

**BEGINNER TO PRO**

# **murach's JavaScript and jQuery**

## **4TH EDITION**

### **(Chapter 2)**

Thanks for downloading this chapter from [Murach's JavaScript and jQuery \(4th Edition\)](#). We hope it will show you how easy it is to learn from any Murach book, with its paired-pages presentation, its “how-to” headings, its practical coding examples, and its clear, concise style.

To view the full table of contents for this book, you can go to our [website](#). From there, you can read more about this book, you can find out about any additional downloads that are available, and you can review our other books on related topics.

Thanks for your interest in our books!



**MIKE MURACH & ASSOCIATES, INC.**

1-800-221-5528 • (559) 440-9071 • Fax: (559) 440-0963

[murachbooks@murach.com](mailto:murachbooks@murach.com) • [www.murach.com](http://www.murach.com)

*Copyright © 2020 Mike Murach & Associates. All rights reserved.*

## **What developers have said about previous editions**

---

“If you are new to web design or an old pro like me, this book is a must-have in my opinion. I love how it starts out with the basics and then moves on to the good stuff. Each chapter is full of examples and sample code showing you how to do the most common techniques that you will face as a web developer/designer. This one will be on my desk for a while!”

Shawn Jackson, Web Developer, Colorado

“This book more clearly teaches JavaScript than any other book that I have seen.”

Eric Mortensen, Northeast Ohio Oracle Users Group

“I just finished a pretty heavy application project, the first serious work I have done with jQuery and Ajax. Along the way, I had to deal with preloading images, manipulations of the DOM, tabs, plugins, and Dialogs.... I kept this book at my side throughout the entire project, and it was indispensable to me. The answers were right there at every turn. All the examples made sense to me, and they all worked!”

Alan Vogt, ETL Consultant, Information Builders, Inc.

“What I like about this and other Murach books is that within minutes of opening the book, you are developing hands-on with the technology in question.”

Charles Zimmerman, Developer

“I have several books on JavaScript, but the best one is this one. The text, examples, descriptions, and even the layout all bring you, the learner, an ease of use that is missing in other books.”

Chris Wallace, Denver Visual Studio User Group

# Get started fast with JavaScript

The goal of this chapter is to get you off to a good start with JavaScript, especially if you're new to programming. If you have programming experience with another language, you should be able to move rapidly through this chapter. Otherwise, take it easy and do the exercises at the end of this chapter.

<b>How to include JavaScript in an HTML document .....</b>	<b>56</b>
How to use the script element .....	56
How to use the noscript element .....	58
<b>The JavaScript syntax .....</b>	<b>60</b>
How to code JavaScript statements .....	60
How to create identifiers .....	62
How to use comments .....	64
<b>How to work with data .....</b>	<b>66</b>
The primitive data types .....	66
How to declare and initialize variables and constants .....	68
How JavaScript handles variables and constants .....	70
<b>How to work with expressions .....</b>	<b>72</b>
How to code arithmetic expressions .....	72
How to use arithmetic expressions in assignment statements .....	74
How to concatenate strings .....	76
How to include special characters in strings .....	78
<b>How to use objects, methods, and properties .....</b>	<b>80</b>
Introduction to objects, methods, and properties .....	80
How to use the parseInt() and parseFloat() .....	82
methods of the window object .....	84
How to use the write() method of the document object .....	84
How to use the toFixed() method of a Number object .....	84
<b>Two illustrative applications .....</b>	<b>86</b>
The Miles Per Gallon application .....	86
The Test Scores application .....	88
<b>Perspective .....</b>	<b>90</b>

## How to include JavaScript in an HTML document

---

In chapter 1, you saw how to use the script element to include the JavaScript for an application that's in a separate file. But you can also code JavaScript directly within a script element. You'll learn more about both ways to use the script element now.

### How to use the script element

---

The table in figure 2-1 presents two attributes of the script element. As you saw in the last chapter, you use this element to include the JavaScript that's in a separate *external file*.

In the script element, the `src` attribute is used to refer to the external file. You can also code a `type` attribute with the value `"text/javascript"` to tell the browser what kind of content the file contains. But with HTML5, that attribute isn't needed because the assumption is that all files that are referred to in script elements contain JavaScript. However, you might see the `type` attribute in legacy code or in online examples.

In the first example in this figure, the `src` attribute refers to a file named `calculate_mpg.js`. The assumption here is that this file is in the same folder as the HTML file. Otherwise, you need to code a relative URL that provides the right path for the file. If, for example, the JavaScript file is in a folder named `javascript` and that folder is in the same folder as the HTML file, the `src` attribute would be coded this way:

```
<script src="javascript/calculate_mpg.js"></script>
```

This works the same as it does for any other file reference in an HTML document.

The second example in this figure shows another way to include JavaScript in an HTML document. Here, the JavaScript is coded directly within the script element. This can be referred to as *embedded JavaScript*.

Your applications will work the same whether the JavaScript is embedded in the script element or included from an external file. However, it's a better practice to use external files. That's because an external file separates the JavaScript from the HTML. Another benefit is that it makes it easier to reuse the code in other pages or applications.

As you can see, the script elements in both of the examples in this figure are coded at the end of the body element. You can also code script elements in the head element of an HTML document. However, it's recommended that you code your script elements in the body element after any other HTML elements. That way, your scripts have access to those HTML elements if they need them. In addition, all of the HTML will load before the JavaScript code is executed, which can prevent a blank page from being displayed for long-running scripts.

## Two attributes of the script element

Attribute	Description
<b>src</b>	Specifies the location (source) of an external JavaScript file.
<b>type</b>	With HTML5, this attribute can be omitted. If you code it, use “text/javascript” for JavaScript code.

### A script element that includes JavaScript that’s in an external file in an HTML page

```
<body>
...
<script src="calculate_mpg.js"></script>
</body>
```

### A script element that embeds JavaScript in an HTML page

```
<body>
...
<script>
    alert("The Calculate MPG application");

    let miles = prompt("Enter miles driven");
    miles = parseInt(miles);

    let gallons = prompt("Enter gallons of gas used");
    gallons = parseInt(gallons);

    let mpg = miles/gallons;
    mpg = parseFloat(mpg);

    alert("Miles per gallon = " + mpg);
</script>
</body>
```

## Description

- You can use a script element to identify an *external JavaScript file* that should be included with an HTML page. This is the more common way of including JavaScript.
- You can also use a script element that contains the JavaScript statements to be included in an HTML page. This can be referred to as *embedded JavaScript*.
- When a script element identifies an external JavaScript file, the JavaScript code in that file runs as if it were embedded in the script element.
- If you code more than one script element, the JavaScript code is included in the sequence in which the elements appear.
- You can also code script elements in the head element of an HTML document. However, it’s recommended that you include your script elements at the end of the body element.

---

Figure 2-1 How to use the script element

## How to use the noscript element

---

In the last figure, you learned how to include JavaScript in an HTML document. However, if a user has JavaScript disabled in their browser, then your JavaScript code won't run and your website won't function properly.

You can use the HTML noscript element to include the content in a web page that you want to display when a user has JavaScript disabled in their browser. In that case, the content within the noscript element is displayed. However, when JavaScript is enabled, the browser ignores the noscript element.

Programmers used to use noscript elements to create an alternate experience for users without JavaScript. However, over time websites became more reliant on JavaScript code to function, and that JavaScript code became more complex, making this strategy harder to implement. Today, noscript elements are used more often to notify a user that they need to enable JavaScript to use the website.

Figure 2-2 shows an HTML page with two noscript elements. The first one is within a header element. It displays a message that the website won't function properly if JavaScript isn't enabled in the browser.

The second noscript element is within the footer element, after a script element with embedded JavaScript that gets and displays the current date. Here, the noscript element provides an alternate experience to displaying the current date.

Next, this figure shows how this HTML page looks in a browser with JavaScript enabled. As you can see, none of the content within the noscript elements is sent to the browser. By contrast, the date produced by the embedded JavaScript code in the script element is sent to the browser.

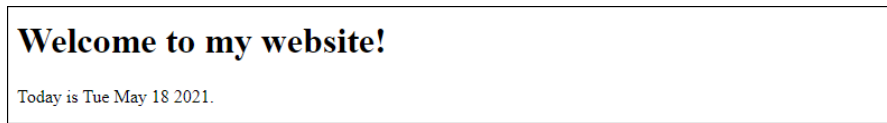
Next, this figure shows how this HTML page looks in a browser with JavaScript disabled. This time, the content within both noscript elements is sent to the browser, while the embedded JavaScript code in the script element doesn't execute.

If you include noscript elements in your HTML page, it can be useful to run your browser with JavaScript disabled so you can make sure these elements work the way you want them to. This figure shows how to disable and enable JavaScript in the Chrome browser.

## An HTML page with two noscript elements

```
<body>
  <header>
    <h1>Welcome to my website!</h1>
    <noscript>
      <h2>To get the most from this website, please enable JavaScript
        in your browser.</h2>
    </noscript>
  </header>
  <!-- main HTML for page goes here -->
  <footer>
    <script>
      const today = new Date();
      document.write(`Today is ${today.toDateString()}.`);
    </script>
    <noscript>
      Today is the first day of the rest of your life.
    </noscript>
  </footer>
</body>
```

## How the page looks in a browser with JavaScript enabled



## How the page looks in a browser with JavaScript disabled



## How to disable JavaScript in Chrome

1. Click the menu button in the upper right corner of the browser window.
2. Select Settings, then scroll to the “Privacy and security” section.
3. Expand Site Settings, then scroll to and expand the JavaScript item.
4. Toggle the Allowed switch to Blocked.

## How to enable JavaScript in Chrome

- Follow the procedure above, but toggle the Blocked switch to Allowed.

## Description

- You can use the noscript element to display content when JavaScript is disabled. This creates an alternate experience for non-JavaScript users.
- More often, you’ll use noscript elements to let users know they need to enable JavaScript.
- You can test your noscript elements by disabling JavaScript in your browser.

Figure 2-2 How to use the noscript element

# The JavaScript syntax

---

The *syntax* of JavaScript refers to the rules that you must follow as you code statements. If you don't adhere to these rules, your web browser won't be able to interpret and execute your statements.

## How to code JavaScript statements

---

Figure 2-3 summarizes the rules for coding *JavaScript statements*. The first rule is that JavaScript is case-sensitive. This means that uppercase and lowercase letters are treated as different letters. For example, *salestax* and *salesTax* are treated as different names.

The second rule is that JavaScript statements should end with a semicolon. That way, it will be easier to tell where one statement ends and the next one begins. To help remind you to include semicolons, many IDEs issue a warning if you omit them.

The third rule is that JavaScript ignores extra whitespace in statements. Since *whitespace* includes spaces, tabs, and new line characters, this lets you break long statements into multiple lines so they're easier to read.

Be careful, though, to follow the guidelines in this figure about where to split a statement. If you don't split a statement at a good spot, JavaScript will sometimes try to help you out by adding a semicolon for you, and that can lead to errors.

Finally, JavaScript has a *strict mode* that restricts the sorts of things you can do in your JavaScript code. For instance, later in this chapter you'll learn how to declare a variable using keywords. In non-strict mode, JavaScript also allows you to declare a variable without using a keyword. However, doing so leads to some unexpected behavior, which can lead to bugs that are hard to track down. In strict mode, though, if you try to declare a variable without a keyword, JavaScript throws an error. This alerts you to problems right away and helps you write safer code.

To enable strict mode, you code the "use strict" directive at the top of a code file or at the top of a function like the one shown in this figure. You should always use strict mode unless you've got a good reason not to. All of the applications for this book include the "use strict" directive at the top of every JavaScript code file.



## A block of JavaScript code

```
const joinList = () => {
  "use strict";

  const emailAddress1 = $("#email_address1").value;
  const emailAddress2 = $("#email_address2").value;
  const firstName = $("#first_name").value;

  if (emailAddress1 == "") {
    alert("Email Address is required.");
  } else if (emailAddress2 == "") {
    alert("Second Email Address is required.");
  } else if (emailAddress1 != emailAddress2) {
    alert("Second Email entry must equal first entry.");
  } else if (firstName == "") {
    alert("First Name is required.");
  } else {
    $("#email_form").submit();
  }
};
```

## The basic syntax rules

- JavaScript is case-sensitive.
- Each JavaScript statement ends with a semicolon.
- JavaScript ignores extra whitespace within statements.
- When JavaScript is in *strict mode*, it disallows certain JavaScript features and coding practices that are considered unsafe.

## How to split a statement over two or more lines

- Split a statement after:
  - an arithmetic or relational operator such as `+`, `-`, `*`, `/`, `=`, `==`, `>`, or `<`
  - an opening brace ( `{` ), bracket ( `[` ), or parenthesis
  - a closing brace ( `}` )
- Do not split a statement after:
  - an identifier, a value, or the *return* keyword
  - a closing bracket ( `]` ) or closing parenthesis

## Description

- A JavaScript *statement* has a syntax that's similar to the syntax of Java.
- *Whitespace* refers to the spaces, tab characters, and return characters in the code, and it is ignored by the compiler. As a result, you can use spaces, tab characters, and return characters to format your code so it's easier to read.
- In some cases, JavaScript will try to correct what it thinks is a missing semicolon by adding a semicolon at the end of a split line. To prevent this, follow the guidelines above for splitting a statement.
- *Strict mode* helps you write safer, cleaner code. To enable strict mode, you can include the "use strict" directive at the top of a code file or the top of a function.

---

Figure 2-3 How to code JavaScript statements

## How to create identifiers

---

Variables, constants, functions, objects, properties, methods, and events must all have names so you can refer to them in your JavaScript code. An *identifier* is the name given to one of these components.

Figure 2-4 shows the rules for creating identifiers in JavaScript. Besides the first four rules, you can't use any of the JavaScript *reserved words* (also known as *keywords*) as an identifier. These are words that are reserved for use within the JavaScript language. You should also avoid using any of the JavaScript global properties or methods as identifiers, which you'll learn more about as you progress through this book.

Besides the rules, you should give your identifiers meaningful names. That means that it should be easy to tell what an identifier refers to and easy to remember how to spell the name. To create names like that, you should avoid abbreviations. If, for example, you abbreviate the name for monthly investment as `mon_inv`, it will be hard to tell what it refers to and hard to remember how you spelled it. But if you spell it out as `monthly_investment`, both problems are solved.

Similarly, you should avoid abbreviations that are specific to one industry or field of study unless you are sure the abbreviation will be widely understood. For example, `mpg` is a common abbreviation for miles per gallon, but `cpm` could stand for a number of different things and should be spelled out.

To create an identifier that has more than one word in it, many JavaScript programmers use a convention called *camel casing*. With this convention, the first letter of each word is uppercase except for the first word. For example, `monthlyInvestment` and `taxRate` are identifiers that use camel casing.

An alternative is to use a convention called *snake casing*. With this convention, all the words in an identifier are lower case and separated by underscore characters. For example, `monthly_investment` and `tax_rate` are identifiers that use snake casing. If the standards in your shop specify one of these conventions, by all means use it. Otherwise, you can use whichever convention you prefer...but be consistent.

In this book, snake casing is used for the ids and class names in the HTML, and camel casing is used for all JavaScript identifiers. That way, it will be easier for you to tell where the names originated.

## Rules for creating identifiers

- Identifiers can only contain letters, numbers, the underscore, and the dollar sign.
- Identifiers can't start with a number.
- Identifiers are case-sensitive.
- Identifiers can be any length.
- Identifiers can't be the same as *reserved words*.
- Avoid using global properties and methods as identifiers. If you use one of them, you won't be able to use the global property or method with the same name.

## Valid identifiers in JavaScript

<code>subtotal</code>	<code>index_1</code>	<code>\$</code>
<code>taxRate</code>	<code>calculate_click</code>	<code>\$log</code>

## Camel casing versus snake casing

<code>taxRate</code>	<code>tax_rate</code>
<code>calculateClick</code>	<code>calculate_click</code>
<code>emailAddress</code>	<code>email_address</code>
<code>futureValue</code>	<code>future_value</code>

## Naming recommendations

- Use meaningful names for identifiers. That way, your identifiers aren't likely to be reserved words or global properties.
- Be consistent: Either use camel casing (`taxRate`) or snake casing (`tax_rate`) to identify the words within the variables in your scripts.
- If you use snake casing, use lowercase for all letters.

## Reserved words in JavaScript

<code>abstract</code>	<code>else</code>	<code>instanceof</code>	<code>switch</code>
<code>arguments</code>	<code>enum</code>	<code>int</code>	<code>synchronized</code>
<code>boolean</code>	<code>eval</code>	<code>interface</code>	<code>this</code>
<code>break</code>	<code>export</code>	<code>let</code>	<code>throw</code>
<code>byte</code>	<code>extends</code>	<code>long</code>	<code>throws</code>
<code>case</code>	<code>false</code>	<code>native</code>	<code>transient</code>
<code>catch</code>	<code>final</code>	<code>new</code>	<code>true</code>
<code>char</code>	<code>finally</code>	<code>null</code>	<code>try</code>
<code>class</code>	<code>float</code>	<code>package</code>	<code>typeof</code>
<code>const</code>	<code>for</code>	<code>private</code>	<code>var</code>
<code>continue</code>	<code>function</code>	<code>protected</code>	<code>void</code>
<code>debugger</code>	<code>goto</code>	<code>public</code>	<code>volatile</code>
<code>default</code>	<code>if</code>	<code>return</code>	<code>while</code>
<code>delete</code>	<code>implements</code>	<code>short</code>	<code>with</code>
<code>do</code>	<code>import</code>	<code>static</code>	<code>yield</code>
<code>double</code>	<code>in</code>	<code>super</code>	

## Description

- *Identifiers* are the names given to variables, functions, objects, properties, and methods.
- In *camel casing*, all of the words within an identifier except the first word start with capital letters. In *snake casing*, all words are lower case and separated by underscores.

Figure 2-4 How to create identifiers

## How to use comments

---

*Comments* let you add descriptive notes to your code that are ignored by the JavaScript engine. Later on, these comments can help you or someone else understand the code whenever it needs to be modified.

The example in figure 2-5 shows how comments can be used to describe or explain portions of code. At the start, a *block comment* describes what the application does. This kind of comment starts with `/*` and ends with `*/`. Everything that's coded between the start and the end is ignored by the JavaScript engine when the application is run.

The other kind of comment is a *single-line comment* that starts with `//`. In the example, the first single-line comment describes what the JavaScript that comes before it on the same line does. By contrast, the second single-line comment takes up a line by itself. It describes what the function that comes after it does.

In addition to describing JavaScript code, comments can be useful when testing an application. If, for example, you want to disable a portion of the JavaScript code, you can enclose it in a block comment. Then, it will be ignored when the application is run. This can be referred to as *commenting out* a portion of code. Later, after you test the rest of the code, you can enable the commented out code by removing the markers for the start and end of the block comment. This can be referred to *uncommenting*.

Comments are also useful when you want to experiment with changes in code. For instance, you can make a copy of a portion of code, comment out the original code, and then paste the copy just above the original code that is now commented out. Then, you can make changes in the copy. But if the changes don't work, you can restore your old code by deleting the new code and uncommenting the old code.

When should you use comments to describe or explain code? Certainly, when the code is so complicated that you may not remember how it works if you have to maintain it later on. This kind of comment is especially useful if someone else is going to have to maintain the code.

On the other hand, you shouldn't use comments to explain code that any professional programmer should understand. That means that you have to strike some sort of balance between too much and too little.

One of the worst problems with comments is changing the way the code works without changing the related comments. Then, the comments mislead the person who is trying to maintain the code, which makes the job even more difficult.

## A portion of JavaScript code that includes comments

```

/* this application validates a user's entries for joining
   our email list
*/
"use strict";

const $ = selector => {                                // the $ function
    return document.querySelector(selector);
};
// this function gets and validates the user entries
const joinList = () => {
    const emailAddress1 = $("#email_address1").value;
    const emailAddress2 = $("#email_address2").value;
    const firstName = $("#first_name").value;
    let isValid = true;                                // set default value

    // validate the first entry
    if (emailAddress1 == "") {
        $("#email_address1_error").textContent = "This field is required.";
        isValid = false;                                // set valid flag to off
    } else {
        $("#email_address1_error").textContent = "";
    }

    // validate the second entry
    ...
    ...
};

```

## The basic syntax rules for JavaScript comments

- Block comments begin with `/*` and end with `*/`.
- Single-line comments begin with two forward slashes and continue to the end of the line.

## Guidelines for using comments

- Use comments to describe portions of code that are hard to understand.
- Use comments to disable portions of code that you don't want to test.
- Don't use comments unnecessarily.

## Description

- JavaScript provides two forms of *comments*: *block comments* and *single-line comments*.
- Comments are ignored when the JavaScript is executed.
- During testing, comments can be used to *comment out* (disable) portions of code that you don't want tested. Then, you can remove the comments when you're ready to test those portions.
- You can also use comments to save a portion of code in its original state while you make changes to a copy of that code.

## How to work with data

---

When you develop JavaScript applications, you frequently work with data, especially the data that users enter into the controls of a form. In the topics that follow, you'll learn how to work with three of the seven types of JavaScript data.

### The primitive data types

---

JavaScript provides for seven *primitive data types*, which are summarized in the table in figure 2-6. The *number data type* is used to represent numerical data, which can be either integer or decimal values. *Integers* are whole numbers, and *decimal values* are numbers that can have one or more decimal digits. The value of either data type can be coded with a preceding plus or minus sign. If the sign is omitted, the value is treated as a positive value. A decimal value can also include a decimal point and one or more digits to the right of the decimal point.

As the last example of the number types in this figure shows, you can also include an exponent when you code a decimal value. If you aren't familiar with this notation, you probably won't need to use it because you won't be working with very large or very small numbers. On the other hand, if you're familiar with scientific notation, you already know that this exponent indicates how many places the decimal point should be moved to the left or right. Numbers that use this notation are called *floating-point numbers*.

The *string data type* is used to store character data. To represent string data, you code the *string* within single or double quotation marks (quotes). Note, however, that you must close the string with the same type of quotation mark that you used to start it. If you code two quotation marks in a row without even a space between them, the result is called an *empty string*, which can be used to represent a string with no characters in it.

The *Boolean data type* is used to store true and false values. To represent Boolean data, you code either the word *true* or *false* with no quotation marks. This data type can be used to represent one of two states.

These first three data types are the ones you'll use the most, which is why this figure and the ones that follow focus on them. However, you should be familiar with the rest of the primitive data types as well.

The *symbol data type* was introduced in ES6, and it can be used to create unique identifiers as properties for objects. This helps ensure that an object doesn't have an identifier with the same name as one in another object. You'll learn how to work with symbols in chapter 12.

The *null data type* and the *undefined data type* are similar in that they both represent the absence of a value. However, they're used differently. The undefined value is used by JavaScript to indicate that an object hasn't been assigned a value by the programmer. The null value, by contrast, is used by programmers to indicate that an object has been intentionally set to no value. You'll see examples of this later in this chapter and book.

The *bigint data type* was introduced in ES2020 (released in June 2020). It represents very large and very small numbers.

## JavaScript's primitive data types

Data type	Represents...
Number	an integer or a decimal value that can start with a positive or negative sign.
String	character (string) data.
Boolean	a value that has two possible states: true or false.
Symbol	a unique value that can't be changed.
Null	no value.
Undefined	a value that hasn't been assigned.
BigInt	an extremely large or extremely small number.

### Examples of number values

```
15           // an integer
-21          // a negative integer
21.5         // a decimal value
-124.82      // a negative decimal value
-3.7e-9      // floating-point notation for -0.0000000037
```

### Examples of string values

```
"JavaScript" // a string with double quotes
'String Data' // a string with single quotes
""           // an empty string
```

### The two Boolean values

```
true        // equivalent to true, yes, or on
false       // equivalent to false, no, or off
```

### The number data type

- A number value can be an *integer*, which is a whole number, or a *decimal value*, which can have one or more decimal positions to the right of the decimal point.
- In JavaScript, decimal values are stored as *floating-point numbers*. In that format, a number consists of a positive or negative sign, one or more significant digits, an optional decimal point, optional decimal digits, and an optional exponent.
- If a result is stored in a number data type that is larger or smaller than the data type can store, it will be stored as the value Infinity or -Infinity.

### The string data type

- A string value is surrounded by double quotes or single quotes. The string must start and end with the same type of quotation mark.
- An *empty string* is a string that contains no characters. It is entered by typing two quotation marks with nothing between them.

### The Boolean data type

- A *Boolean value* can be true or false. Boolean values are often used in conditional statements, as you'll see in chapter 3.

Figure 2-6 The primitive data types

## How to declare and initialize variables and constants

---

A *variable* stores a value that can change as the program executes. To *declare* a variable in JavaScript, you code the *let* keyword followed by the identifier (or name) that you want to use for the variable. JavaScript also provides an older keyword, *var*, that you can use to declare and initialize variables. As you'll see in a moment, though, it's recommended that you use *let* instead.

When you declare a variable, you should also assign an initial value to it. To *initialize* a variable, you code an equals sign (the *assignment operator*) and a value after the variable name. The value that you assign to a variable determines its data type.

The first two statements in the first group of examples in figure 2-7 show how to declare a variable and initialize it with a numeric value. To do that, you assign a *numeric literal* to the variable, which consists of an integer or decimal value. Here, the first statement declares a variable named *count* and assigns an integer value of 1 to it. The second statement declares a variable named *subtotal* and assigns a decimal value of 74.95 to it.

The third and fourth statements show how to declare a variable and initialize it with a string value. To do that, you assign a *string literal* to the variable. A string literal consists of a value that's enclosed in single or double quotes. The third statement, for example, declares a variable named *name* and assigns a value of "Joseph" to it. The fourth statement is similar, except no value is included between the quotes. As you learned in the last topic, this represents an empty string that doesn't contain any characters.

The fifth statement shows how to declare a variable and initialize it with a Boolean value. To do that, you assign the keyword *true* or *false* to the variable. Here, the value *false* is assigned to a variable named *isValid*.

The sixth statement shows that, in addition to assigning a *literal value* to a variable, you can assign another variable to it. In this case, the value of the *subtotal* variable in the second statement is assigned to a variable named *total*. You can also assign an expression to a variable, which you'll see a bit later.

The last statement shows that you can declare and initialize two or more variables in a single statement. To do that, you code *let* followed by the variable name, an equals sign, and a value for each variable, separated by commas.

It's also possible to declare a variable without initializing it. In this case, JavaScript initializes the variable with *undefined*. However, this is considered a poor practice because uninitialized variables are a frequent source of bugs. As a result, you should always initialize your variables, even if you just assign a null value.

A *constant* (sometimes called, somewhat inaccurately, a *constant variable*) stores a value that cannot change once it's been initialized. The syntax for declaring a constant is the same as for a variable, except you use the *const* keyword. In addition, a constant must be initialized when it's declared.

Once a variable is initialized, you can use the assignment operator to reassign its value as shown here. If the new value is a different data type, the



## How to declare and initialize a variable

### Syntax

```
let variableName = value;
```

### Examples

```
let count = 1;           // integer value of 1
let subtotal = 74.95;    // decimal value of 74.95
let name = "Joseph";     // string value of "Joseph"
let email = "";          // empty string
let isValid = false;     // Boolean value of false
let total = subtotal;    // assigns value of subtotal variable
let x = 0, y = 0;        // declares and initializes 2 variables
```

## How to declare and initialize a constant

### Syntax

```
const constantName = value;
```

### Examples

```
// same as above but with const keyword
```

## How to reassign the value of a variable

```
let count = 1;
count = count + 1;          // value of count is now 2
```

## What happens when you try to reassign the value of a constant

```
const count = 1;
count = count + 1;          // TypeError: Assignment to constant variable.
```

## Description

- A *variable* stores a value that can change as the program executes. A *constant* (or *constant variable*) stores a value that may not be reassigned once it's been created.
- To *declare* a variable, code the keyword *let* and a variable name. To *initialize* a variable, use the *assignment operator* (=) to assign a value to it.
- To declare and initialize a constant, do the same but use the *const* keyword.
- You can also declare a variable with the *var* keyword, but that's not recommended.
- The data type of a variable or constant is determined by the value that's assigned to it.
- Although you can declare a variable without initializing it, that's not recommended. A constant must be initialized when it's declared.
- You can use commas to declare more than one variable or constant in a single statement.
- The value that's assigned to a variable or constant can be a *literal value*, or *literal*, another variable or constant, or an expression like the ones you'll learn in a moment.
- To code a *string literal*, you enclose it in single or double quotes. To code a *numeric literal*, you code an integer or decimal value that isn't enclosed in quotes.
- To assign a Boolean value to a variable, you use the *true* and *false* keywords.

---

Figure 2-7 How to declare and initialize variables and constants

data type of the variable changes. By contrast, you can't reassign the value of a constant. If you try, JavaScript throws an error.

Many programmers consider it a best practice to use constants instead of variables unless they're certain they need to change a value. That's because changing a variable's value during program execution can be a source of bugs.

If you use constants and you try to change a constant's value, JavaScript will alert you by throwing an error. Then, you can evaluate your code and decide if you really need to change that value. If so, you can change the constant to a variable by changing the *const* keyword to *let*.

## How JavaScript handles variables and constants

---

Figure 2-8 presents some basic information about how JavaScript treats the variables and constants that you declare in your code. Understanding this can save you trouble later on.

When JavaScript code loads, the JavaScript engine scans it and stores any variables or constants declared with the *let*, *const*, or *var* keyword in memory first. Then, it rescans and executes the code in the order that the code appears. This process is called *hoisting*, because it seems as if the variables and constants have been “hoisted” to the top of the file.

For variables declared with *var*, JavaScript also initializes them to undefined when they're hoisted. As a result, if your code refers to one of these variables before it's declared, JavaScript won't tell you that it doesn't exist. This is shown in the first group of examples in this figure.

In the first example in this group, the first statement displays the value of a variable named `val1` in the browser. Then, the second statement declares and initializes that variable. Because `val1` is declared with *var*, JavaScript hoists it and initializes it to undefined. As a result, this code displays “undefined” in the browser. In the second example, the variable is declared and initialized before it's used. As a result, this code displays “value”.

By contrast, variables declared with *let* and constants declared with *const* aren't initialized when they're hoisted. Because of that, they can't be used until you initialize them, even though they exist in memory. This period of time, from when JavaScript scans and stores a variable or constant to when you initialize it, is called the *temporal dead zone*, or *TDZ*. If you try to access a variable or constant in the TDZ, JavaScript throws an error.

This is shown in the second group of examples in this figure. In the first example in this group, the first statement displays the value of a variable named `val1` in the browser. Then, the second statement declares and initializes the variable. As a result, the `val1` variable is in the TDZ until the second statement runs. Because of that, trying to access it in the first statement causes an error.

To avoid these kinds of problems, you should always declare and initialize a variable or constant before you use it. In addition, you should use the *let* keyword to declare variables. That way, if you accidentally try to use a variable before you declare it, JavaScript will let you know by throwing an error.

## How JavaScript processes variable and constant declarations

- JavaScript places all variable and constant declarations in memory first. This is called *hoisting*.
- When a variable that's declared with the *var* keyword is hoisted, JavaScript initializes it with a value of *undefined*. Because of this, you can access the variable before you declare it, but its value will be *undefined*.
- When a variable that's declared with the *let* keyword is hoisted, JavaScript doesn't initialize it. That's also true of a constant. Because of this, you can't access the variable or constant before you declare it.

### A variable declared with the *var* keyword

#### Accessed before it's declared

```
alert(val1);           // displays "undefined"
var val1 = "value";
```

#### Accessed after it's declared

```
var val1 = "value";
alert(val1);           // displays "value"
```

### A variable declared with the *let* keyword

#### Accessed before it's declared

```
alert(val2);           // ReferenceError - nothing displays
let val2 = "value";
```

#### Accessed after it's declared

```
let val2 = "value";
alert(val2);           // displays "value"
```

## Description

- When the JavaScript engine interprets a script, it scans the code and puts all variable and constant declarations in memory first. This is called *hoisting*, because it's as if the variables and constants are 'hoisted' to the top of the script.
- Variables declared with *var* are initialized when they're hoisted, while variables declared with *let* and *const* are not.
- A variable declared with *var* is always accessible. That is, JavaScript won't throw an error if you try to access it before you declare it. However, this usually isn't what you want, and it can lead to hard-to-find bugs.
- A variable declared with *let* or *const* is inaccessible between the time it's hoisted and when you declare it. This period is called the *temporal dead zone*, or *TDZ*.
- If you try to access a variable in the TDZ, JavaScript will throw an error. This is more helpful behavior, as it lets you know that you're doing something you shouldn't.
- To avoid problems, you should declare and initialize your variables before you use them.
- In general, you should avoid using the *var* keyword in new code. However, you'll see it often in legacy code and in online examples.

---

Figure 2-8 How JavaScript handles variable and constant declarations

## How to work with expressions

---

In the preceding figures, you learned how to declare variables and constants and assign literal values and the values in other variables or constants to them. Now, you'll learn how to assign expressions to a variable or constant. An *expression* uses *operators* to perform operations on values.

### How to code arithmetic expressions

---

An *arithmetic expression* can be as simple as a single value or it can be a series of operations that result in a single value. In figure 2-9, you can see the operators for coding arithmetic expressions. If you've programmed in another language, these are probably similar to what you've been using. In particular, the first four *arithmetic operators* are common to most programming languages.

Most modern languages also have a *modulus operator* that calculates the remainder when the left value is divided by the right value. In the example for this operator,  $13 \% 4$  means the remainder of  $13 / 4$ . Then, since  $13 / 4$  is 3 with a remainder of 1, 1 is the result of the expression.

In contrast to the first five operators in this figure, the increment and decrement operators add or subtract one from a variable. To complicate matters, though, these operators can be coded before or after a variable name, and that can affect the result. To avoid confusion, then, it's best to only code these operators after the variable names and only in simple expressions like the one that you'll see in the next figure.

When an expression includes two or more operators, the *order of precedence* determines which operators are applied first. This order is summarized in the second table in this figure. For instance, all multiplication and division operations are done from left to right before any addition and subtraction operations are done.

To override this order, though, you can use parentheses. Then, the expressions in the innermost sets of parentheses are done first, followed by the expressions in the next sets of parentheses, and so on. This is typical of all programming languages, as well as basic algebra, and the examples in this figure show how this works.

## JavaScript's arithmetic operators

Operator	Name	Description
+	Addition	Adds two operands.
-	Subtraction	Subtracts the right operand from the left operand.
*	Multiplication	Multiplies two operands.
/	Division	Divides the right operand into the left operand. The result is always a floating-point number.
%	Modulus	Divides the right operand into the left operand and returns the remainder.
++	Increment	Adds 1 to the operand.
--	Decrement	Subtracts 1 from the operand.

## The order of precedence for arithmetic operations

Order	Operators	Direction	Description
1	++	Left to right	Increment operator
2	--	Left to right	Decrement operator
3	* / %	Left to right	Multiplication, division, modulus
4	+ -	Left to right	Addition, subtraction

## Examples of simple arithmetic expressions

Example	Result
5 + 7	12
5 - 12	-7
6 * 7	42
13 / 4	3.25
13 % 4	1
counter++	counter = counter + 1
counter--	counter = counter - 1
3 + 4 * 5	23 (the multiplication is done first)
(3 + 4) * 5	35 (the addition is done first)
13 % 4 + 9	10 (the modulus is done first)
13 % (4 + 9)	0 (the addition is done first)

## Description

- An *arithmetic expression* consists of one or more operands that are operated upon by *arithmetic operators*.
- An arithmetic expression is evaluated based on the *order of precedence* of the operators. To override the order of precedence, you can use parentheses.
- Because the use of increment and decrement operators can be confusing, it's best to only use these operators in expressions that consist of a variable name followed by the operator as shown above.

Figure 2-9 How to code arithmetic expressions

## How to use arithmetic expressions in assignment statements

---

Now that you know how to code arithmetic expressions, figure 2-10 shows how to use these expressions with variables and constants as you code *assignment statements*. Here, the first two examples show how you can use the multiplication and addition operators in JavaScript statements.

This is followed by a table that presents three of the *compound assignment operators*. These operators provide a shorthand way to code common assignment statements. For instance, the `+=` operator modifies the value of the variable on the left of the operator by adding the value of the expression on the right to the value of the variable on the left. When you use this operator, the variable on the left must already exist and have a value assigned to it.

The other two operators in this table work similarly, but the `-=` operator subtracts the result of the expression on the right from the variable on the left. And the `*=` operator multiplies the variable on the left by the result of the expression on the right. The first group of examples after this table illustrates how these operators work.

The second example after the table shows three ways to increment a variable by adding 1 to it. As you will see throughout this book, this is a common JavaScript requirement. Here, the first statement assigns a value of 1 to a variable named `counter`.

Then, the second statement uses an arithmetic expression to add 1 to the value of the `counter`. This shows that a variable name can be used on both sides of the `=` operator. The third statement adds one to the `counter` by using the `+=` operator. When you use this operator, you don't need to code the variable name on the right side of the `=` operator, which makes the code more concise.

The last statement in this example uses the `++` operator shown in the previous figure to add one to the `counter`. This illustrates the best way to use increment and decrement operators. Here, the numeric expression consists only of a variable name followed by the increment operator.

The last example illustrates a potential problem that you should be aware of. Because decimal values are stored internally as floating-point numbers, the results of arithmetic operations aren't always precise. In this example, the `salesTax` result, which should be 7.495, is 7.495000000000001. Although this result is extremely close to 7.495, it isn't equal to 7.495, which could lead to a programming problem if you expect a comparison of the two values to be equal. The solution is to round the result. You'll learn one way to do that later in this chapter.

### Code that calculates sales tax

```
const subtotal = 200;
const taxPercent = .05;
const taxAmount = subtotal * taxPercent;           // 10
const total = subtotal + taxAmount;                // 210
```

### Code that calculates the perimeter of a rectangle

```
const width = 4.25;
const length = 8.5;
const perimeter = (2 * width) + (2 * length)        // (8.5 + 17) = 25.5
```

### The most useful compound assignment operators

Operator	Description
+=	Adds the result of the expression to the variable.
-=	Subtracts the result of the expression from the variable.
*=	Multiplies the variable value by the result of the expression.

### Statements that use the compound assignment operators

```
let subtotal = 74.95;
subtotal += 20.00;           // subtotal = 94.95
let counter = 10;
counter -= 1;                 // counter = 9
let price = 100;
price *= .8;                  // price = 80
```

### Three ways to increment a variable named counter by 1

```
let counter = 1;              // counter = 1
counter = counter + 1;         // counter now = 2
counter += 1;                  // counter now = 3
counter++;                     // counter now = 4
```

### A floating-point result that isn't precise

```
const subtotal = 74.95;       // subtotal = 74.95
const salesTax = subtotal * .1; // salesTax = 7.4950000000000001
```

### Description

- Besides the assignment operator (=), JavaScript provides for *compound assignment operators*. These operators are a shorthand way to code common assignment operations.
- JavaScript also offers /= and %= compound operators, but you won't use them often.
- When you do some types of arithmetic operations with decimal values, the results aren't always precise, although they are extremely close. That's because decimal values are stored internally as floating-point numbers. The primary problem with this is that an equality comparison may not return true.

Figure 2-10 How to use arithmetic expressions in assignment statements

## How to concatenate strings

---

Figure 2-11 shows how to *concatenate*, or *join*, two or more strings. This means that one string is added to the end of another string.

To concatenate strings, you can use the `+` sign as a *concatenation operator*. This is illustrated by the first example in this figure. Here, the first code statement assigns string literals to the constants named `firstName` and `lastName`. Then, the next statement creates a *string expression* by concatenating `lastName`, a string literal that consists of a comma and a space, and `firstName`. The result of this concatenation is

**Hopper, Grace**

which is stored in a new constant named `name`.

In the second example, you can see how the `+=` operator can be used to get the same results. When the expressions that you're working with are strings, this operator does a simple concatenation.

Another way to concatenate strings is to use a *template literal*. A template literal is like a string literal, except it's enclosed in tick marks (```) rather than single or double quotes. Then, you embed string literals, variables, or constants within the template literal. To do that, you enclose the embedded string within braces (`{ }`) that are preceded by a dollar sign (`$`).

In the third example, you can see how this works. Here, the `lastName` and `firstName` constants are embedded within a template literal that contains a comma and a space.

Sometimes, the strings you're concatenating are so long that your code needs to be on more than one line. The last example in this figure shows how to do that with the `+` operator and with a template literal. The important thing to note here is that when you use the `+` operator, you need to code a `+` operator at the end of each line that will continue to a new line. When you use a template literal, though, you don't need to do anything special at the end of lines.

Note that the constants to be concatenated in this example aren't all strings. Instead, the `greeting` constant is a string, the `price` constant is a number, and the `isValid` constant is a Boolean. When you concatenate a string and any other data type, the other type is converted to a string and then the strings are concatenated.

The concatenation technique you use is mostly a matter of personal preference. In general, though, template literals are more readable, while the `+` and `+=` operators perform slightly faster.

When working with multiple lines, however, you should know that the spaces and line breaks in your code are preserved in a template literal. This won't matter if you're outputting the string as HTML, but it can cause formatting issues in the dialog boxes produced by the `alert()` and `prompt()` methods of the window object. You'll learn about these methods in figure 2-13.



## The concatenation operators for strings

Operator	Description
+	Concatenates two values.
+=	Concatenates the result of the expression to the end of the variable.

### Constants used for the first three examples

```
const firstName = "Grace", lastName = "Hopper";
```

### How to concatenate string variables with the + operator

```
const name = lastName + ", " + firstName;           // name is "Hopper, Grace"
```

### How to concatenate string variables with the += operator

```
let name = lastName;           // name is "Hopper"
name += ", ";                  // name is "Hopper, "
name += firstName;             // name is "Hopper, Grace"
```

### How to concatenate string variables with a template literal

```
const name = `${lastName}, ${firstName}`;           // name is "Hopper, Grace"
```

### How to concatenate on multiple lines

```
const greeting = "Hello";
const price = 15.99;           // number data type
const isValid = true;          // boolean data type
```

#### With the + operator

```
const message = greeting + "! Is the price really " +
    "just " + price + "? Answer: " + isValid + ".";
// message is "Hello! Is the price really just 15.99? Answer: true."
```

#### With a template literal

```
const message = `${greeting}! Is the price really
    just ${price}? Answer: ${isValid}.`;
// message is "Hello! Is the price really just 15.99? Answer: true."
```

## Description

- To *concatenate*, or *join*, two or more strings, you can use the + or += operator.
- You can also use a *template literal*, which allows you to embed strings directly within a string that functions as a template.
- A template literal is enclosed in tick marks (``) rather than quote marks. Then, within the template literal, the embedded strings are enclosed in braces ({}) that are preceded by a dollar sign (\$).
- You can concatenate strings with values of other data types, like numbers or Booleans. When you do, JavaScript converts the non-string value to a string and then concatenates it.
- You can concatenate long strings on multiple lines with the + operator or with a template literal.
- A template literal is easier to read, but the + and += operators perform faster.

Figure 2-11 How to concatenate strings

## How to include special characters in strings

---

The first table in figure 2-12 summarizes four of the many *escape sequences* that you can use when you work with strings. These sequences let you put characters in a string that you can't put in just by pressing the appropriate key on the keyboard. For instance, the `\n` escape sequence is equivalent to pressing the Enter key in the middle of a string. And the `\'` sequence is equivalent to pressing the key for a single quotation mark.

Escape sequences are needed so the JavaScript engine can interpret code correctly. For instance, since single and double quotations marks are used to identify strings in JavaScript statements, coding them within the strings would cause syntax errors. But when the quotation marks are preceded by escape characters, the JavaScript engine can interpret them correctly.

The code examples below the table show some strings with escape sequences that are passed to the `alert()` method. You'll learn more about this method in the next figure. For now, just know that it causes the browser to display a dialog box like the two shown below the code examples. Note how the escape sequences in the strings are displayed in these dialog boxes.

The second table in this figure summarizes four of the many codes that you can use to include *Unicode characters* in strings. These codes let you include letters from other languages in a string, like letters that include accents, umlauts, and tildes. They also let you include middle Eastern script and Korean, Chinese, and Japanese ideographs, as well as symbols like the copyright symbol and the registered trademark symbol.

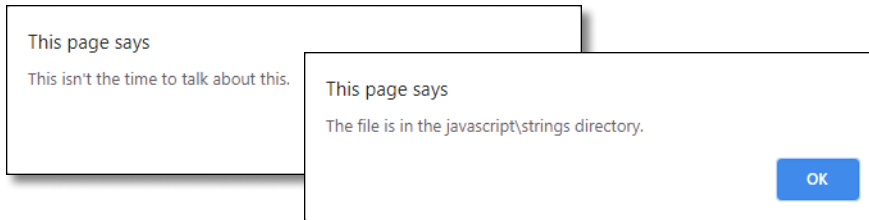
The code example below the table shows a string with Unicode characters. If you study the dialog box that displays this string, you can see that it contains a heart symbol, a trademark symbol, a smiley face symbol, and a copyright symbol. You can use the URL at the bottom of this figure to look up the codes for more Unicode characters.

## Some of the escape sequences that can be used in strings

Operator	Description
<code>\n</code>	Starts a new line in a string. Doesn't affect HTML but does work with the text in alerts and prompts.
<code>\'</code>	Puts a single quotation mark in a string.
<code>\"</code>	Puts a double quotation mark in a string.
<code>\\</code>	Puts a backslash in a string.

## How escape sequences can be used in a string

```
alert("This isn't the time to talk about this.");
alert("The file is in the javascript\\strings directory.");
```

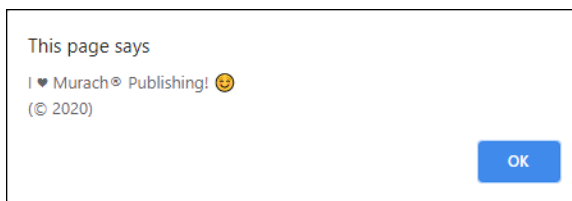


## Some of the codes for Unicode characters

Code	Character	Description
<code>\u00A9</code>	©	Copyright
<code>\u00AE</code>	®	Registered trademark
<code>\u263A</code>	☺	Smiley face
<code>\u2665</code>	♥	Heart

## How Unicode characters can be used in a string

```
alert("I \u2665 Murach\u00AE Publishing! \u263A \n(\u00A9 2020)");
```



## A website that lists all the Unicode characters

[https://en.wikipedia.org/wiki/List\\_of\\_Unicode\\_characters](https://en.wikipedia.org/wiki/List_of_Unicode_characters)

## Description

- You can use *escape sequences* to insert special characters within a string, like a return character that starts a new line or a quotation mark.
- You can use *Unicode characters* to include letters, punctuation, scripts, and ideographs from most languages, as well as some symbols.

Figure 2-12 How to include special characters in strings

# How to use objects, methods, and properties

---

In simple terms, an *object* is a collection of methods and properties. A *method* performs a function or does an action. A *property* is a data item that relates to the object. When you develop JavaScript applications, you will often work with objects, methods, and properties. You'll learn how to use the methods and properties of many objects as you progress through this book.

## Introduction to objects, methods, and properties

---

To get you started with objects, methods, and properties, figure 2-13 shows how to use some of the methods and properties of the *window object*, which is a common JavaScript object. To *call* (execute) a method of an object, you use the syntax in the summary after the tables. That is, you code the object name, a *dot operator* (period), the method name, and any *parameters* that the method requires within parentheses.

In the syntax summaries in this book, some words are italicized and some aren't. The words that aren't italicized are keywords that always stay the same, like *alert*. You can see this in the first table, where the syntax for the `alert()` method shows that you code the word `alert` just as it is in the summary. By contrast, the italicized words are the ones that you need to supply, like the string parameter you supply to the `alert()` method.

In the first example after the syntax summary, you can see how the `alert()` method of the window object is called:

```
window.alert("This is a test of the alert method");
```

In this case, the one parameter that's passed to it is "This is a test of the alert method". So that message is displayed in the alert dialog box.

In the second example, you can see how the `prompt()` method of the window object is called. This time, though, the object name is omitted. For the window object (but only the window object), that's okay because the window object is the *global object* for JavaScript applications.

As you can see, the `prompt()` method accepts two parameters. The first one is a message, and the second one is an optional default value for a user entry. When the `prompt()` method is executed, it displays a dialog box like the one in this figure. Here, you can see the message and the default value that were passed to the method as parameters. At this point, the user can change the default value or leave it as is, and then click on the OK button to store the entry in the constant named `userEntry`. Or, the user can click on the Cancel button to cancel the entry, which returns a null value.

To access a property of an object, you use a similar syntax. However, you code the property name after the dot operator as illustrated by the second syntax summary. Unlike methods, properties don't require parameters in parentheses. This is illustrated by the statement that follows the syntax. This statement uses the `alert()` method of the window object to display the `location` property of the window object.

## Common methods of the window object

Method	Description
<code>alert(string)</code>	Displays a dialog box that contains the string that's passed to it by the parameter along with an OK button.
<code>prompt(string, default)</code>	Displays a dialog box that contains the string in the first parameter, the default value in the second parameter, an OK button, and a Cancel button. If the user enters a value and clicks OK, that value is returned as a string. If the user clicks Cancel, null is returned.

## One property of the window object

Property	Description
<code>location</code>	The URL of the current web page.

## The syntax for calling a method of an object

`objectName.methodName(parameters)`

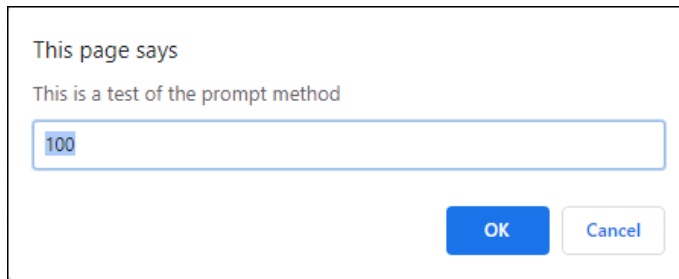
### A statement that calls the alert() method of the window object

```
window.alert("This is a test of the alert method");
```

### A statement that calls the prompt() method with the object name omitted

```
const userEntry = prompt("This is a test of the prompt method", 100);
```

### The prompt dialog box that's displayed



## The syntax for accessing a property of an object

`objectName.propertyName`

### A statement that displays the location property of the window object

```
alert(window.location); // Displays the URL of the current page
```

## Description

- An *object* has *methods* that perform functions that are related to the object as well as *properties* that represent the data or attributes that are associated with the object.
- When you *call* a method, you may need to pass one or more *parameters* to it by coding them within the parentheses after the method name, separated by commas.
- The *window object* is the *global object* for JavaScript, and JavaScript lets you omit the object name and *dot operator* (period) when referring to the window object.

Figure 2-13 Introduction to objects, methods, and properties

## How to use the `parseInt()` and `parseFloat()` methods of the window object

---

The `parseInt()` and `parseFloat()` methods are used to convert strings to numbers. The `parseInt()` method converts a string to an integer, and the `parseFloat()` method converts a string to a decimal value. If the string can't be converted to a number, the value *NaN* is returned. *NaN* means “Not a Number”. You'll learn one way to check whether a value is a number in the next chapter.

These methods are needed because the values that are returned by the `prompt()` method and the values that the user enters into text boxes are treated as strings. This is illustrated by the first group of examples in figure 2-14. For this group, assume that the default value in the `prompt()` method isn't changed by the user. As a result, the first statement in this group stores 12345.6789 as a string in a variable named `entryA`. Then, the third statement in this group converts the string to an integer value of 12345.

Note that the object name isn't coded before the method name in these examples. Again, that's okay because `window` is the global object for JavaScript. Note too that the `parseInt()` method doesn't round the value. It just removes, or truncates, any decimal portion of the string value.

The last four statements in the first group of examples show what happens when the `parseInt()` or `parseFloat()` method is used to convert a value that isn't a number. In that case, the value *NaN* is returned.

Note, however, that these methods can convert values that consist of one or more numeric characters followed by one or more nonnumeric characters. In that case, these methods simply drop the nonnumeric characters. For example, if a string contains the value 72.5%, the `parseFloat()` method will convert it to a decimal value of 72.5.

The second group of examples in this figure shows how to get the same results by coding the `prompt()` method as the parameter of the `parseInt()` and `parseFloat()` methods. In this way, the value that's returned by the `prompt()` method is immediately passed to the `parseInt()` or `parseFloat()` method.

Note in the first group of examples that `entryA`, `entryB`, and `entryC` are variables because their values change. By contrast, in the second group of examples, `entryA`, `entryB`, and `entryC` are constants because their values don't change. Because working with constants is safer, the second group of examples is preferable. However, if you find that coding a method as a parameter for another method makes your code hard to read, it's OK to use variables and be less concise.

## Two methods of the window object

Method	Description
<code>parseInt(string)</code>	Converts the string it receives to an integer data type and returns that value. If it can't convert the string to an integer, it returns NaN.
<code>parseFloat(string)</code>	Converts the string it receives to a decimal data type and returns that value. If it can't convert the string to a decimal value, it returns NaN.

### Examples that use the `parseInt()` and `parseFloat()` methods

```
let entryA = prompt("Enter any value", 12345.6789);
alert(entryA);                                     // displays 12345.6789
entryA = parseInt(entryA);
alert(entryA);                                     // displays 12345

let entryB = prompt("Enter any value", 12345.6789);
alert(entryB);                                     // displays 12345.6789
entryB = parseFloat(entryB);
alert(entryB);                                     // displays 12345.6789

let entryC = prompt("Enter any value", "Hello");
alert(entryC);                                     // displays Hello
entryC = parseInt(entryC);
alert(entryC);                                     // displays NaN
```

### A more concise way to code these examples

```
const entryA = parseInt(prompt("Enter any value", 12345.6789));
alert(entryA);                                     // displays 12345

const entryB = parseFloat(prompt("Enter any value", 12345.6789));
alert(entryB);                                     // displays 12345.6789

const entryC = parseInt(prompt("Enter any value", "Hello"));
alert(entryC);                                     // displays NaN
```

### Description

- You can use the `parseInt()` or `parseFloat()` method to convert string data to numeric data.
- *NaN* is a value that means “Not a Number”. It’s returned by the `parseInt()` and `parseFloat()` methods when the value that’s being parsed isn’t a number.
- To make your code concise, you can embed one method as the parameter of another. For instance, you can code the `prompt()` method as the parameter for the `parseInt()` or `parseFloat()` method.

Figure 2-14 How to use the `parseInt()` and `parseFloat()` methods

## How to use the write() method of the document object

---

Figure 2-15 shows how to use the write() method of the *document object*. This method writes its data into the body of the document so it's displayed in the browser window.

The example in this figure shows how to use the write() method. Here, an HTML page contains two script elements, each with embedded JavaScript code that uses the write() method to send some data to the document. Then, this figure shows what the HTML and embedded JavaScript look like in the browser.

Each script element in this example is placed at the spot in the HTML that the data it sends to the document should go. So, the first script element writes the current date within a <p> element in the main section, and the second script element writes a copyright notice in the footer. Also note that the JavaScript code in the second script element can access the today object that's in the first script element.

Although not shown here, you can also include HTML elements within the parentheses of the write() method. For instance, you could code the first write() method like this:

```
document.write(`<b>${today.toDateString()}</b>`);
```

Then, the date would be displayed in bold in the browser.

## How to use the toFixed() method of the Number object

---

Figure 2-15 also shows how to use the toFixed() method of the Number object. You'll learn more about the Number object in chapter 4. For now, just know that you can use the toFixed() method with variables and constants of the number data type to round a decimal value and convert it to a string.

The code example below the table shows how this works. Here, a constant named pi is initialized with a decimal value that has five decimal places. As a result, the data type of pi is numeric.

Then, the alert() method is called, and the value returned by calling the toFixed() method of pi is passed as the parameter. The toFixed() method is passed the numeric literal 3, which means that it will round the value of pi to 3 decimal places. The result is that the value of 3.14159 is rounded to 3.142, and the rounded result is displayed in the browser.



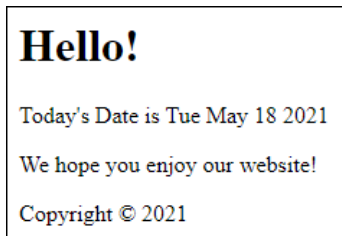
## A method of the document object

Method	Description
<code>write(string)</code>	Writes the string that's passed to it into the document.

## An example that uses the write() method

```
<body>
  </main>
  <h1>Hello!</h1>
  <p>Today's Date is
    <script>
      const today = new Date();
      document.write(today.toDateString());
    </script>
  </p>
  <p>We hope you enjoy our website!</p>
</main>
<footer>
  <script>
    document.write(`Copyright \u00A9 ${today.getFullYear()}`);
  </script>
</footer>
</body>
```

## The output in a browser



## A method of the Number object

Method	Description
<code>toFixed(n)</code>	Rounds a number to <i>n</i> decimal places and converts it to a string.

## An example that uses the toFixed() method

```
const pi = 3.14159;
alert(pi.toFixed(3));           // displays 3.142
```

## Description

- The *document object* is the object that lets you work with the Document Object Model (DOM) that represents all of the HTML elements of the page.
- To output text or HTML directly to your page, place script elements where you want the text or HTML to appear and call the `write()` method.
- The Number object provides methods for working with numeric data. You can use its `toFixed()` method to produce a string that's rounded to a specified number of digits.

Figure 2-15 How to use the `write()` method and the `toFixed()` method

## Two illustrative applications

---

This chapter ends by presenting two applications that illustrate the skills that you've just learned. These aren't realistic applications because they get the user entries from prompt statements instead of from controls on a form. However, these applications will get you started with JavaScript.

### The Miles Per Gallon application

---

Figure 2-16 presents a simple application that issues two prompt statements that let the user enter the number of miles driven and the number of gallons of gasoline used. Then, the application calculates the miles per gallon (MPG) and displays the user's entries and the MPG calculation in the HTML document.

In the JavaScript, you can see how the user's entries are parsed into decimal values and then stored in constants named miles and gallons. After that, miles is divided by gallons, parsed into a decimal, and stored in a constant named mpg. Last, a template literal is used to create a string that contains the user's entries and the calculation within three `<p>` elements. Then, this string is sent to the document using the `write()` method of the document object. Notice that the script element is below the `h1` element so the data is written below that element.

Can you guess what will happen if the user enters invalid data in one of the prompt dialog boxes? Then, the `parseFloat()` method will return `NaN` instead of a number, and the calculation won't work. Instead, the MPG value will display as `NaN`. Unlike other languages, though, the JavaScript will run to completion instead of crashing when the calculation can't be done, so the web page will be displayed.

Incidentally, the dialog boxes in this figure are the ones for a Chrome browser. When you use other browsers, the dialog boxes will work the same but look slightly different.

## The two prompts of the Miles Per Gallon application

The image shows two overlapping browser prompts. The top prompt has the text 'This page says' and 'Enter miles driven' with a text input field containing '375'. The bottom prompt has the text 'This page says' and 'Enter gallons of gas used' with a text input field containing '12'. Both prompts have 'OK' and 'Cancel' buttons at the bottom right.

## The results displayed in the browser

The image shows a browser window with a blue border. The title bar says 'The Miles Per Gallon Calculator'. The content area displays the following text:

```
Miles:    375
Gallons:  12
MPG:      31.25
```

## The body element of the HTML file with embedded JavaScript

```
<body>
  <h1>The Miles Per Gallon Calculator</h1>
  <script>
    "use strict";

    // get miles driven from user
    const miles = parseInt(prompt("Enter miles driven"));

    // get gallons used from user
    const gallons = parseInt(prompt("Enter gallons of gas used"));

    // calculate mpg
    const mpg = parseFloat(miles/gallons);

    const html = `<p><label>Miles: </label>${miles}</p>
                  <p><label>Gallons: </label>${gallons}</p>
                  <p><label>MPG: </label>${mpg.toFixed(2)}</p>`;
    document.write(html);
  </script>
</body>
```

Figure 2-16 The Miles Per Gallon application

## The Test Scores application

---

Figure 2-17 presents a simple application that uses `prompt()` methods to let the user enter three test scores. After the third one is entered, this application calculates the average test score. However, unlike the MPG application, this one uses an external JavaScript file, rather than JavaScript embedded within its script element.

The JavaScript starts by declaring a variable name `total` and initializing it to zero. A variable is used here instead of a constant because it will be changed as the program executes. As you'll see, each time the user enters a test score, it's added to this `total` variable.

Next, the JavaScript has three statements that use `prompt()` methods that ask the user to enter a test score. Each statement converts the user's entry to an integer value and stores it in a constant. The value of that constant is then added to the `total` variable.

Then, the `total` variable is divided by 3 to calculate the average score. The result of that calculation is passed to the `parseInt()` method and stored in a constant. Finally, the constants that hold the user's entries and the test average are embedded in a template literal and sent to the document for display.

Notice that the JavaScript code for this application and the last one starts with the "use strict" directive so they are in strict mode. You should include this directive in all your applications.

### The first prompt of the Test Scores application

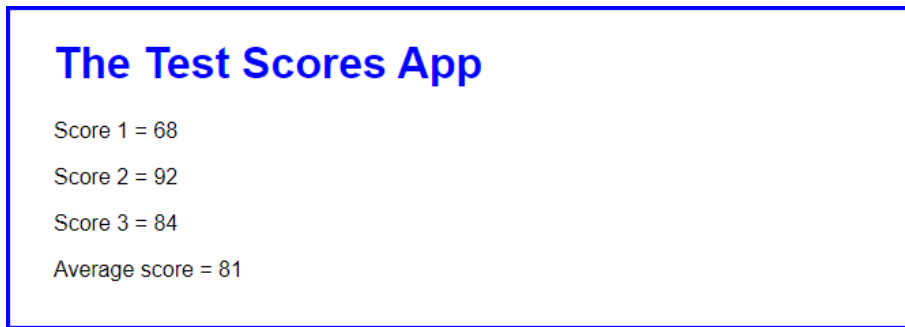


This page says

Enter test score

OK Cancel

### The results displayed in the browser



## The Test Scores App

Score 1 = 68

Score 2 = 92

Score 3 = 84

Average score = 81

### The body element of the HTML with an external JavaScript file

```
<body>
  <h1>The Test Scores App</h1>
  <script src="test_scores.js"></script>
</body>
```

### The test\_scores.js file

```
"use strict";

let total = 0;           // initialize total variable

const score1 = parseInt(prompt("Enter test score"));
total += score1;

const score2 = parseInt(prompt("Enter test score"));
total += score2;

const score3 = parseInt(prompt("Enter test score"));
total += score3;

const average = parseInt(total/3);

const html = `<p>Score 1 = ${score1}</p>
               <p>Score 2 = ${score2}</p>
               <p>Score 3 = ${score3}</p>
               <p>Average score = ${average}</p>`;
document.write(html);
```

Figure 2-17 The Test Scores application

## Perspective

---

If you have programming experience, you can now see that JavaScript syntax is similar to other languages like Java and C#. As a result, you should have breezed through this chapter.

On the other hand, if you're new to programming and you understand all of the code in both of the applications in this chapter, you're off to a good start. Otherwise, you need to study the applications until you understand every line of code in each application. You should also do the exercises that follow.

## Terms

---

external JavaScript file	empty string	modulus operator
embedded JavaScript	Boolean data type	order of precedence
JavaScript statement	Boolean value	compound assignment
whitespace	symbol data type	operator
strict mode	null data type	concatenate
identifier	undefined data type	join
reserved word	bigint data type	concatenation operator
keyword	variable	string expression
camel casing	declare a variable	template literal
snake casing	initialize a variable	escape sequence
comment	assignment statement	Unicode character
block comment	assignment operator	object
single-line comment	literal value	method
comment out	literal	property
uncomment	string literal	call a method
primitive data type	numeric literal	dot operator (dot)
number data type	constant	parameter
integer	hoisting	window object
decimal value	temporal dead zone	global object
floating-point number	(TDZ)	NaN
string data type	arithmetic expression	document object
string	arithmetic operator	

## Summary

---

- The JavaScript for an HTML document page is commonly coded in an *external JavaScript file* that's identified by a script element. However, the JavaScript can also be *embedded* in a script element in the head or body of a document.
- A JavaScript *statement* has a *syntax* that's similar to Java's. When you write JavaScript code, you should use *strict mode* to help you write safer code.
- JavaScript's *identifiers* are case-sensitive and usually coded with either *camel casing* or *snake casing*. Its *comments* can be block or single-line.

- JavaScript provides seven *primitive data types*. The three most common are the *number data type*, which provides for both *integers* and *decimal values*; the *string data type*, which provides for character (*string*) data; and the *Boolean data type*, which provides for true and false values.
- After you *declare* a *variable* or *constant*, you *initialize* it by using the *assignment operator*.
- To assign a numeric value to a variable or constant, you can use *arithmetic expressions* that include *arithmetic operators*, variable and constant names, and *numeric literals*.
- To assign a string value to a variable or constant, you can use *string expressions* that include *concatenation operators*, variable and constant names, *string literals*, and *template literals*. Within a string literal or template literal, you can use *escape sequences* and *Unicode characters* to provide special characters and symbols.
- JavaScript provides many *objects* with *methods* and *properties* that you can *call* or refer to in your applications. Since the *window object* is the *global object* for JavaScript, you can omit it when referring to its methods or properties.
- The *document object* provides methods for working with the DOM, and the *Number object* provides methods for working with numeric data.

## Before you do the exercises for this book...

If you haven't already done so, you should install the Chrome browser and the downloads for this book as described in appendix A (Windows) and appendix B (macOS).

## Exercise 2-1 Test and modify the Miles Per Gallon application

In this exercise, you'll test the MPG application and then modify the JavaScript so it accepts and displays decimal values. When you're done, the browser window should look the one in figure 2-16, but it should work with decimal values:

### The Miles Per Gallon Calculator

Miles driven = 247.93

Gallons of gas = 7.82

Miles per gallon = 31.70

### Test the application

1. Open your text editor or IDE. Then, open the application in this folder:  
`javascript_jquery\exercises\ch02\mpg`

2. Run the application by opening the index.html file in a browser. Then, enter decimal values for the miles and gallons, and note that these values are converted to integers.
3. Run the application again by reloading the file in the browser. Then, enter invalid values like zeros or spaces, and note the result.

### **Modify the application**

4. Modify the JavaScript so it accepts and displays decimal values for the miles and gallons. Then, reload the file in the browser to run the application again, enter decimal values with two or more decimal places, and note the result.
5. Modify the JavaScript so all of the values are rounded to two decimal places. Then, run the application one more time and note the result.
6. If you have any problems when you test your exercises, please use Chrome's developer tools as shown in figure 1-17 of the last chapter.

## **Exercise 2-2      Test and modify the Test Scores application**

In this exercise, you'll test the Test Scores application and then modify the JavaScript code so it uses concatenation operators instead of a template literal to display the output.

### **Test the application**

1. Open your text editor or IDE. Then, open the application in this folder:  
`javascript_jquery\exercises\ch02\test_scores`
2. Run the application by opening the index.html file in a browser, test it with valid entries, and note the result.
3. Open the test\_scores.js file, and note that it starts with the "use strict" directive.
4. Delete the let keyword from the declaration for the total variable, save the file, and run the application again. This time, no dialog boxes will be displayed. That's because you can't declare a variable without using a keyword in strict mode. To see the error that JavaScript threw, open the developer tools and then display the Console panel. When you're done, add the let keyword back to the variable declaration.

### **Modify the application**

5. Modify this application so the value that's assigned to the html constant uses the concatenation operator (+) instead of a template literal.
6. Save the changes and run the application one more time. The output should be displayed just as it was in figure 2-17.



## Exercise 2-3 Create a simple application

Copying and modifying an existing application is often a good way to start a new application. So in this exercise, you'll modify the Miles Per Gallon application so it gets the length and width of a rectangle from the user, calculates the area and the perimeter of the rectangle, and displays the results in the browser like this:

### The Area and Perimeter Calculator

Length: 25

Width: 10

Area: 250

Perimeter: 70

1. Open your text editor or IDE. Then, open the application in this folder: **javascript\_jquery\exercises\ch02\rectangle**
2. Open the index.html file, and note that it contains the code for the Miles Per Gallon application.
3. Modify the code for this application so it works for the new application. (The area of a rectangle is length times width. The perimeter is 2 times length plus 2 times width.) Be sure to change the link element within the head element so it links to the correct CSS file.

# How to build your JavaScript and jQuery skills

The easiest way is to let [Murach's JavaScript and jQuery \(4th Edition\)](#) be your guide! So if you've enjoyed this chapter, I hope you'll get your own copy of the book today. You can use it to:

- Teach yourself how to use JavaScript and jQuery to create websites that deliver the dynamic user interfaces and fast response times that today's users expect
- Pick up a new skill whenever you want or need to by focusing on material that's new to you
- Look up coding details or refresh your memory on forgotten details when you're in the middle of developing a web application
- Loan to your colleagues who are always asking you questions about JavaScript scripting



Mike Murach, Publisher

To get your copy, you can order online at [www.murach.com](http://www.murach.com) or call us at 1-800-221-5528 (toll-free in the U.S. and Canada). And remember, when you order directly from us, this book comes with my personal guarantee:

## 100% Guarantee

*When you buy directly from us, you must be satisfied. Try our books for 30 days or our eBooks for 14 days. They must outperform any competing book or course you've ever tried, or return your purchase for a prompt refund....no questions asked.*

Thanks for your interest in Murach books!

A handwritten signature in black ink that reads "Mike".