callbacks, making us handle errors at every nesting level.

To support error handling, `Promise` objects provide a `catch()` method. This is a lot like `then()`: you call it and pass in a handler function. However, while the handler passed to `then()` is called when the asynchronous operation *succeeds*, the handler passed to `catch()` is called when the asynchronous operation *fails*.

If you add `catch()` to the end of a promise chain, then it will be called when any of the asynchronous function calls fail. So you can implement an operation as several consecutive asynchronous function calls, and have a single place to handle all errors.

Try this version of our `fetch()` code. We've added an error handler using `catch()`, and also modified the URL so the request will fail.

JS

```
const fetchPromise = fetch(
  "bad-scheme://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);

fetchPromise
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    return response.json();
  })
  .then((data) => {
    console.log(data[0].name);
  })
  .catch((error) => {
    console.error(`Could not get products: ${error}`);
  });
```

Try running this version: you should see the error logged by our `catch()` handler.

Promise terminology

Promises come with some quite specific terminology that it's worth getting clear about.

First, a promise can be in one of three states:

- **pending**: the promise has been created, and the asynchronous function it's associated with has not succeeded or failed yet. This is the state your promise is in when it's returned from a call to `fetch()`, and the request is still being made.
- **fulfilled**: the asynchronous function has succeeded. When a promise is fulfilled, its `then()` handler is called.
- **rejected**: the asynchronous function has failed. When a promise is rejected, its `catch()` handler is called.

Note that what "succeeded" or "failed" means here is up to the API in question. For example, `fetch()` rejects the returned promise if (among other reasons) a network error prevented the request being sent, but fulfills the promise if the server sent a response, even if the response was an error like 404 Not Found.

Sometimes, we use the term **settled** to cover both **fulfilled** and **rejected**.

A promise is **resolved** if it is settled, or if it has been "locked in" to follow the state of another promise.

The article Let's talk about how to talk about promises gives a great explanation of the details of this terminology.

Combining multiple promises

The promise chain is what you need when your operation consists of several asynchronous functions, and you need each one to complete before starting the next one. But there are other ways you might need to combine asynchronous function calls, and the `Promise` API provides some helpers for them.

Sometimes, you need all the promises to be fulfilled, but they don't depend on each other. In a case like that, it's much more efficient to start them all off together, then be notified when they have all fulfilled. The `Promise.all()` method is what you need here. It takes an array of promises and returns a single promise.

The promise returned by `Promise.all()` is:

- fulfilled when and if *all* the promises in the array are fulfilled. In this case, the `then()` handler is called with an array of all the responses, in the same order that the promises were passed into `all()`.
- rejected when and if *any* of the promises in the array are rejected. In this case, the `catch()` handler is called with the error thrown by the promise that rejected.

For example:

JS

```
const fetchPromise1 = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);
const fetchPromise2 = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/not-found",
);
const fetchPromise3 = fetch(
  "https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json",
);

Promise.all([fetchPromise1, fetchPromise2, fetchPromise3])
  .then((responses) => {
    for (const response of responses) {
      console.log(`${response.url}: ${response.status}`);
    }
  })
  .catch((error) => {
    console.error(`Failed to fetch: ${error}`);
  });
```

Here, we're making three `fetch()` requests to three different URLs. If they all succeed, we will log the response status of each one. If any of them fail, then we're logging the failure.

With the URLs we've provided, all the requests should be fulfilled, although for the second, the server will return `404` (Not Found) instead of `200` (OK) because the requested file does not exist. So the output should be:

```
https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json: 200
https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/not-found: 404
https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json: 200
```

If we try the same code with a badly formed URL, like this:

JS

```
const fetchPromise1 = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);
const fetchPromise2 = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/not-found",
);
const fetchPromise3 = fetch(
  "bad-scheme://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json",
);

Promise.all([fetchPromise1, fetchPromise2, fetchPromise3])
  .then((responses) => {
    for (const response of responses) {
      console.log(`${response.url}: ${response.status}`);
```

```
    }
  })
  .catch((error) => {
    console.error(`Failed to fetch: ${error}`);
  });
```

Then we can expect the `catch()` handler to run, and we should see something like:

```
Failed to fetch: TypeError: Failed to fetch
```

Sometimes, you might need any one of a set of promises to be fulfilled, and don't care which one. In that case, you want `Promise.any()`. This is like `Promise.all()`, except that it is fulfilled as soon as any of the array of promises is fulfilled, or rejected if all of them are rejected:

JS

```
const fetchPromise1 = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);
const fetchPromise2 = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/not-found",
);
const fetchPromise3 = fetch(
  "https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json",
);

Promise.any([fetchPromise1, fetchPromise2, fetchPromise3])
  .then((response) => {
    console.log(`${response.url}: ${response.status}`);
  })
  .catch((error) => {
    console.error(`Failed to fetch: ${error}`);
  });
```

Note that in this case we can't predict which fetch request will complete first.

These are just two of the extra `Promise` functions for combining multiple promises. To learn about the rest, see the `Promise` reference documentation.

async and await

The `async` keyword gives you a simpler way to work with asynchronous promise-based code. Adding `async` at the start of a function makes it an async function:

JS

```
async function myFunction() {
  // This is an async function
}
```

Inside an async function, you can use the `await` keyword before a call to a function that returns a promise. This makes the code wait at that point until the promise is settled, at which point the fulfilled value of the promise is treated as a return value, or the rejected value is thrown.

This enables you to write code that uses asynchronous functions but looks like synchronous code. For example, we could use it to rewrite our fetch example:

JS

```
async function fetchProducts() {
  try {
    // after this line, our function will wait for the `fetch()` call to be settled
    // the `fetch()` call will either return a Response or throw an error
    const response = await fetch(
      "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
    );
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    // after this line, our function will wait for the `response.json()` call to be settled
    // the `response.json()` call will either return the parsed JSON object or throw an error
    const data = await response.json();
    console.log(data[0].name);
  } catch (error) {
    console.error(`Could not get products: ${error}`);
  }
}

fetchProducts();
```

Here, we are calling `await fetch()`, and instead of getting a `Promise`, our caller gets back a fully complete `Response` object, just as if `fetch()` were a synchronous function!

We can even use a `try...catch` block for error handling, exactly as we would if the code were synchronous.

Note though that async functions always return a promise, so you can't do something like:

JS

```
async function fetchProducts() {
  try {
    const response = await fetch(
      "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
    );
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    const data = await response.json();
    return data;
  } catch (error) {
    console.error(`Could not get products: ${error}`);
  }
}

const promise = fetchProducts();
console.log(promise[0].name); // "promise" is a Promise object, so this will not work
```

Instead, you'd need to do something like:

JS

```
async function fetchProducts() {
  try {
    const response = await fetch(
      "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
    );
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
```

```
    }
    const data = await response.json();
    return data;
  } catch (error) {
    console.error(`Could not get products: ${error}`);
  }
}

const promise = fetchProducts();
promise.then((data) => console.log(data[0].name));
```

Also, note that you can only use `await` inside an `async` function, unless your code is in a JavaScript module. That means you can't do this in a normal script:

JS

```
try {
  // using await outside an async function is only allowed in a module
  const response = await fetch(
    "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
  );
  if (!response.ok) {
    throw new Error(`HTTP error: ${response.status}`);
  }
  const data = await response.json();
  console.log(data[0].name);
} catch (error) {
  console.error(`Could not get products: ${error}`);
}
```

You'll probably use `async` functions a lot where you might otherwise use promise chains, and they make working with promises much more intuitive.

Keep in mind that just like a promise chain, `await` forces asynchronous operations to be completed in series. This is necessary if the result of the next operation depends on the result of the last one, but if that's not the case then something like `Promise.all()` will be more performant.

Conclusion

Promises are the foundation of asynchronous programming in modern JavaScript. They make it easier to express and reason about sequences of asynchronous operations without deeply nested callbacks, and they support a style of error handling that is similar to the synchronous `try...catch` statement.

The `async` and `await` keywords make it easier to build an operation from a series of consecutive asynchronous function calls, avoiding the need to create explicit promise chains, and allowing you to write code that looks just like synchronous code.

Promises work in the latest versions of all modern browsers; the only place where promise support will be a problem is in Opera Mini and IE11 and earlier versions.

We didn't touch on all features of promises in this article, just the most interesting and useful ones. As you start to learn more about promises, you'll come across more features and techniques.

Many modern Web APIs are promise-based, including WebRTC, Web Audio API, Media Capture and Streams API, and many more.

See also

- `Promise()`

- Using promises
- We have a problem with promises by Nolan Lawson
- Let's talk about how to talk about promises

Help improve MDN

Was this page helpful to you?

Yes No

Learn how to contribute.

This page was last modified on Jan 1, 2024 by MDN contributors.

How to implement a promise-based API

In the last article we discussed how to use APIs that return promises. In this article we'll look at the other side — how to *implement* APIs that return promises. This is a much less common task than using promise-based APIs, but it's still worth knowing about.

Prerequisites:	A reasonable understanding of JavaScript fundamentals, including event handling and the basics of promises.
Objective:	To understand how to implement promise-based APIs.

Generally, when you implement a promise-based API, you'll be wrapping an asynchronous operation, which might use events, or plain callbacks, or a message-passing model. You'll arrange for a `Promise` object to handle the success or failure of that operation properly.

Implementing an alarm() API

In this example we'll implement a promise-based alarm API, called `alarm()`. It will take as arguments the name of the person to wake up and a delay in milliseconds to wait before waking the person up. After the delay, the function will send a "Wake up!" message, including the name of the person we need to wake up.

Wrapping setTimeout()

We'll use the `setTimeout()` API to implement our `alarm()` function. The `setTimeout()` API takes as arguments a callback function and a delay, given in milliseconds. When `setTimeout()` is called, it starts a timer set to the given delay, and when the time expires, it calls the given function.

In the example below, we call `setTimeout()` with a callback function and a delay of 1000 milliseconds:

HTML Play

```
<button id="set-alarm">Set alarm</button>
<div id="output"></div>
```

JS Play

```
const output = document.querySelector("#output");
const button = document.querySelector("#set-alarm");

function setAlarm() {
  setTimeout(() => {
    output.textContent = "Wake up!";
  }, 1000);
}

button.addEventListener("click", setAlarm);
```

Play

The Promise() constructor

Our `alarm()` function will return a `Promise` that is fulfilled when the timer expires. It will pass a "Wake up!" message into the `then()` handler, and will reject the promise if the caller supplies a negative delay value.

The key component here is the `Promise()` constructor. The `Promise()` constructor takes a single function as an argument. We'll call this function the `executor`. When you create a new promise you supply the implementation of the executor.

This executor function itself takes two arguments, which are both also functions, and which are conventionally called `resolve` and `reject`. In your executor implementation, you call the underlying asynchronous function. If the asynchronous function succeeds, you call `resolve`, and if it fails, you call `reject`. If the executor function throws an error, `reject` is called automatically. You can pass a single parameter of any type into `resolve` and `reject`.

So we can implement `alarm()` like this:

```
JS
function alarm(person, delay) {
  return new Promise((resolve, reject) => {
    if (delay < 0) {
      throw new Error("Alarm delay must not be negative");
    }
    setTimeout(() => {
      resolve(`Wake up, ${person}!`);
    }, delay);
  });
}
```

This function creates and returns a new `Promise`. Inside the executor for the promise, we:

- check that `delay` is not negative, and throw an error if it is.
- call `setTimeout()`, passing a callback and `delay`. The callback will be called when the timer expires, and in the callback we call `resolve`, passing in our `"Wake up!"` message.

Using the alarm() API

This part should be quite familiar from the last article. We can call `alarm()`, and on the returned promise call `then()` and `catch()` to set handlers for promise fulfillment and rejection.

```
JS                                                                    Play
const name = document.querySelector("#name");
const delay = document.querySelector("#delay");
const button = document.querySelector("#set-alarm");
const output = document.querySelector("#output");

function alarm(person, delay) {
  return new Promise((resolve, reject) => {
    if (delay < 0) {
      throw new Error("Alarm delay must not be negative");
    }
    setTimeout(() => {
      resolve(`Wake up, ${person}!`);
    }, delay);
  });
}

button.addEventListener("click", () => {
  alarm(name.value, delay.value)
```

```
    .then((message) => (output.textContent = message))
    .catch((error) => (output.textContent = `Couldn't set alarm: ${error}`));
});
```

Play

Try setting different values for "Name" and "Delay". Try setting a negative value for "Delay".

Using async and await with the alarm() API

Since `alarm()` returns a `Promise`, we can do everything with it that we could do with any other promise: promise chaining, `Promise.all()`, and `async` / `await`:

JS Play

```
const name = document.querySelector("#name");
const delay = document.querySelector("#delay");
const button = document.querySelector("#set-alarm");
const output = document.querySelector("#output");

function alarm(person, delay) {
  return new Promise((resolve, reject) => {
    if (delay < 0) {
      throw new Error("Alarm delay must not be negative");
    }
    setTimeout(() => {
      resolve(`Wake up, ${person}!`);
    }, delay);
  });
}

button.addEventListener("click", async () => {
  try {
    const message = await alarm(name.value, delay.value);
    output.textContent = message;
  } catch (error) {
    output.textContent = `Couldn't set alarm: ${error}`;
  }
});
```

Play

See also

- `Promise()` constructor
- Using promises

Help improve MDN

Was this page helpful to you?

Yes No

Learn how to contribute.

This page was last modified on Jan 1, 2024 by MDN contributors.

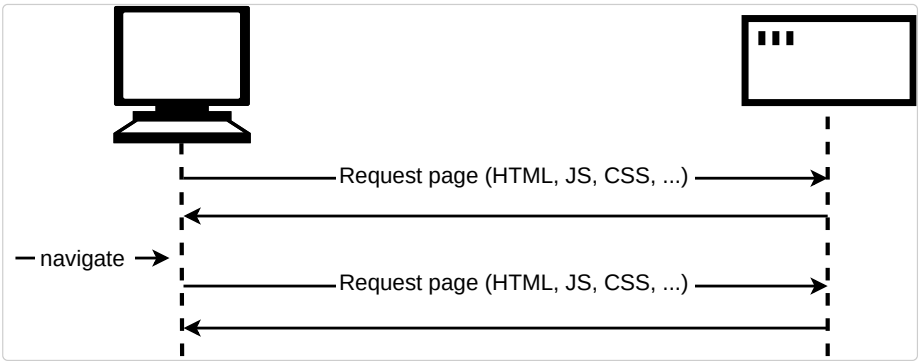
Fetching data from the server

Another very common task in modern websites and applications is retrieving individual data items from the server to update sections of a webpage without having to load an entire new page. This seemingly small detail has had a huge impact on the performance and behavior of sites, so in this article, we'll explain the concept and look at technologies that make it possible: in particular, the Fetch API.

Prerequisites:	JavaScript basics (see first steps, building blocks, JavaScript objects), the basics of Client-side APIs
Objective:	To learn how to fetch data from the server and use it to update the contents of a web page.

What is the problem here?

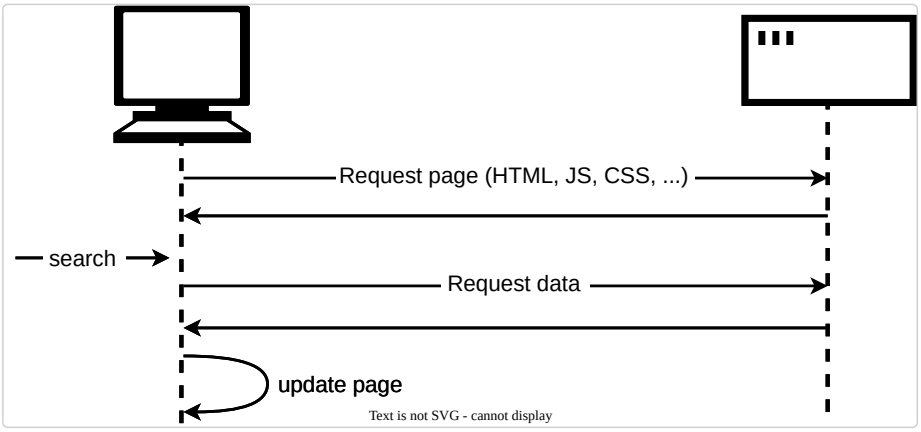
A web page consists of an HTML page and (usually) various other files, such as stylesheets, scripts, and images. The basic model of page loading on the Web is that your browser makes one or more HTTP requests to the server for the files needed to display the page, and the server responds with the requested files. If you visit another page, the browser requests the new files, and the server responds with them.



This model works perfectly well for many sites. But consider a website that's very data-driven. For example, a library website like the Vancouver Public Library . Among other things you could think of a site like this as a user interface to a database. It might let you search for a particular genre of book, or might show you recommendations for books you might like, based on books you've previously borrowed. When you do this, it needs to update the page with the new set of books to display. But note that most of the page content — including items like the page header, sidebar, and footer — stays the same.

The trouble with the traditional model here is that we'd have to fetch and load the entire page, even when we only need to update one part of it. This is inefficient and can result in a poor user experience.

So instead of the traditional model, many websites use JavaScript APIs to request data from the server and update the page content without a page load. So when the user searches for a new product, the browser only requests the data which is needed to update the page — the set of new books to display, for instance.



The main API here is the Fetch API. This enables JavaScript running in a page to make an HTTP request to a server to retrieve specific resources. When the server provides them, the JavaScript can use the data to update the page, typically by using DOM manipulation APIs. The data requested is often JSON, which is a good format for transferring structured data, but can also be HTML or just text.

This is a common pattern for data-driven sites such as Amazon, YouTube, eBay, and so on. With this model:

- Page updates are a lot quicker and you don't have to wait for the page to refresh, meaning that the site feels faster and more responsive.
- Less data is downloaded on each update, meaning less wasted bandwidth. This may not be such a big issue on a desktop on a broadband connection, but it's a major issue on mobile devices and in countries that don't have ubiquitous fast internet service.

> **Note:** In the early days, this general technique was known as Asynchronous JavaScript and XML (Ajax), because it tended to request XML data. This is normally not the case these days (you'd be more likely to request JSON), but the result is still the same, and the term "Ajax" is still often used to describe the technique.

To speed things up even further, some sites also store assets and data on the user's computer when they are first requested, meaning that on subsequent visits they use the local versions instead of downloading fresh copies every time the page is first loaded. The content is only reloaded from the server when it has been updated.

The Fetch API

Let's walk through a couple of examples of the Fetch API.

Fetching text content

For this example, we'll request data out of a few different text files and use them to populate a content area.

This series of files will act as our fake database; in a real application, we'd be more likely to use a server-side language like PHP, Python, or Node to request our data from a database. Here, however, we want to keep it simple and concentrate on the client-side part of this.

To begin this example, make a local copy of fetch-start.html and the four text files — verse1.txt , verse2.txt , verse3.txt , and verse4.txt — in a new directory on your computer. In this example, we will fetch a different verse of the poem (which you may well recognize) when it's selected in the drop-down menu.

Just inside the `<script>` element, add the following code. This stores references to the `<select>` and `<pre>` elements and adds a listener to the `<select>` element, so that when the user selects a new value, the new value is passed to the function named

`updateDisplay()` as a parameter.

JS

```
const verseChoose = document.querySelector("select");
const poemDisplay = document.querySelector("pre");

verseChoose.addEventListener("change", () => {
  const verse = verseChoose.value;
  updateDisplay(verse);
});
```

Let's define our `updateDisplay()` function. First of all, put the following beneath your previous code block — this is the empty shell of the function.

JS

```
function updateDisplay(verse) {

}
```

We'll start our function by constructing a relative URL pointing to the text file we want to load, as we'll need it later. The value of the `<select>` element at any time is the same as the text inside the selected `<option>` (unless you specify a different value in a value attribute) — so for example "Verse 1". The corresponding verse text file is "verse1.txt", and is in the same directory as the HTML file, therefore just the file name will do.

However, web servers tend to be case-sensitive, and the file name doesn't have a space in it. To convert "Verse 1" to "verse1.txt" we need to convert the 'V' to lower case, remove the space, and add ".txt" on the end. This can be done with `replace()`, `toLowerCase()`, and template literal. Add the following lines inside your `updateDisplay()` function:

JS

```
verse = verse.replace(" ", "").toLowerCase();
const url = `${verse}.txt`;
```

Finally we're ready to use the Fetch API:

JS

```
// Call `fetch()`, passing in the URL.
fetch(url)
  // fetch() returns a promise. When we have received a response from the server,
  // the promise's `then()` handler is called with the response.
  .then((response) => {
    // Our handler throws an error if the request did not succeed.
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    // Otherwise (if the response succeeded), our handler fetches the response
    // as text by calling response.text(), and immediately returns the promise
    // returned by `response.text()`.
    return response.text();
  })
  // When response.text() has succeeded, the `then()` handler is called with
  // the text, and we copy it into the `poemDisplay` box.
  .then((text) => {
    poemDisplay.textContent = text;
  })
  // Catch any errors that might happen, and display a message
  // in the `poemDisplay` box.
  .catch((error) => {
```

```
    poemDisplay.textContent = `Could not fetch verse: ${error}`;
  });
```

There's quite a lot to unpack in here.

First, the entry point to the Fetch API is a global function called `fetch()`, that takes the URL as a parameter (it takes another optional parameter for custom settings, but we're not using that here).

Next, `fetch()` is an asynchronous API which returns a `Promise`. If you don't know what that is, read the module on asynchronous JavaScript, and in particular the article on promises, then come back here. You'll find that article also talks about the `fetch()` API!

So because `fetch()` returns a promise, we pass a function into the `then()` method of the returned promise. This method will be called when the HTTP request has received a response from the server. In the handler, we check that the request succeeded, and throw an error if it didn't. Otherwise, we call `response.text()`, to get the response body as text.

It turns out that `response.text()` is *also* asynchronous, so we return the promise it returns, and pass a function into the `then()` method of this new promise. This function will be called when the response text is ready, and inside it we will update our `<pre>` block with the text.

Finally, we chain a `catch()` handler at the end, to catch any errors thrown in either of the asynchronous functions we called or their handlers.

One problem with the example as it stands is that it won't show any of the poem when it first loads. To fix this, add the following two lines at the bottom of your code (just above the closing `</script>` tag) to load verse 1 by default, and make sure the `<select>` element always shows the correct value:

```
JS
updateDisplay("Verse 1");
verseChoose.value = "Verse 1";
```

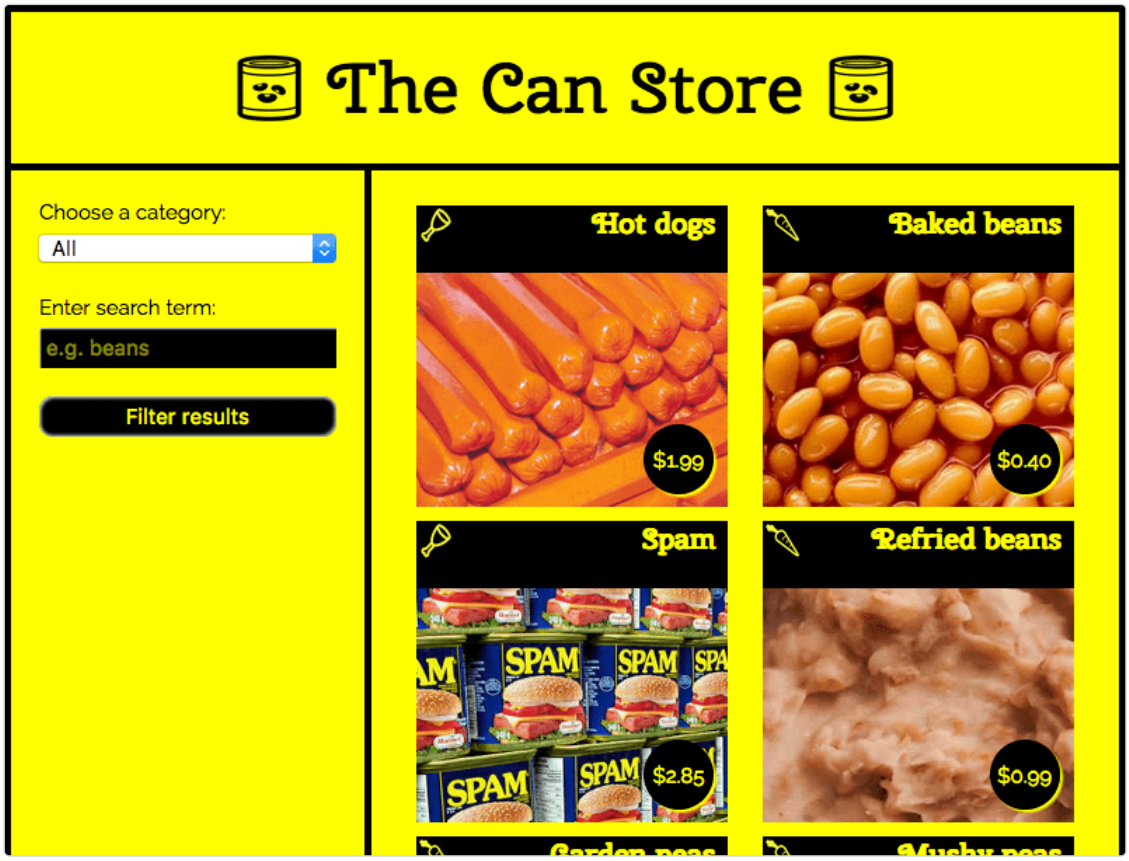
Serving your example from a server

Modern browsers will not run HTTP requests if you just run the example from a local file. This is because of security restrictions (for more on web security, read Website security).

To get around this, we need to test the example by running it through a local web server. To find out how to do this, read our guide to setting up a local testing server.

The can store

In this example we have created a sample site called The Can Store — it's a fictional supermarket that only sells canned goods. You can find this example live on GitHub , and see the source code .



By default, the site displays all the products, but you can use the form controls in the left-hand column to filter them by category, or search term, or both.

There is quite a lot of complex code that deals with filtering the products by category and search terms, manipulating strings so the data displays correctly in the UI, etc. We won't discuss all of it in the article, but you can find extensive comments in the code (see can-script.js).

We will, however, explain the Fetch code.

The first block that uses Fetch can be found at the start of the JavaScript:

JS

```
fetch("products.json")
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    return response.json();
  })
  .then((json) => initialize(json))
  .catch((err) => console.error(`Fetch problem: ${err.message}`));
```

The `fetch()` function returns a promise. If this completes successfully, the function inside the first `.then()` block contains the `response` returned from the network.

Inside this function we:

- check that the server didn't return an error (such as `404 Not Found`). If it did, we throw the error.
- call `json()` on the response. This will retrieve the data as a JSON object. We return the promise returned by `response.json()`.

Next we pass a function into the `then()` method of that returned promise. This function will be passed an object containing the response data as JSON, which we pass into the `initialize()` function. This function which starts the process of displaying all the products in the user interface.

To handle errors, we chain a `.catch()` block onto the end of the chain. This runs if the promise fails for some reason. Inside it, we include a function that is passed as a parameter, an `err` object. This `err` object can be used to report the nature of the error that has occurred, in this case we do it with a simple `console.error()`.

However, a complete website would handle this error more gracefully by displaying a message on the user's screen and perhaps offering options to remedy the situation, but we don't need anything more than a simple `console.error()`.

You can test the failure case yourself:

1. Make a local copy of the example files.
2. Run the code through a web server (as described above, in Serving your example from a server).
3. Modify the path to the file being fetched, to something like 'produc.json' (make sure it is misspelled).
4. Now load the index file in your browser (via `localhost:8000`) and look in your browser developer console. You'll see a message similar to "Fetch problem: HTTP error: 404".

The second Fetch block can be found inside the `fetchBlob()` function:

JS

```
fetch(url)
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    return response.blob();
  })
  .then((blob) => showProduct(blob, product))
  .catch((err) => console.error(`Fetch problem: ${err.message}`));
```

This works in much the same way as the previous one, except that instead of using `json()`, we use `blob()`. In this case we want to return our response as an image file, and the data format we use for that is Blob (the term is an abbreviation of "Binary Large Object" and can basically be used to represent large file-like objects, such as images or video files).

Once we've successfully received our blob, we pass it into our `showProduct()` function, which displays it.

The XMLHttpRequest API

Sometimes, especially in older code, you'll see another API called `XMLHttpRequest` (often abbreviated as "XHR") used to make HTTP requests. This predated Fetch, and was really the first API widely used to implement AJAX. We recommend you use Fetch if you can: it's a simpler API and has more features than `XMLHttpRequest`. We won't go through an example that uses `XMLHttpRequest`, but we will show you what the `XMLHttpRequest` version of our first can store request would look like:

JS

```
const request = new XMLHttpRequest();

try {
  request.open("GET", "products.json");

  request.responseType = "json";
```

```
  request.addEventListener("load", () => initialize(request.response));
  request.addEventListener("error", () => console.error("XHR error"));

  request.send();
} catch (error) {
  console.error(`XHR error ${request.status}`);
}
```

There are five stages to this:

1. Create a new `XMLHttpRequest` object.
2. Call its `open()` method to initialize it.
3. Add an event listener to its `load` event, which fires when the response has completed successfully. In the listener we call `initialize()` with the data.
4. Add an event listener to its `error` event, which fires when the request encounters an error
5. Send the request.

We also have to wrap the whole thing in the try...catch block, to handle any errors thrown by `open()` or `send()`.

Hopefully you think the Fetch API is an improvement over this. In particular, see how we have to handle errors in two different places.

Summary

This article shows how to start working with Fetch to fetch data from the server.

See also

There are however a lot of different subjects discussed in this article, which has only really scratched the surface. For a lot more detail on these subjects, try the following articles:

- Using Fetch
- Promises
- Working with JSON data
- An overview of HTTP
- Server-side website programming

Introducing JavaScript objects

In JavaScript, most things are objects, from core JavaScript features like arrays to the browser APIs built on top of JavaScript. You can even create your own objects to encapsulate related functions and variables into efficient packages and act as handy data containers. The object-based nature of JavaScript is important to understand if you want to go further with your knowledge of the language, therefore we've provided this module to help you. Here we teach object theory and syntax in detail, then look at how to create your own objects.

Prerequisites

Before starting this module, you should have some familiarity with HTML and CSS. You are advised to work through the Introduction to HTML and Introduction to CSS modules before starting on JavaScript.

You should also have some familiarity with JavaScript basics before looking at JavaScript objects in detail. Before attempting this module, work through JavaScript first steps and JavaScript building blocks.

> **Note:** If you are working on a computer/tablet/other devices where you are not able to create your own files, you could try out (most of) the code examples in an online coding program such as JSBin or Glitch .

Guides

Object basics

In the first article looking at JavaScript objects, we'll look at fundamental JavaScript object syntax, and revisit some JavaScript features we've already looked at earlier on in the course, reiterating the fact that many of the features you've already dealt with are in fact objects.

Object prototypes

Prototypes are the mechanism by which JavaScript objects inherit features from one another, and they work differently from inheritance mechanisms in classical object-oriented programming languages. In this article, we explore how prototype chains work.

Object-oriented programming

In this article, we'll describe some of the basic principles of "classical" object-oriented programming, and look at the ways it is different from the prototype model in JavaScript.

Classes in JavaScript

JavaScript provides some features for people wanting to implement "classical" object-oriented programs, and in this article, we'll describe these features.

Working with JSON data

JavaScript Object Notation (JSON) is a standard text-based format for representing structured data based on JavaScript object syntax, which is commonly used for representing and transmitting data on the web (i.e., sending some data from the server to the client, so it can be displayed on a web page). You'll come across it quite often, so in this article, we give you all you need to work with JSON using JavaScript, including parsing the JSON so you can access data items within it, and writing your own JSON.

[Object building practice]

In previous articles we looked at all the essential JavaScript object theory and syntax details, giving you a solid base to start from. In this article we dive into a practical exercise, giving you some more practice in building custom JavaScript objects, which produce something fun and colorful — some colored bouncing balls.

Assessments

[Adding features to our bouncing balls demo]

In this assessment, you are expected to use the bouncing balls demo from the previous article as a starting point, and add some new and interesting features to it.

See also

[Learn JavaScript]

An excellent resource for aspiring web developers — Learn JavaScript in an interactive environment, with short lessons and interactive tests, guided by automated assessment. The first 40 lessons are free, and the complete course is available for a small one-time payment.

JavaScript object basics

In this article, we'll look at fundamental JavaScript object syntax, and revisit some JavaScript features that we've already seen earlier in the course, reiterating the fact that many of the features you've already dealt with are objects.

Prerequisites:	A basic understanding of HTML and CSS, familiarity with JavaScript basics (see First steps and Building blocks).
Objective:	To understand the basics of working with objects in JavaScript: creating objects, accessing and modifying object properties, and using constructors.

Object basics

An object is a collection of related data and/or functionality. These usually consist of several variables and functions (which are called properties and methods when they are inside objects). Let's work through an example to understand what they look like.

To begin with, make a local copy of our oojs.html file. This contains very little — a `<script>` element for us to write our source code into. We'll use this as a basis for exploring basic object syntax. While working with this example you should have your developer tools JavaScript console open and ready to type in some commands.

As with many things in JavaScript, creating an object often begins with defining and initializing a variable. Try entering the following line below the JavaScript code that's already in your file, then saving and refreshing:

JS

```
const person = {};
```

Now open your browser's JavaScript console, enter `person` into it, and press `Enter`/`Return`. You should get a result similar to one of the below lines:

```
[object Object]
Object { }
{ }
```

Congratulations, you've just created your first object. Job done! But this is an empty object, so we can't really do much with it. Let's update the JavaScript object in our file to look like this:

JS

```
const person = {
  name: ["Bob", "Smith"],
  age: 32,
  bio: function () {
    console.log(`${this.name[0]} ${this.name[1]} is ${this.age} years old.`);
  },
  introduceSelf: function () {
    console.log(`Hi! I'm ${this.name[0]}.`);
  },
};
```

After saving and refreshing, try entering some of the following into the JavaScript console on your browser devtools:

```
JS
person.name;
person.name[0];
person.age;
person.bio();
// "Bob Smith is 32 years old."
person.introduceSelf();
// "Hi! I'm Bob."
```

You have now got some data and functionality inside your object, and are now able to access them with some nice simple syntax!

So what is going on here? Well, an object is made up of multiple members, each of which has a name (e.g. `name` and `age` above), and a value (e.g. `['Bob', 'Smith']` and `32`). Each name/value pair must be separated by a comma, and the name and value in each case are separated by a colon. The syntax always follows this pattern:

```
JS
const objectName = {
  member1Name: member1Value,
  member2Name: member2Value,
  member3Name: member3Value,
};
```

The value of an object member can be pretty much anything — in our person object we've got a number, an array, and two functions. The first two items are data items, and are referred to as the object's **properties**. The last two items are functions that allow the object to do something with that data, and are referred to as the object's **methods**.

When the object's members are functions there's a simpler syntax. Instead of `bio: function ()` we can write `bio()`. Like this:

```
JS
const person = {
  name: ["Bob", "Smith"],
  age: 32,
  bio() {
    console.log(`${this.name[0]} ${this.name[1]} is ${this.age} years old.`);
  },
  introduceSelf() {
    console.log(`Hi! I'm ${this.name[0]}.`);
  },
};
```

From now on, we'll use this shorter syntax.

An object like this is referred to as an **object literal** — we've literally written out the object contents as we've come to create it. This is different compared to objects instantiated from classes, which we'll look at later on.

It is very common to create an object using an object literal when you want to transfer a series of structured, related data items in some manner, for example sending a request to the server to be put into a database. Sending a single object is much more efficient than sending several items individually, and it is easier to work with than an array, when you want to identify individual items by name.

Dot notation

Above, you accessed the object's properties and methods using **dot notation**. The object name (person) acts as the **namespace** — it must be entered first to access anything inside the object. Next you write a dot, then the item you want to access — this can be the name of a simple property, an item of an array property, or a call to one of the object's methods, for example:

```
JS
person.age;
person.bio();
```

Objects as object properties

An object property can itself be an object. For example, try changing the `name` member from

```
JS
const person = {
  name: ["Bob", "Smith"],
};
```

to

```
JS
const person = {
  name: {
    first: "Bob",
    last: "Smith",
  },
  // …
};
```

To access these items you just need to chain the extra step onto the end with another dot. Try these in the JS console:

```
JS
person.name.first;
person.name.last;
```

If you do this, you'll also need to go through your method code and change any instances of

```
JS
name[0];
name[1];
```

to

```
JS
name.first;
name.last;
```

Otherwise, your methods will no longer work.

Bracket notation

Bracket notation provides an alternative way to access object properties. Instead of using dot notation like this:

```
JS
person.age;
person.name.first;
```

You can instead use square brackets:

JS

```
person["age"];
person["name"]["first"];
```

This looks very similar to how you access the items in an array, and it is basically the same thing — instead of using an index number to select an item, you are using the name associated with each member's value. It is no wonder that objects are sometimes called **associative arrays** — they map strings to values in the same way that arrays map numbers to values.

Dot notation is generally preferred over bracket notation because it is more succinct and easier to read. However there are some cases where you have to use square brackets. For example, if an object property name is held in a variable, then you can't use dot notation to access the value, but you can access the value using bracket notation.

In the example below, the `logProperty()` function can use `person[propertyName]` to retrieve the value of the property named in `propertyName`.

JS

```
const person = {
  name: ["Bob", "Smith"],
  age: 32,
};

function logProperty(propertyName) {
  console.log(person[propertyName]);
}

logProperty("name");
// ["Bob", "Smith"]
logProperty("age");
// 32
```

Setting object members

So far we've only looked at retrieving (or **getting**) object members — you can also **set** (update) the value of object members by declaring the member you want to set (using dot or bracket notation), like this:

JS

```
person.age = 45;
person["name"]["last"] = "Cratchit";
```

Try entering the above lines, and then getting the members again to see how they've changed, like so:

JS

```
person.age;
person["name"]["last"];
```

Setting members doesn't just stop at updating the values of existing properties and methods; you can also create completely new members. Try these in the JS console:

JS

```
person["eyes"] = "hazel";
person.farewell = function () {
  console.log("Bye everybody!");
};
```

You can now test out your new members:

```
JS
person["eyes"];
person.farewell();
// "Bye everybody!"
```

One useful aspect of bracket notation is that it can be used to set not only member values dynamically, but member names too. Let's say we wanted users to be able to store custom value types in their people data, by typing the member name and value into two text inputs. We could get those values like this:

```
JS
const myDataName = nameInput.value;
const myDataValue = nameValue.value;
```

We could then add this new member name and value to the `person` object like this:

```
JS
person[myDataName] = myDataValue;
```

To test this, try adding the following lines into your code, just below the closing curly brace of the `person` object:

```
JS
const myDataName = "height";
const myDataValue = "1.75m";
person[myDataName] = myDataValue;
```

Now try saving and refreshing, and entering the following into your text input:

```
JS
person.height;
```

Adding a property to an object using the method above isn't possible with dot notation, which can only accept a literal member name, not a variable value pointing to a name.

What is "this"?

You may have noticed something slightly strange in our methods. Look at this one for example:

```
JS
introduceSelf() {
  console.log(`Hi! I'm ${this.name[0]}.`);
}
```

You are probably wondering what "this" is. The `this` keyword refers to the current object the code is being written inside — so in this case `this` is equivalent to `person`. So why not just write `person` instead?

Well, when you only have to create a single object literal, it's not so useful. But if you create more than one, `this` enables you to use the same method definition for every object you create.

Let's illustrate what we mean with a simplified pair of person objects:

```
JS
const person1 = {
  name: "Chris",
```

```
  introduceSelf() {
    console.log(`Hi! I'm ${this.name}.`);
  },
};

const person2 = {
  name: "Deepti",
  introduceSelf() {
    console.log(`Hi! I'm ${this.name}.`);
  },
};
```

In this case, `person1.introduceSelf()` outputs "Hi! I'm Chris."; `person2.introduceSelf()` on the other hand outputs "Hi! I'm Deepti.", even though the method's code is exactly the same in each case. This isn't hugely useful when you are writing out object literals by hand, but it will be essential when we start using **constructors** to create more than one object from a single object definition, and that's the subject of the next section.

Introducing constructors

Using object literals is fine when you only need to create one object, but if you have to create more than one, as in the previous section, they're seriously inadequate. We have to write out the same code for every object we create, and if we want to change some properties of the object - like adding a `height` property - then we have to remember to update every object.

We would like a way to define the "shape" of an object — the set of methods and the properties it can have — and then create as many objects as we like, just updating the values for the properties that are different.

The first version of this is just a function:

```
JS
function createPerson(name) {
  const obj = {};
  obj.name = name;
  obj.introduceSelf = function () {
    console.log(`Hi! I'm ${this.name}.`);
  };
  return obj;
}
```

This function creates and returns a new object each time we call it. The object will have two members:

- a property `name`
- a method `introduceSelf()`.

Note that `createPerson()` takes a parameter `name` to set the value of the `name` property, but the value of the `introduceSelf()` method will be the same for all objects created using this function. This is a very common pattern for creating objects.

Now we can create as many objects as we like, reusing the definition:

```
JS
const salva = createPerson("Salva");
salva.introduceSelf();
// "Hi! I'm Salva."

const frankie = createPerson("Frankie");
```

```
frankie.introduceSelf();
// "Hi! I'm Frankie."
```

This works fine but is a bit long-winded: we have to create an empty object, initialize it, and return it. A better way is to use a **constructor**. A constructor is just a function called using the `new` keyword. When you call a constructor, it will:

- create a new object
- bind `this` to the new object, so you can refer to `this` in your constructor code
- run the code in the constructor
- return the new object.

Constructors, by convention, start with a capital letter and are named for the type of object they create. So we could rewrite our example like this:

JS

```
function Person(name) {
  this.name = name;
  this.introduceSelf = function () {
    console.log(`Hi! I'm ${this.name}.`);
  };
}
```

To call `Person()` as a constructor, we use `new`:

JS

```
const salva = new Person("Salva");
salva.introduceSelf();
// "Hi! I'm Salva."

const frankie = new Person("Frankie");
frankie.introduceSelf();
// "Hi! I'm Frankie."
```

You've been using objects all along

As you've been going through these examples, you have probably been thinking that the dot notation you've been using is very familiar. That's because you've been using it throughout the course! Every time we've been working through an example that uses a built-in browser API or JavaScript object, we've been using objects, because such features are built using exactly the same kind of object structures that we've been looking at here, albeit more complex ones than in our own basic custom examples.

So when you used string methods like:

JS

```
myString.split(",");
```

You were using a method available on a `String` object. Every time you create a string in your code, that string is automatically created as an instance of `String`, and therefore has several common methods and properties available on it.

When you accessed the document object model using lines like this:

JS

```
const myDiv = document.createElement("div");
const myVideo = document.querySelector("video");
```

You were using methods available on a `Document` object. For each webpage loaded, an instance of `Document` is created, called `document`, which represents the entire page's structure, content, and other features such as its URL. Again, this means that it has several common methods and properties available on it.

The same is true of pretty much any other built-in object or API you've been using — `Array`, `Math`, and so on.

Note that built in objects and APIs don't always create object instances automatically. As an example, the Notifications API — which allows modern browsers to fire system notifications — requires you to instantiate a new object instance using the constructor for each notification you want to fire. Try entering the following into your JavaScript console:

```
JS
const myNotification = new Notification("Hello!");
```

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: Object basics.

Summary

Congratulations, you've reached the end of our first JS objects article — you should now have a good idea of how to work with objects in JavaScript — including creating your own simple objects. You should also appreciate that objects are very useful as structures for storing related data and functionality — if you tried to keep track of all the properties and methods in our `person` object as separate variables and functions, it would be inefficient and frustrating, and we'd run the risk of clashing with other variables and functions that have the same names. Objects let us keep the information safely locked away in their own package, out of harm's way.

In the next article we'll look at **prototypes**, which is the fundamental way that JavaScript lets an object inherit properties from other objects.

Object prototypes

Prototypes are the mechanism by which JavaScript objects inherit features from one another. In this article, we explain what a prototype is, how prototype chains work, and how a prototype for an object can be set.

Prerequisites:	Understanding JavaScript functions, familiarity with JavaScript basics (see First steps and Building blocks), and OOJS basics (see Introduction to objects).
Objective:	To understand JavaScript object prototypes, how prototype chains work, and how to set the prototype of an object.

The prototype chain

In the browser's console, try creating an object literal:

```
JS
const myObject = {
  city: "Madrid",
  greet() {
    console.log(`Greetings from ${this.city}`);
  },
};

myObject.greet(); // Greetings from Madrid
```

This is an object with one data property, `city`, and one method, `greet()`. If you type the object's name *followed by a period* into the console, like `myObject.`, then the console will pop up a list of all the properties available to this object. You'll see that as well as `city` and `greet`, there are lots of other properties!

```
__defineGetter__
__defineSetter__
__lookupGetter__
__lookupSetter__
__proto__
city
constructor
greet
hasOwnProperty
isPrototypeOf
propertyIsEnumerable
toLocaleString
toString
valueOf
```

Try accessing one of them:

```
JS
myObject.toString(); // "[object Object]"
```

It works (even if it's not obvious what `toString()` does).

What are these extra properties, and where do they come from?

Every object in JavaScript has a built-in property, which is called its **prototype**. The prototype is itself an object, so the prototype will have its own prototype, making what's called a **prototype chain**. The chain ends when we reach a prototype that has `null` for its own prototype.

> **Note:** The property of an object that points to its prototype is **not** called `prototype`. Its name is not standard, but in practice all browsers use `__proto__`. The standard way to access an object's prototype is the `Object.getPrototypeOf()` method.

When you try to access a property of an object: if the property can't be found in the object itself, the prototype is searched for the property. If the property still can't be found, then the prototype's prototype is searched, and so on until either the property is found, or the end of the chain is reached, in which case `undefined` is returned.

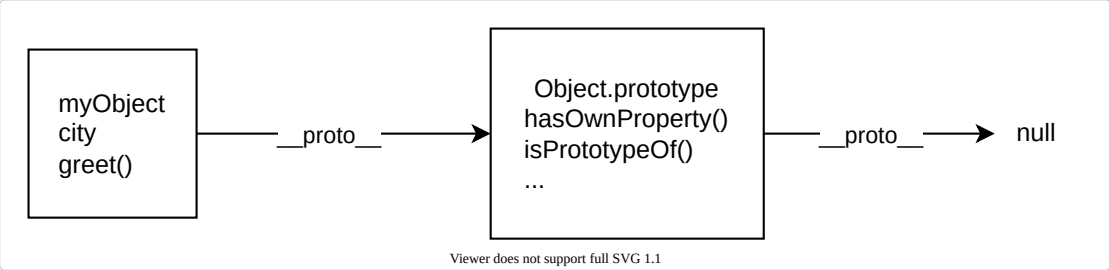
So when we call `myObject.toString()`, the browser:

- looks for `toString` in `myObject`
- can't find it there, so looks in the prototype object of `myObject` for `toString`
- finds it there, and calls it.

What is the prototype for `myObject`? To find out, we can use the function `Object.getPrototypeOf()`:

```
JS
Object.getPrototypeOf(myObject); // Object { }
```

This is an object called `Object.prototype`, and it is the most basic prototype, that all objects have by default. The prototype of `Object.prototype` is `null`, so it's at the end of the prototype chain:



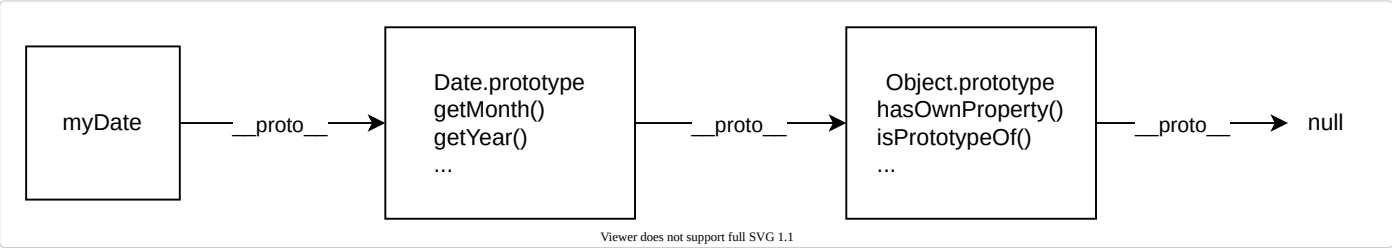
The prototype of an object is not always `Object.prototype`. Try this:

```
JS
const myDate = new Date();
let object = myDate;

do {
  object = Object.getPrototypeOf(object);
  console.log(object);
} while (object);

// Date.prototype
// Object { }
// null
```

This code creates a `Date` object, then walks up the prototype chain, logging the prototypes. It shows us that the prototype of `myDate` is a `Date.prototype` object, and the prototype of *that* is `Object.prototype`.



In fact, when you call familiar methods, like `myDate2.getMonth()`, you are calling a method that's defined on `Date.prototype`.

Shadowing properties

What happens if you define a property in an object, when a property with the same name is defined in the object's prototype? Let's see:

```
JS
const myDate = new Date(1995, 11, 17);

console.log(myDate.getYear()); // 95

myDate.getYear = function () {
  console.log("something else!");
};

myDate.getYear(); // 'something else!'
```

This should be predictable, given the description of the prototype chain. When we call `getYear()` the browser first looks in `myDate` for a property with that name, and only checks the prototype if `myDate` does not define it. So when we add `getYear()` to `myDate`, then the version in `myDate` is called.

This is called "shadowing" the property.

Setting a prototype

There are various ways of setting an object's prototype in JavaScript, and here we'll describe two: `Object.create()` and constructors.

Using Object.create

The `Object.create()` method creates a new object and allows you to specify an object that will be used as the new object's prototype.

Here's an example:

```
JS
const personPrototype = {
  greet() {
    console.log("hello!");
  },
};

const carl = Object.create(personPrototype);
carl.greet(); // hello!
```

Here we create an object `personPrototype`, which has a `greet()` method. We then use `Object.create()` to create a new object with `personPrototype` as its prototype. Now we can call `greet()` on the new object, and the prototype provides its implementation.

Using a constructor

In JavaScript, all functions have a property named `prototype`. When you call a function as a constructor, this property is set as the prototype of the newly constructed object (by convention, in the property named `__proto__`).

So if we set the `prototype` of a constructor, we can ensure that all objects created with that constructor are given that prototype:

JS

```
const personPrototype = {
  greet() {
    console.log(`hello, my name is ${this.name}!`);
  },
};

function Person(name) {
  this.name = name;
}

Object.assign(Person.prototype, personPrototype);
// or
// Person.prototype.greet = personPrototype.greet;
```

Here we create:

- an object `personPrototype`, which has a `greet()` method
- a `Person()` constructor function which initializes the name of the person to create.

We then put the methods defined in `personPrototype` onto the `Person` function's `prototype` property using Object.assign.

After this code, objects created using `Person()` will get `Person.prototype` as their prototype, which automatically contains the `greet` method.

JS

```
const reuben = new Person("Reuben");
reuben.greet(); // hello, my name is Reuben!
```

This also explains why we said earlier that the prototype of `myDate` is called `Date.prototype`: it's the `prototype` property of the `Date` constructor.

Own properties

The objects we create using the `Person` constructor above have two properties:

- a `name` property, which is set in the constructor, so it appears directly on `Person` objects
- a `greet()` method, which is set in the prototype.

It's common to see this pattern, in which methods are defined on the prototype, but data properties are defined in the constructor. That's because methods are usually the same for every object we create, while we often want each object to have its own value for its data properties (just as here where every person has a different name).

Properties that are defined directly in the object, like `name` here, are called **own properties**, and you can check whether a property is an own property using the static `Object.hasOwn()` method:

```
JS
const irma = new Person("Irma");

console.log(Object.hasOwn(irma, "name")); // true
console.log(Object.hasOwn(irma, "greet")); // false
```

> **Note:** You can also use the non-static `Object.hasOwnProperty()` method here, but we recommend that you use `Object.hasOwn()` if you can.

Prototypes and inheritance

Prototypes are a powerful and very flexible feature of JavaScript, making it possible to reuse code and combine objects.

In particular they support a version of **inheritance**. Inheritance is a feature of object-oriented programming languages that lets programmers express the idea that some objects in a system are more specialized versions of other objects.

For example, if we're modeling a school, we might have *professors* and *students*: they are both *people*, so have some features in common (for example, they both have names), but each might add extra features (for example, professors have a subject that they teach), or might implement the same feature in different ways. In an OOP system we might say that professors and students both **inherit from** people.

You can see how in JavaScript, if `Professor` and `Student` objects can have `Person` prototypes, then they can inherit the common properties, while adding and redefining those properties which need to differ.

In the next article we'll discuss inheritance along with the other main features of object-oriented programming languages, and see how JavaScript supports them.

Summary

This article has covered JavaScript object prototypes, including how prototype object chains allow objects to inherit features from one another, the prototype property and how it can be used to add methods to constructors, and other related topics.

In the next article we'll look at the concepts underlying object-oriented programming.

Object-oriented programming

Object-oriented programming (OOP) is a programming paradigm fundamental to many programming languages, including Java and C++. In this article, we'll provide an overview of the basic concepts of OOP. We'll describe three main concepts: **classes and instances**, **inheritance**, and **encapsulation**. For now, we'll describe these concepts without reference to JavaScript in particular, so all the examples are given in pseudocode.

> **Note:** To be precise, the features described here are of a particular style of OOP called **class-based** or "classical" OOP. When people talk about OOP, this is generally the type that they mean.

After that, in JavaScript, we'll look at how constructors and the prototype chain relate to these OOP concepts, and how they differ. In the next article, we'll look at some additional features of JavaScript that make it easier to implement object-oriented programs.

Prerequisites:	Understanding JavaScript functions, familiarity with JavaScript basics (see First steps and Building blocks), and OOJS basics (see Introduction to objects and Object prototypes).
Objective:	To understand the basic concepts of class-based object-oriented programming.

Object-oriented programming is about modeling a system as a collection of objects, where each object represents some particular aspect of the system. Objects contain both functions (or methods) and data. An object provides a public interface to other code that wants to use it but maintains its own private, internal state; other parts of the system don't have to care about what is going on inside the object.

Classes and instances

When we model a problem in terms of objects in OOP, we create abstract definitions representing the types of objects we want to have in our system. For example, if we were modeling a school, we might want to have objects representing professors. Every professor has some properties in common: they all have a name and a subject that they teach. Additionally, every professor can do certain things: they can all grade a paper and they can introduce themselves to their students at the start of the year, for example.

So `Professor` could be a **class** in our system. The definition of the class lists the data and methods that every professor has.

In pseudocode, a `Professor` class could be written like this:

```
class Professor
    properties
        name
        teaches
    methods
        grade(paper)
        introduceSelf()
```

This defines a `Professor` class with:

- two data properties: `name` and `teaches`
- two methods: `grade()` to grade a paper and `introduceSelf()` to introduce themselves.

On its own, a class doesn't do anything: it's a kind of template for creating concrete objects of that type. Each concrete professor we create is called an **instance** of the `Professor` class. The process of creating an instance is performed by a special function called a **constructor**. We pass values to the constructor for any internal state that we want to initialize in the new instance.

Generally, the constructor is written out as part of the class definition, and it usually has the same name as the class itself:

```
class Professor
    properties
        name
        teaches
    constructor
        Professor(name, teaches)
    methods
        grade(paper)
        introduceSelf()
```

This constructor takes two parameters, so we can initialize the `name` and `teaches` properties when we create a new concrete professor.

Now that we have a constructor, we can create some professors. Programming languages often use the keyword `new` to signal that a constructor is being called.

```
JS
walsh = new Professor("Walsh", "Psychology");
lillian = new Professor("Lillian", "Poetry");

walsh.teaches; // 'Psychology'
walsh.introduceSelf(); // 'My name is Professor Walsh and I will be your Psychology professor.'

lillian.teaches; // 'Poetry'
lillian.introduceSelf(); // 'My name is Professor Lillian and I will be your Poetry professor.'
```

This creates two objects, both instances of the `Professor` class.

Inheritance

Suppose in our school we also want to represent students. Unlike professors, students can't grade papers, don't teach a particular subject, and belong to a particular year.

However, students do have a name and may also want to introduce themselves, so we might write out the definition of a student class like this:

```
class Student
    properties
        name
        year
    constructor
        Student(name, year)
    methods
        introduceSelf()
```

It would be helpful if we could represent the fact that students and professors share some properties, or more accurately, the fact that on some level, they are the *same kind of thing*. **Inheritance** lets us do this.

We start by observing that students and professors are both people, and people have names and want to introduce themselves. We can model this by defining a new class `Person`, where we define all the common properties of people. Then, `Professor` and `Student` can both **derive** from `Person`, adding their extra properties:

```
class Person
    properties
        name
    constructor
        Person(name)
    methods
        introduceSelf()

class Professor : extends Person
    properties
        teaches
    constructor
        Professor(name, teaches)
    methods
        grade(paper)
        introduceSelf()

class Student : extends Person
    properties
        year
    constructor
        Student(name, year)
    methods
        introduceSelf()
```

In this case, we would say that `Person` is the **superclass** or **parent class** of both `Professor` and `Student`. Conversely, `Professor` and `Student` are **subclasses** or **child classes** of `Person`.

You might notice that `introduceSelf()` is defined in all three classes. The reason for this is that while all people want to introduce themselves, the way they do so is different:

JS

```
walsh = new Professor("Walsh", "Psychology");
walsh.introduceSelf(); // 'My name is Professor Walsh and I will be your Psychology professor.'

summers = new Student("Summers", 1);
summers.introduceSelf(); // 'My name is Summers and I'm in the first year.'
```

We might have a default implementation of `introduceSelf()` for people who aren't students *or* professors:

JS

```
pratt = new Person("Pratt");
pratt.introduceSelf(); // 'My name is Pratt.'
```

This feature - when a method has the same name but a different implementation in different classes - is called **polymorphism**. When a method in a subclass replaces the superclass's implementation, we say that the subclass **overrides** the version in the superclass.

Encapsulation

Objects provide an interface to other code that wants to use them but maintain their own internal state. The object's internal state is kept **private**, meaning that it can only be accessed by the object's own methods, not from other objects. Keeping an object's

internal state private, and generally making a clear division between its public interface and its private internal state, is called **encapsulation**.

This is a useful feature because it enables the programmer to change the internal implementation of an object without having to find and update all the code that uses it: it creates a kind of firewall between this object and the rest of the system.

For example, suppose students are allowed to study archery if they are in the second year or above. We could implement this just by exposing the student's `year` property, and other code could examine that to decide whether the student can take the course:

```
JS
if (student.year > 1) {
  // allow the student into the class
}
```

The problem is, if we decide to change the criteria for allowing students to study archery - for example by also requiring the parent or guardian to give their permission - we'd need to update every place in our system that performs this test. It would be better to have a `canStudyArchery()` method on `Student` objects, that implements the logic in one place:

```
class Student : extends Person
    properties
       year
    constructor
       Student(name, year)
    methods
       introduceSelf()
       canStudyArchery() { return this.year > 1 }
```

```
JS
if (student.canStudyArchery()) {
  // allow the student into the class
}
```

That way, if we want to change the rules about studying archery, we only have to update the `Student` class, and all the code using it will still work.

In many OOP languages, we can prevent other code from accessing an object's internal state by marking some properties as `private`. This will generate an error if code outside the object tries to access them:

```
class Student : extends Person
    properties
       private year
    constructor
        Student(name, year)
    methods
       introduceSelf()
       canStudyArchery() { return this.year > 1 }

student = new Student('Weber', 1)
student.year // error: 'year' is a private property of Student
```

In languages that don't enforce access like this, programmers use naming conventions, such as starting the name with an underscore, to indicate that the property should be considered private.

OOP and JavaScript

In this article, we've described some of the basic features of class-based object-oriented programming as implemented in languages like Java and C++.

In the two previous articles, we looked at a couple of core JavaScript features: constructors and prototypes. These features certainly have some relation to some of the OOP concepts described above.

- **constructors** in JavaScript provide us with something like a class definition, enabling us to define the "shape" of an object, including any methods it contains, in a single place. But prototypes can be used here, too. For example, if a method is defined on a constructor's `prototype` property, then all objects created using that constructor get that method via their prototype, and we don't need to define it in the constructor.
- **the prototype chain** seems like a natural way to implement inheritance. For example, if we can have a `Student` object whose prototype is `Person`, then it can inherit `name` and override `introduceSelf()`.

But it's worth understanding the differences between these features and the "classical" OOP concepts described above. We'll highlight a couple of them here.

First, in class-based OOP, classes and objects are two separate constructs, and objects are always created as instances of classes. Also, there is a distinction between the feature used to define a class (the class syntax itself) and the feature used to instantiate an object (a constructor). In JavaScript, we can and often do create objects without any separate class definition, either using a function or an object literal. This can make working with objects much more lightweight than it is in classical OOP.

Second, although a prototype chain looks like an inheritance hierarchy and behaves like it in some ways, it's different in others. When a subclass is instantiated, a single object is created which combines properties defined in the subclass with properties defined further up the hierarchy. With prototyping, each level of the hierarchy is represented by a separate object, and they are linked together via the `__proto__` property. The prototype chain's behavior is less like inheritance and more like **delegation**. Delegation is a programming pattern where an object, when asked to perform a task, can perform the task itself or ask another object (its **delegate**) to perform the task on its behalf. In many ways, delegation is a more flexible way of combining objects than inheritance (for one thing, it's possible to change or completely replace the delegate at run time).

That said, constructors and prototypes can be used to implement class-based OOP patterns in JavaScript. But using them directly to implement features like inheritance is tricky, so JavaScript provides extra features, layered on top of the prototype model, that map more directly to the concepts of class-based OOP. These extra features are the subject of the next article.

Summary

This article has described the basic features of class-based object oriented programming, and briefly looked at how JavaScript constructors and prototypes compare with these concepts.

In the next article, we'll look at the features JavaScript provides to support class-based object-oriented programming.

Classes in JavaScript

In the last article, we introduced some basic concepts of object-oriented programming (OOP), and discussed an example where we used OOP principles to model professors and students in a school.

We also talked about how it's possible to use prototypes and constructors to implement a model like this, and that JavaScript also provides features that map more closely to classical OOP concepts.

In this article, we'll go through these features. It's worth keeping in mind that the features described here are not a new way of combining objects: under the hood, they still use prototypes. They're just a way to make it easier to set up a prototype chain.

Prerequisites:	A basic understanding of HTML and CSS, familiarity with JavaScript basics (see First steps and Building blocks) and OOJS basics (see Introduction to objects, Object prototypes, and Object-oriented programming).
Objective:	To understand how to use the features JavaScript provides to implement "classical" object-oriented programs.

Classes and constructors

You can declare a class using the `class` keyword. Here's a class declaration for our `Person` from the previous article:

JS

```
class Person {
  name;

  constructor(name) {
    this.name = name;
  }

  introduceSelf() {
    console.log(`Hi! I'm ${this.name}`);
  }
}
```

This declares a class called `Person`, with:

- a `name` property.
- a constructor that takes a `name` parameter that is used to initialize the new object's `name` property
- an `introduceSelf()` method that can refer to the object's properties using `this`.

The `name;` declaration is optional: you could omit it, and the line `this.name = name;` in the constructor will create the `name` property before initializing it. However, listing properties explicitly in the class declaration might make it easier for people reading your code to see which properties are part of this class.

You could also initialize the property to a default value when you declare it, with a line like `name = '';`.

The constructor is defined using the `constructor` keyword. Just like a constructor outside a class definition, it will:

- create a new object

- bind `this` to the new object, so you can refer to `this` in your constructor code
- run the code in the constructor
- return the new object.

Given the class declaration code above, you can create and use a new `Person` instance like this:

JS

```
const giles = new Person("Giles");

giles.introduceSelf(); // Hi! I'm Giles
```

Note that we call the constructor using the name of the class, `Person` in this example.

Omitting constructors

If you don't need to do any special initialization, you can omit the constructor, and a default constructor will be generated for you:

JS

```
class Animal {
  sleep() {
    console.log("zzzzzzz");
  }
}

const spot = new Animal();

spot.sleep(); // 'zzzzzzz'
```

Inheritance

Given our `Person` class above, let's define the `Professor` subclass.

JS

```
class Professor extends Person {
  teaches;

  constructor(name, teaches) {
    super(name);
    this.teaches = teaches;
  }

  introduceSelf() {
    console.log(
      `My name is ${this.name}, and I will be your ${this.teaches} professor.`,
    );
  }

  grade(paper) {
    const grade = Math.floor(Math.random() * (5 - 1) + 1);
    console.log(grade);
  }
}
```

We use the `extends` keyword to say that this class inherits from another class.

The `Professor` class adds a new property `teaches`, so we declare that.

Since we want to set `teaches` when a new `Professor` is created, we define a constructor, which takes the `name` and `teaches` as arguments. The first thing this constructor does is call the superclass constructor using `super()`, passing up the `name` parameter. The superclass constructor takes care of setting `name`. After that, the `Professor` constructor sets the `teaches` property.

> **Note:** If a subclass has any of its own initialization to do, it **must** first call the superclass constructor using `super()`, passing up any parameters that the superclass constructor is expecting.

We've also overridden the `introduceSelf()` method from the superclass, and added a new method `grade()`, to grade a paper (our professor isn't very good, and just assigns random grades to papers).

With this declaration we can now create and use professors:

```
JS
const walsh = new Professor("Walsh", "Psychology");
walsh.introduceSelf(); // 'My name is Walsh, and I will be your Psychology professor'

walsh.grade("my paper"); // some random grade
```

Encapsulation

Finally, let's see how to implement encapsulation in JavaScript. In the last article we discussed how we would like to make the `year` property of `Student` private, so we could change the rules about archery classes without breaking any code that uses the `Student` class.

Here's a declaration of the `Student` class that does just that:

```
JS
class Student extends Person {
  #year;

  constructor(name, year) {
    super(name);
    this.#year = year;
  }

  introduceSelf() {
    console.log(`Hi! I'm ${this.name}, and I'm in year ${this.#year}.`);
  }

  canStudyArchery() {
    return this.#year > 1;
  }
}
```

In this class declaration, `#year` is a private data property. We can construct a `Student` object, and it can use `#year` internally, but if code outside the object tries to access `#year` the browser throws an error:

```
JS
const summers = new Student("Summers", 2);

summers.introduceSelf(); // Hi! I'm Summers, and I'm in year 2.
summers.canStudyArchery(); // true

summers.#year; // SyntaxError
```

> **Note:** Code run in the Chrome console can access private properties outside the class. This is a DevTools-only relaxation of the JavaScript syntax restriction.

Private data properties must be declared in the class declaration, and their names start with `#`.

Private methods

You can have private methods as well as private data properties. Just like private data properties, their names start with `#`, and they can only be called by the object's own methods:

```
JS
class Example {
  somePublicMethod() {
    this.#somePrivateMethod();
  }

  #somePrivateMethod() {
    console.log("You called me?");
  }
}

const myExample = new Example();

myExample.somePublicMethod(); // 'You called me?'

myExample.#somePrivateMethod(); // SyntaxError
```

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: Object-oriented JavaScript.

Summary

In this article, we've gone through the main tools available in JavaScript for writing object-oriented programs. We haven't covered everything here, but this should be enough to get you started. Our article on Classes is a good place to learn more.

Working with JSON

JavaScript Object Notation (JSON) is a standard text-based format for representing structured data based on JavaScript object syntax. It is commonly used for transmitting data in web applications (e.g., sending some data from the server to the client, so it can be displayed on a web page, or vice versa). You'll come across it quite often, so in this article, we give you all you need to work with JSON using JavaScript, including parsing JSON so you can access data within it, and creating JSON.

Prerequisites:	A basic understanding of HTML and CSS, familiarity with JavaScript basics (see First steps and Building blocks) and OOJS basics (see Introduction to objects).
Objective:	To understand how to work with data stored in JSON, and create your own JSON strings.

No, really, what is JSON?

JSON is a text-based data format following JavaScript object syntax, which was popularized by Douglas Crockford . Even though it closely resembles JavaScript object literal syntax, it can be used independently from JavaScript, and many programming environments feature the ability to read (parse) and generate JSON.

JSON exists as a string — useful when you want to transmit data across a network. It needs to be converted to a native JavaScript object when you want to access the data. This is not a big issue — JavaScript provides a global JSON object that has methods available for converting between the two.

> **Note:** Converting a string to a native object is called *deserialization*, while converting a native object to a string so it can be transmitted across the network is called *serialization*.

A JSON string can be stored in its own file, which is basically just a text file with an extension of `.json`, and a MIME type of `application/json`.

JSON structure

As described above, JSON is a string whose format very much resembles JavaScript object literal format. You can include the same basic data types inside JSON as you can in a standard JavaScript object — strings, numbers, arrays, booleans, and other object literals. This allows you to construct a data hierarchy, like so:

JSON

```
{
  "squadName": "Super hero squad",
  "homeTown": "Metro City",
  "formed": 2016,
  "secretBase": "Super tower",
  "active": true,
  "members": [
    {
      "name": "Molecule Man",
      "age": 29,
      "secretIdentity": "Dan Jukes",
      "powers": ["Radiation resistance", "Turning tiny", "Radiation blast"]
    },
    {
```

```
      "name": "Madame Uppercut",
      "age": 39,
      "secretIdentity": "Jane Wilson",
      "powers": [
        "Million tonne punch",
        "Damage resistance",
        "Superhuman reflexes"
      ]
    },
    {
      "name": "Eternal Flame",
      "age": 1000000,
      "secretIdentity": "Unknown",
      "powers": [
        "Immortality",
        "Heat Immunity",
        "Inferno",
        "Teleportation",
        "Interdimensional travel"
      ]
    }
  ]
}
```

If we loaded this string into a JavaScript program and parsed it into a variable called `superHeroes` for example, we could then access the data inside it using the same dot/bracket notation we looked at in the JavaScript object basics article. For example:

JS

```
superHeroes.homeTown;
superHeroes["active"];
```

To access data further down the hierarchy, you have to chain the required property names and array indexes together. For example, to access the third superpower of the second hero listed in the members list, you'd do this:

JS

```
superHeroes["members"][1]["powers"][2];
```

1. First, we have the variable name — `superHeroes`.
2. Inside that, we want to access the `members` property, so we use `["members"]`.
3. `members` contains an array populated by objects. We want to access the second object inside the array, so we use `[1]`.
4. Inside this object, we want to access the `powers` property, so we use `["powers"]`.
5. Inside the `powers` property is an array containing the selected hero's superpowers. We want the third one, so we use `[2]`.

> **Note:** We've made the JSON seen above available inside a variable in our JSONTest.html example (see the source code). Try loading this up and then accessing data inside the variable via your browser's JavaScript console.

Arrays as JSON

Above we mentioned that JSON text basically looks like a JavaScript object inside a string. We can also convert arrays to/from JSON. Below is also valid JSON, for example:

JSON

```
[
  {
```

```
    "name": "Molecule Man",
    "age": 29,
    "secretIdentity": "Dan Jukes",
    "powers": ["Radiation resistance", "Turning tiny", "Radiation blast"]
  },
  {
    "name": "Madame Uppercut",
    "age": 39,
    "secretIdentity": "Jane Wilson",
    "powers": [
      "Million tonne punch",
      "Damage resistance",
      "Superhuman reflexes"
    ]
  }
]
```

The above is perfectly valid JSON. You'd just have to access array items (in its parsed version) by starting with an array index, for example `[0]["powers"][0]`.

Other notes

- JSON is purely a string with a specified data format — it contains only properties, no methods.
- JSON requires double quotes to be used around strings and property names. Single quotes are not valid other than surrounding the entire JSON string.
- Even a single misplaced comma or colon can cause a JSON file to go wrong, and not work. You should be careful to validate any data you are attempting to use (although computer-generated JSON is less likely to include errors, as long as the generator program is working correctly). You can validate JSON using an application like JSONLint .
- JSON can actually take the form of any data type that is valid for inclusion inside JSON, not just arrays or objects. So for example, a single string or number would be valid JSON.
- Unlike in JavaScript code in which object properties may be unquoted, in JSON only quoted strings may be used as properties.

Active learning: Working through a JSON example

So, let's work through an example to show how we could make use of some JSON formatted data on a website.

Getting started

To begin with, make local copies of our heroes.html and style.css files. The latter contains some simple CSS to style our page, while the former contains some very simple body HTML, plus a `<script>` element to contain the JavaScript code we will be writing in this exercise:

HTML

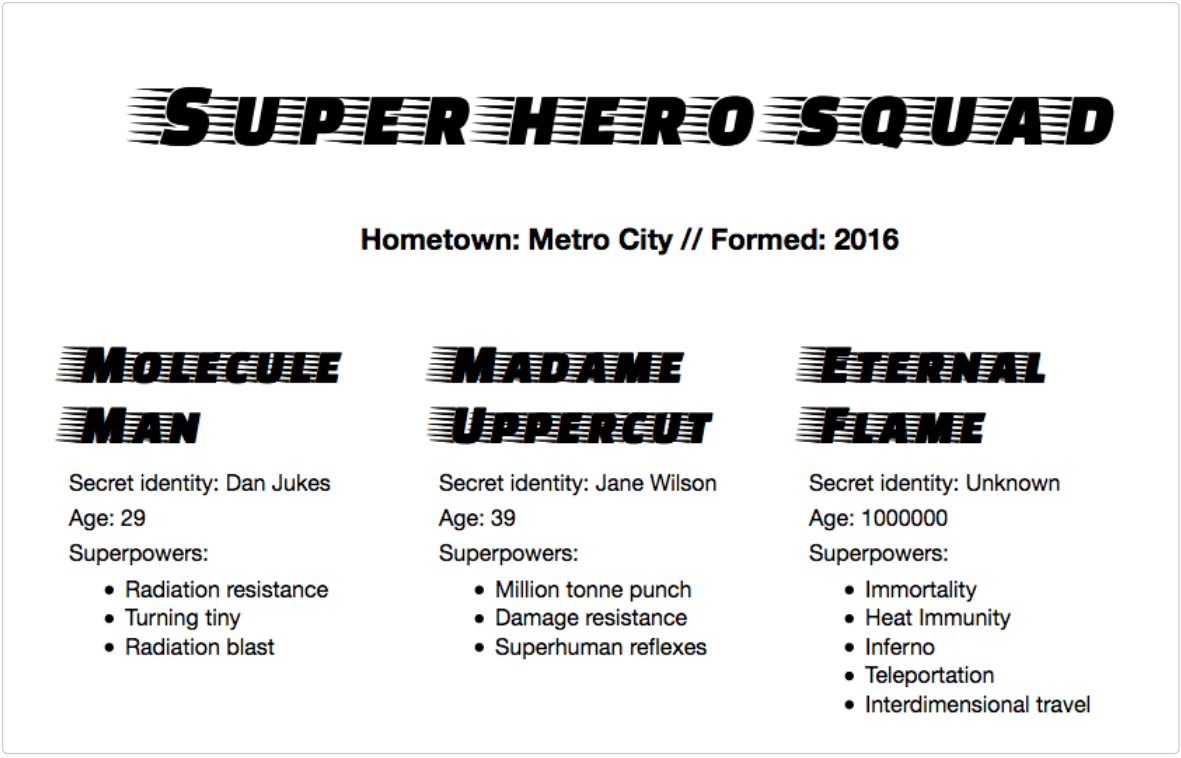
```
<header>
...
</header>

<section>
...
</section>

<script>
...
</script>
```

We have made our JSON data available on our GitHub, at https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json .

We are going to load the JSON into our script, and use some nifty DOM manipulation to display it, like this:



Top-level function

The top-level function looks like this:

```
JS
async function populate() {
  const requestURL =
    "https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json";
  const request = new Request(requestURL);

  const response = await fetch(request);
  const superHeroes = await response.json();

  populateHeader(superHeroes);
  populateHeroes(superHeroes);
}
```

To obtain the JSON, we use an API called Fetch. This API allows us to make network requests to retrieve resources from a server via JavaScript (e.g. images, text, JSON, even HTML snippets), meaning that we can update small sections of content without having to reload the entire page.

In our function, the first four lines use the Fetch API to fetch the JSON from the server:

- we declare the `requestURL` variable to store the GitHub URL
- we use the URL to initialize a new `Request` object.
- we make the network request using the `fetch()` function, and this returns a `Response` object
- we retrieve the response as JSON using the `json()` function of the `Response` object.

> **Note:** The `fetch()` API is **asynchronous**. We'll learn a lot about asynchronous functions in the next module, but for now, we'll just say that we need to add the keyword `async` before the name of the function that uses the fetch API, and add the keyword `await` before the calls to any asynchronous functions.

After all that, the `superHeroes` variable will contain the JavaScript object based on the JSON. We are then passing that object to two function calls — the first one fills the `<header>` with the correct data, while the second one creates an information card for each hero on the team, and inserts it into the `<section>`.

Populating the header

Now that we've retrieved the JSON data and converted it into a JavaScript object, let's make use of it by writing the two functions we referenced above. First of all, add the following function definition below the previous code:

JS

```
function populateHeader(obj) {
  const header = document.querySelector("header");
  const myH1 = document.createElement("h1");
  myH1.textContent = obj.squadName;
  header.appendChild(myH1);

  const myPara = document.createElement("p");
  myPara.textContent = `Hometown: ${obj.homeTown} // Formed: ${obj.formed}`;
  header.appendChild(myPara);
}
```

Here we first create an h1 element with `createElement()`, set its `textContent` to equal the `squadName` property of the object, then append it to the header using `appendChild()`. We then do a very similar operation with a paragraph: create it, set its text content and append it to the header. The only difference is that its text is set to a template literal containing both the `homeTown` and `formed` properties of the object.

Creating the hero information cards

Next, add the following function at the bottom of the code, which creates and displays the superhero cards:

JS

```
function populateHeroes(obj) {
  const section = document.querySelector("section");
  const heroes = obj.members;

  for (const hero of heroes) {
    const myArticle = document.createElement("article");
    const myH2 = document.createElement("h2");
    const myPara1 = document.createElement("p");
    const myPara2 = document.createElement("p");
    const myPara3 = document.createElement("p");
    const myList = document.createElement("ul");

    myH2.textContent = hero.name;
    myPara1.textContent = `Secret identity: ${hero.secretIdentity}`;
    myPara2.textContent = `Age: ${hero.age}`;
    myPara3.textContent = "Superpowers:";

    const superPowers = hero.powers;
    for (const power of superPowers) {
      const listItem = document.createElement("li");
      listItem.textContent = power;
      myList.appendChild(listItem);
```

```
    }

    myArticle.appendChild(myH2);
    myArticle.appendChild(myPara1);
    myArticle.appendChild(myPara2);
    myArticle.appendChild(myPara3);
    myArticle.appendChild(myList);

    section.appendChild(myArticle);
  }
}
```

To start with, we store the `members` property of the JavaScript object in a new variable. This array contains multiple objects that contain the information for each hero.

Next, we use a for...of loop to loop through each object in the array. For each one, we:

1. Create several new elements: an `<article>`, an `<h2>`, three `<p>`s, and a `<ul>`.
2. Set the `<h2>` to contain the current hero's `name`.
3. Fill the three paragraphs with their `secretIdentity`, `age`, and a line saying "Superpowers:" to introduce the information in the list.
4. Store the `powers` property in another new constant called `superPowers` — this contains an array that lists the current hero's superpowers.
5. Use another `for...of` loop to loop through the current hero's superpowers — for each one we create an `<li>` element, put the superpower inside it, then put the `listItem` inside the `<ul>` element (`myList`) using `appendChild()`.
6. The very last thing we do is to append the `<h2>`, `<p>`s, and `<ul>` inside the `<article>` (`myArticle`), then append the `<article>` inside the `<section>`. The order in which things are appended is important, as this is the order they will be displayed inside the HTML.

> **Note:** If you are having trouble getting the example to work, try referring to our heroes-finished.html source code (see it running live also.)

> **Note:** If you are having trouble following the dot/bracket notation we are using to access the JavaScript object, it can help to have the superheroes.json file open in another tab or your text editor, and refer to it as you look at our JavaScript. You should also refer back to our JavaScript object basics article for more information on dot and bracket notation.

Calling the top-level function

Finally, we need to call our top-level `populate()` function:

```
JS
populate();
```

Converting between objects and text

The above example was simple in terms of accessing the JavaScript object, because we converted the network response directly into a JavaScript object using `response.json()`.

But sometimes we aren't so lucky — sometimes we receive a raw JSON string, and we need to convert it to an object ourselves. And when we want to send a JavaScript object across the network, we need to convert it to JSON (a string) before sending it.

Luckily, these two problems are so common in web development that a built-in JSON object is available in browsers, which contains the following two methods:

- `parse()`: Accepts a JSON string as a parameter, and returns the corresponding JavaScript object.
- `stringify()`: Accepts an object as a parameter, and returns the equivalent JSON string.

You can see the first one in action in our heroes-finished-json-parse.html example (see the source code) — this does exactly the same thing as the example we built up earlier, except that:

- we retrieve the response as text rather than JSON, by calling the `text()` method of the response
- we then use `parse()` to convert the text to a JavaScript object.

The key snippet of code is here:

```
JS
async function populate() {
  const requestURL =
    "https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json";
  const request = new Request(requestURL);

  const response = await fetch(request);
  const superHeroesText = await response.text();

  const superHeroes = JSON.parse(superHeroesText);
  populateHeader(superHeroes);
  populateHeroes(superHeroes);
}
```

As you might guess, `stringify()` works the opposite way. Try entering the following lines into your browser's JavaScript console one by one to see it in action:

```
JS
let myObj = { name: "Chris", age: 38 };
myObj;
let myString = JSON.stringify(myObj);
myString;
```

Here we're creating a JavaScript object, then checking what it contains, then converting it to a JSON string using `stringify()` — saving the return value in a new variable — then checking it again.

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: JSON.

Summary

In this article, we've given you a simple guide to using JSON in your programs, including how to create and parse JSON, and how to access data locked inside it. In the next article, we'll begin looking at object-oriented JavaScript.

See also

- JSON reference
- Fetch API overview

- Using Fetch
- HTTP request methods
- Official JSON website with link to ECMA standard

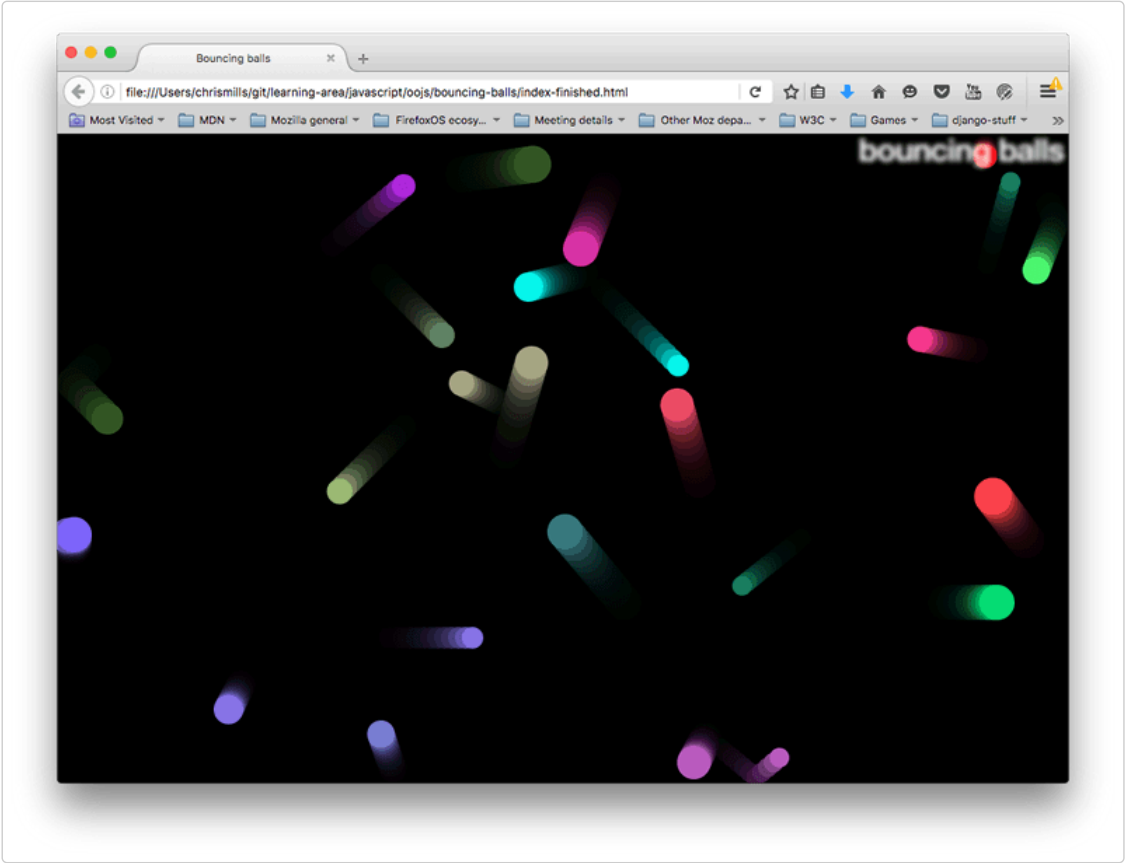
This page was last modified on Jan 1, 2024 by MDN contributors.

In previous articles we looked at all the essential JavaScript object theory and syntax details, giving you a solid base to start from. In this article we dive into a practical exercise, giving you some more practice in building custom JavaScript objects, with a fun and colorful result.

Prerequisites:	A basic understanding of HTML and CSS, familiarity with JavaScript basics (see First steps and Building blocks) and OOJS basics (see Introduction to objects).
Objective:	To get some practice with using objects and object-oriented techniques in a real-world context.

Let's bounce some balls

In this article we will write a classic "bouncing balls" demo, to show you how useful objects can be in JavaScript. Our little balls will bounce around on the screen, and change color when they touch each other. The finished example will look a little something like this:



This example will make use of the Canvas API for drawing the balls to the screen, and the `requestAnimationFrame` API for animating the whole display — you don't need to have any previous knowledge of these APIs, and we hope that by the time you've finished this article you'll be interested in exploring them more. Along the way, we'll make use of some nifty objects, and show you a couple of nice techniques like bouncing balls off walls, and checking whether they have hit each other (otherwise known as *collision detection*).

Getting started

To begin with, make local copies of our `index.html`, `style.css`, and `main.js` files. These contain the following, respectively:

1. A very simple HTML document featuring an h1 element, a `<canvas>` element to draw our balls on, and elements to apply our CSS and JavaScript to our HTML.
2. Some very simple styles, which mainly serve to style and position the `<h1>`, and get rid of any scrollbars or margin around the edge of the page (so that it looks nice and neat).
3. Some JavaScript that serves to set up the `<canvas>` element and provide a general function that we're going to use.

The first part of the script looks like so:

```
JS
const canvas = document.querySelector("canvas");
const ctx = canvas.getContext("2d");

const width = (canvas.width = window.innerWidth);
const height = (canvas.height = window.innerHeight);
```

This script gets a reference to the `<canvas>` element, then calls the `getContext()` method on it to give us a context on which we can start to draw. The resulting constant (`ctx`) is the object that directly represents the drawing area of the canvas and allows us to draw 2D shapes on it.

Next, we set constants called `width` and `height`, and the width and height of the canvas element (represented by the `canvas.width` and `canvas.height` properties) to equal the width and height of the browser viewport (the area which the webpage appears on — this can be gotten from the `Window.innerWidth` and `Window.innerHeight` properties).

Note that we are chaining multiple assignments together, to get the variables all set quicker — this is perfectly OK.

Then we have two helper functions:

```
JS
function random(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}

function randomRGB() {
  return `rgb(${random(0, 255)} ${random(0, 255)} ${random(0, 255)})`;
}
```

The `random()` function takes two numbers as arguments, and returns a random number in the range between the two. The `randomRGB()` function generates a random color represented as an `rgb()` string.

Modeling a ball in our program

Our program will feature lots of balls bouncing around the screen. Since these balls will all behave in the same way, it makes sense to represent them with an object. Let's start by adding the following class definition to the bottom of our code.

```
JS
class Ball {
  constructor(x, y, velX, velY, color, size) {
    this.x = x;
    this.y = y;
    this.velX = velX;
    this.velY = velY;
    this.color = color;
```

```
    this.size = size;
  }
}
```

So far this class only contains a constructor, in which we can initialize the properties each ball needs in order to function in our program:

- `x` and `y` coordinates — the horizontal and vertical coordinates where the ball starts on the screen. This can range between 0 (top left hand corner) to the width and height of the browser viewport (bottom right-hand corner).
- horizontal and vertical velocity (`velX` and `velY`) — each ball is given a horizontal and vertical velocity; in real terms these values are regularly added to the `x` / `y` coordinate values when we animate the balls, to move them by this much on each frame.
- `color` — each ball gets a color.
- `size` — each ball gets a size — this is its radius, in pixels.

This handles the properties, but what about the methods? We want to get our balls to actually do something in our program.

Drawing the ball

First add the following `draw()` method to the `Ball` class:

JS

```
draw() {
  ctx.beginPath();
  ctx.fillStyle = this.color;
  ctx.arc(this.x, this.y, this.size, 0, 2 * Math.PI);
  ctx.fill();
}
```

Using this function, we can tell the ball to draw itself onto the screen, by calling a series of members of the 2D canvas context we defined earlier (`ctx`). The context is like the paper, and now we want to command our pen to draw something on it:

- First, we use `beginPath()` to state that we want to draw a shape on the paper.
- Next, we use `fillStyle` to define what color we want the shape to be — we set it to our ball's `color` property.
- Next, we use the `arc()` method to trace an arc shape on the paper. Its parameters are:
  - The `x` and `y` position of the arc's center — we are specifying the ball's `x` and `y` properties.
  - The radius of the arc — in this case, the ball's `size` property.
  - The last two parameters specify the start and end number of degrees around the circle that the arc is drawn between. Here we specify 0 degrees, and `2 * PI` , which is the equivalent of 360 degrees in radians (annoyingly, you have to specify this in radians). That gives us a complete circle. If you had specified only `1 * PI` , you'd get a semi-circle (180 degrees).
- Last of all, we use the `fill()` method, which basically states "finish drawing the path we started with `beginPath()` , and fill the area it takes up with the color we specified earlier in `fillStyle` ."

You can start testing your object out already.

1. Save the code so far, and load the HTML file in a browser.
2. Open the browser's JavaScript console, and then refresh the page so that the canvas size changes to the smaller visible viewport that remains when the console opens.
3. Type in the following to create a new ball instance:

   JS

```
const testBall = new Ball(50, 100, 4, 4, "blue", 10);
```

4. Try calling its members:

```
testBall.x;
testBall.size;
testBall.color;
testBall.draw();
```

5. When you enter the last line, you should see the ball draw itself somewhere on the canvas.

Updating the ball's data

We can draw the ball in position, but to actually move the ball, we need an update function of some kind. Add the following code inside the class definition for `Ball`:

```
update() {
  if ((this.x + this.size) >= width) {
    this.velX = -(this.velX);
  }

  if ((this.x - this.size) <= 0) {
    this.velX = -(this.velX);
  }

  if ((this.y + this.size) >= height) {
    this.velY = -(this.velY);
  }

  if ((this.y - this.size) <= 0) {
    this.velY = -(this.velY);
  }

  this.x += this.velX;
  this.y += this.velY;
}
```

The first four parts of the function check whether the ball has reached the edge of the canvas. If it has, we reverse the polarity of the relevant velocity to make the ball travel in the opposite direction. So for example, if the ball was traveling upwards (negative `velY`), then the vertical velocity is changed so that it starts to travel downwards instead (positive `velY`).

In the four cases, we are checking to see:

- if the `x` coordinate is greater than the width of the canvas (the ball is going off the right edge).
- if the `x` coordinate is smaller than 0 (the ball is going off the left edge).
- if the `y` coordinate is greater than the height of the canvas (the ball is going off the bottom edge).
- if the `y` coordinate is smaller than 0 (the ball is going off the top edge).

In each case, we include the `size` of the ball in the calculation because the `x`/`y` coordinates are in the center of the ball, but we want the edge of the ball to bounce off the perimeter — we don't want the ball to go halfway off the screen before it starts to bounce back.

The last two lines add the `velX` value to the `x` coordinate, and the `velY` value to the `y` coordinate — the ball is in effect moved each time this method is called.

This will do for now; let's get on with some animation!

Animating the ball

Now let's make this fun. We are now going to start adding balls to the canvas, and animating them.

First, we need to create somewhere to store all our balls and then populate it. The following will do this job — add it to the bottom of your code now:

```
JS
const balls = [];

while (balls.length < 25) {
  const size = random(10, 20);
  const ball = new Ball(
    // ball position always drawn at least one ball width
    // away from the edge of the canvas, to avoid drawing errors
    random(0 + size, width - size),
    random(0 + size, height - size),
    random(-7, 7),
    random(-7, 7),
    randomRGB(),
    size,
  );

  balls.push(ball);
}
```

The `while` loop creates a new instance of our `Ball()` using random values generated with our `random()` and `randomRGB()` functions, then `push()` es it onto the end of our balls array, but only while the number of balls in the array is less than 25. So when we have 25 balls in the array, no more balls will be pushed. You can try varying the number in `balls.length < 25` to get more or fewer balls in the array. Depending on how much processing power your computer/browser has, specifying several thousand balls might slow down the animation rather a lot!

Next, add the following to the bottom of your code:

```
JS
function loop() {
  ctx.fillStyle = "rgb(0 0 0 / 25%)";
  ctx.fillRect(0, 0, width, height);

  for (const ball of balls) {
    ball.draw();
    ball.update();
  }

  requestAnimationFrame(loop);
}
```

All programs that animate things generally involve an animation loop, which serves to update the information in the program and then render the resulting view on each frame of the animation; this is the basis for most games and other such programs. Our `loop()` function does the following:

- Sets the canvas fill color to semi-transparent black, then draws a rectangle of the color across the whole width and height of the canvas, using `fillRect()` (the four parameters provide a start coordinate, and a width and height for the rectangle drawn). This serves to cover up the previous frame's drawing before the next one is drawn. If you don't do this, you'll just see long

snakes worming their way around the canvas instead of balls moving! The color of the fill is set to semi-transparent, `rgb(0 0 0 / 25%)`, to allow the previous few frames to shine through slightly, producing the little trails behind the balls as they move. If you changed 0.25 to 1, you won't see them at all any more. Try varying this number to see the effect it has.

- Loops through all the balls in the `balls` array, and runs each ball's `draw()` and `update()` function to draw each one on the screen, then do the necessary updates to position and velocity in time for the next frame.
- Runs the function again using the `requestAnimationFrame()` method — when this method is repeatedly run and passed the same function name, it runs that function a set number of times per second to create a smooth animation. This is generally done recursively — which means that the function is calling itself every time it runs, so it runs over and over again.

Finally, add the following line to the bottom of your code — we need to call the function once to get the animation started.

JS

```
loop();
```

That's it for the basics — try saving and refreshing to test your bouncing balls out!

Adding collision detection

Now for a bit of fun, let's add some collision detection to our program, so our balls know when they have hit another ball.

First, add the following method definition to your `Ball` class.

JS

```
collisionDetect() {
  for (const ball of balls) {
    if (this !== ball) {
      const dx = this.x - ball.x;
      const dy = this.y - ball.y;
      const distance = Math.sqrt(dx * dx + dy * dy);

      if (distance < this.size + ball.size) {
        ball.color = this.color = randomRGB();
      }
    }
  }
}
```

This method is a little complex, so don't worry if you don't understand exactly how it works for now. An explanation follows:

- For each ball, we need to check every other ball to see if it has collided with the current ball. To do this, we start another `for...of` loop to loop through all the balls in the `balls[]` array.
- Immediately inside the for loop, we use an `if` statement to check whether the current ball being looped through is the same ball as the one we are currently checking. We don't want to check whether a ball has collided with itself! To do this, we check whether the current ball (i.e., the ball whose collisionDetect method is being invoked) is the same as the loop ball (i.e., the ball that is being referred to by the current iteration of the for loop in the collisionDetect method). We then use `!` to negate the check, so that the code inside the `if` statement only runs if they are **not** the same.
- We then use a common algorithm to check the collision of two circles. We are basically checking whether any of the two circle's areas overlap. This is explained further in 2D collision detection.
- If a collision is detected, the code inside the inner `if` statement is run. In this case, we only set the `color` property of both the circles to a new random color. We could have done something far more complex, like get the balls to bounce off each other realistically, but that would have been far more complex to implement. For such physics simulations, developers tend to use a games or physics libraries such as PhysicsJS , matter.js , Phaser , etc.

You also need to call this method in each frame of the animation. Update your `loop()` function to call `ball.collisionDetect()` after `ball.update()`:

JS

```
function loop() {
  ctx.fillStyle = "rgb(0 0 0 / 25%)";
  ctx.fillRect(0, 0, width, height);

  for (const ball of balls) {
    ball.draw();
    ball.update();
    ball.collisionDetect();
  }

  requestAnimationFrame(loop);
}
```

Save and refresh the demo again, and you'll see your balls change color when they collide!

> **Note:** If you have trouble getting this example to work, try comparing your JavaScript code against our finished version (also see it running live).

Summary

We hope you had fun writing your own real-world random bouncing balls example, using various object and object-oriented techniques from throughout the module! This should have given you some useful practice in using objects, and good real-world context.

That's it for object articles — all that remains now is for you to test your skills in the object assessment.

See also

- Canvas tutorial — a beginner's guide to using 2D canvas.
- requestAnimationFrame()
- 2D collision detection
- 3D collision detection
- 2D breakout game using pure JavaScript — a great beginner's tutorial showing how to build a 2D game.
- 2D breakout game using Phaser — explains the basics of building a 2D game using a JavaScript game library.