

BASICS OF JAVA SCRIPT AND PHP PROGRAM BEGINNERS GUIDE



by STEVE RICHARDS

BASICS OF JAVA SCRIPT AND PHP BEGINNERS GUIDE

By STEVE RICHARDS

Copyrights @1993 by NaSteve Richards

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews

CONTENTS

- 1 FUNDAMENTAL JAVASCRIPT AND CONCEPTS**
- 2 TYPES OF LOOPS IN JAVASCRIPT**
- 3 ENABLING JAVASCRIPT IN THE BROWERS**
- 4 WORKING WITH AN TEXT**
- 5 THE DOCUMENT OBJECT**
- 6 THE EVENT HANDLERS**
- 7 INDROCTION OF NODE.JS**
- 8 MATH NUMBER AND DATA OBJECT**
- 9 HANDLING THE STRING**

1 Fundamental JavaScript And Concepts1

Generally, when we hear the term programming we immediately think of other people typing an incomprehensible string of letters and numbers. Programming looks like magic beyond the realm of mere mortals. Nevertheless, the concepts in programming are not difficult to grasp as they always have real life applications. JavaScript, although it is not as simple as HTML or CSS, is not an overly complicated language. Unlike other languages, its "grammar" is more or less descriptive and intuitive making it a good fit for a first programming language. Basically, learning JavaScript is like learning a new language, but a new language that is similar to English. Once we learn the new words, and understand how to put them together to form "sentences" we'll be good to go.

Syntax

Every language has its own set of characters and words that go along with a set of rules as to how to arrange these characters and words together into a well-formed sentence. These rules are also known as the language syntax and it is the syntax that holds the language together and gives it meaning.

Before start with some examples of JavaScript syntax, let us first set up the environment for JavaScript. As discussed previously, JavaScript code is always a part of the HTML code. Therefore, in order to work with JavaScript we will first need to create a basic HTML document. So to start, let us open a text editor (like Notepad) and type in the HTML code for the most basic web page. In addition to the basic HTML tags, we will include a `<script>` element in the `<head>` section where we will start placing the JavaScript code.

```
<!doctype html>
```

```
<head>
<title>First Steps in JavaScript</title>
<script type="text/javascript">
</script>
</head>
<body>
</body>
</html>
```

Let us save this document as firststeps.html. If we are using Notepad, we have to remember to change the Save as Type field to 'All files'.

Statements

To express ourselves in everyday common language we use sentences as the basic form of communication. Similarly, in JavaScript we also form sentences to express our intentions which are more formally called statements. A JavaScript sentence is the basic unit of communication, usually representing a single step in the program. And just like we put sentences together to express an opinion, we combine statements together to create a program.

Let us look at a simple JavaScript statement and see what it does. Between the opening and closing <script> tag of the html document places the following text:

```
alert("JavaScript is starting to make a little sense.");
```

In further examples we will not show the complete HTML code unless it is necessary, but for initial reference your document should look like the following:

```
<!doctype html>
<head>
<title>First Steps in JavaScript</title>
<script type="text/javascript">
alert("JavaScript is starting to make a little
sense.");
```

```
</script>
</head>
<body>
</body>
</html>
```

We can save the firststeps.html document and open it in a web browser. Once the page opens, we will get an alert window with the message "JavaScript is starting to make a little sense".

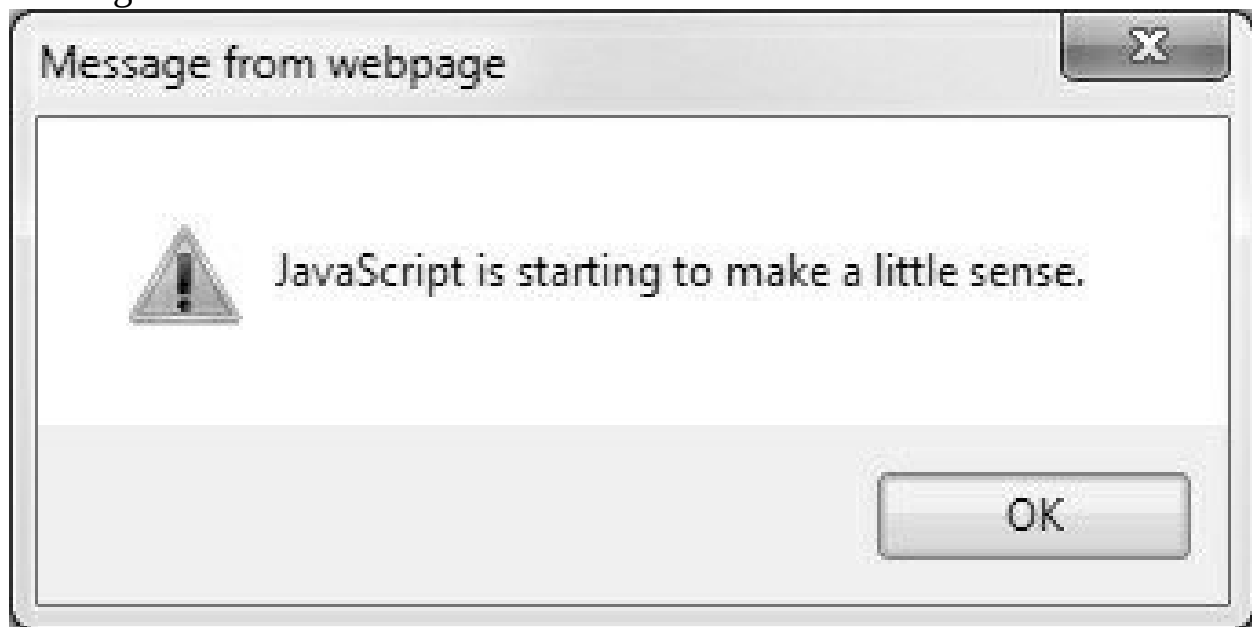


Image 1. Alert window

Now that we know what the effect is, let us go back to the JavaScript statement and interpret it into common language so it makes more sense.

```
alert("JavaScript is starting to make a little sense.");
```

JavaScript statements are instructions which are executed by the web browser. The statement starts with a command, presented by keyword. The keyword identifies the action that needs to be performed. In this case the keyword `alert` makes the web browser open a dialog box and display a message. If we just had the statement `alert();` the dialog box would have been empty, however in our case the statement consists of a specific input, the actual message text, also known as an argument. Finally, just like every sentence ends with a period, a JavaScript sentence ends with a semicolon. The semicolon makes it clear that the statement is over and once the

interpreter executes it, it should move on to the next item.

Now we are ready to translate the JavaScript statement. Its plain English interpretation would be, "Open a dialog box and display the text JavaScript is starting to make a little sense' in that box."

Note: When passing text arguments, we can use either double quote marks ("sense") or single quote marks ('sense') present the text.

Before we move on, let us look at another JavaScript statement. In the `<script>` element replace the previous code with the following and preview it in a web browser to see the results: `document.write("<p>JavaScript is starting to make a little sense.</p>");`

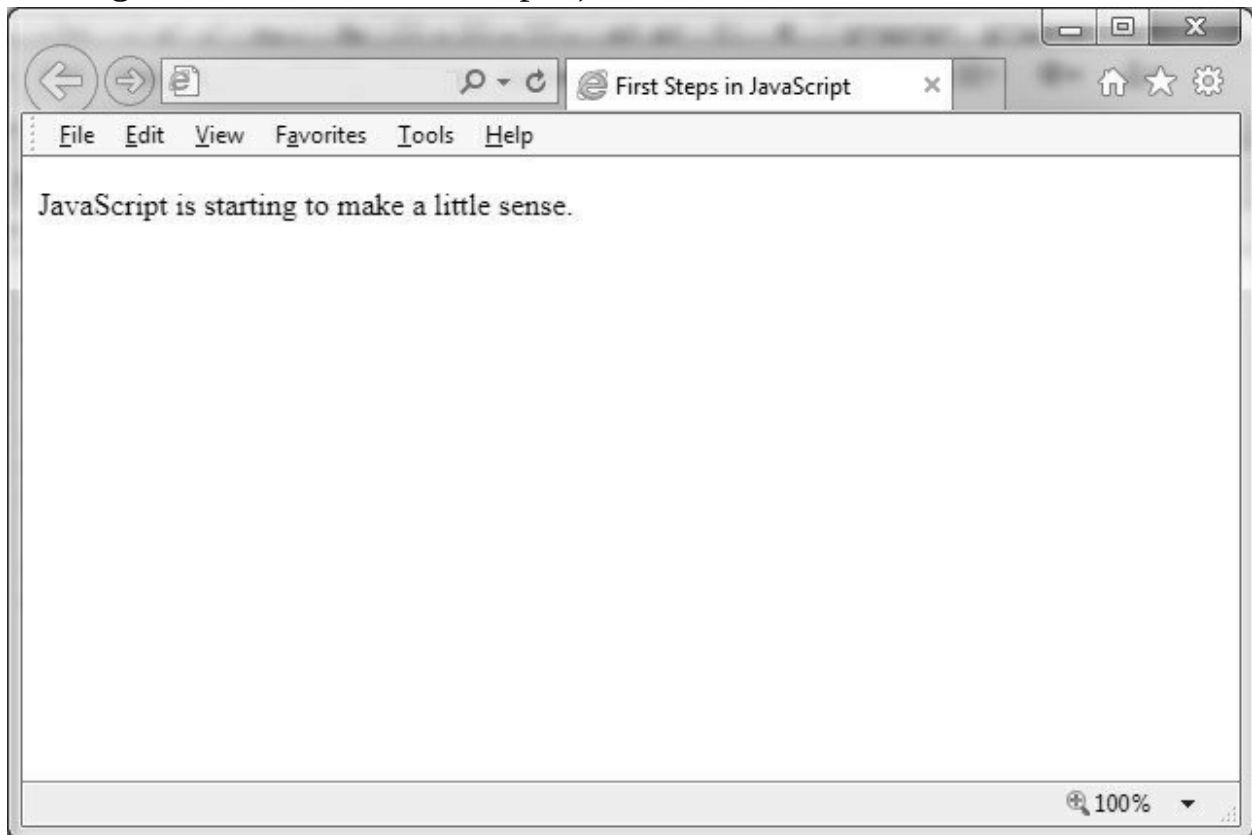


Image 2. Example of a document.write statement

What we can see from the results is that the previously empty document, now has one paragraph of text. Following the previous interpretation of how JavaScript works and from the web browser results we can correctly assume that the `document.write` keyword commands the browser to write directly onto the web page. Similar to `alert()`, it writes whatever is placed between the opening and closing brackets.

Variables

One of the fundamental aspect of JavaScript, and any programming language in general, is the concept of variables. A variable is a way to declare and store information which can later be used. This information can vary with the circumstances and hence the name variable.

Let us look at the following

statement. `var name =`

`"Martin";`

In plain language this is the same as saying "My name is Martin". The keyword `var` is JavaScript speak for "create a variable", or in a programming dialect, "declare a variable". What follows is the name of the variable which can be anything we choose with certain limits. Assigning a value to the variable is done with the `=` sign, which is not immediately necessary, as this can happen later. We can declare an undefined variable in one statement and assign it a value in a later statement. For example:

`var`

`name;`

`name`

`=`

`"Martin";`

As mentioned previously, variable names can be anything, like `name`, `abc`, `R2D2`, with a few rules. Variable names can contain letters, numbers, dollar sign (`$`), or lower line (`_`), other special characters are not allowed.

Furthermore, a variable name cannot begin with a number, any other allowed value is acceptable. Finally, variable names are case-sensitive, meaning that the interpreter in the web browser makes a distinction between uppercase and lowercase letters, making `'score'` different from `'Score'`.

Note: Although we can use almost anything for a variable name, it is wise to use names which are meaningful as this will help us and other programmers to better understand the written code.

Variable Types

Based on the type of data, variables come in different flavors. The three most basic types are number, string, and boolean.

A number variable is represented by a numeric character. This variable can accept whole integers, negative integers and fractional integers. Numbers are frequently used in calculations, hence our number variables are often included in mathematical operations. The following statement declares a variable named age and assigns it a value of 35.

```
var age = 35;
```

A string variable is used to represent any series of letters like words or sentences. Strings are represented as a series of characters enclosed within quotation marks, with the quotation marks signaling to the interpreter that what follows is a string variable. JavaScript allows us to use both the double quotes (") or the single quote (') marks, but we have to be mindful to use the same type of quotation mark. `var location = "California";`

A boolean variable is rather simple as it can accept only one of two values: true or false. This variable is used when we create JavaScript programs that we want to intelligently react to user actions.

User Variables

JavaScript would not be fun if it didn't allow us to share our thoughts and create or alter the variables directly. One of the simplest ways to "give" our input is to use the `prompt()` command.

```
var name = prompt("What is your name?", "")  
document.write(name);
```

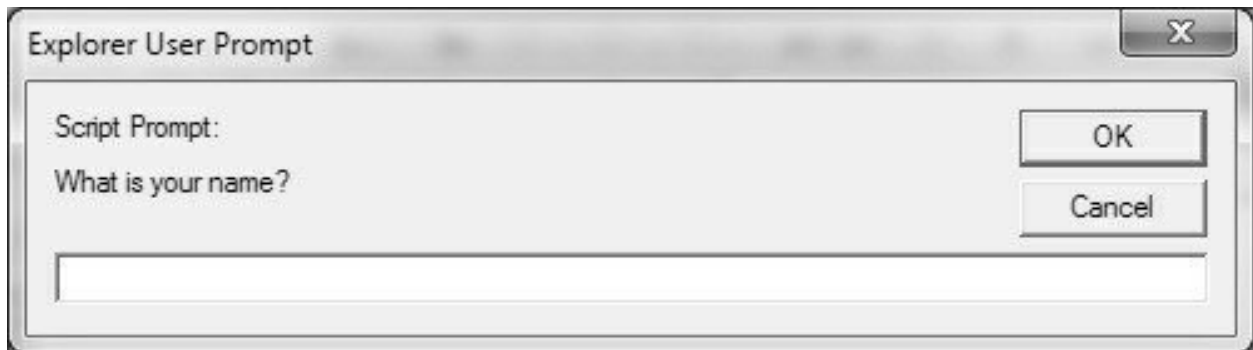


Image 3. A custom prompt dialog box

The result of the `prompt()` command is a dialog box. Instead of just displaying a message like the alert dialog box, the prompt dialog box can also receive an answer. Hence, in the syntax for a prompt dialog box it is necessary to provide two arguments between the parentheses separated with a comma. The first argument is the prompt text that is displayed in the box, while the second argument is the default value for the text box, and consequently, the variable.

In the example above, the prompt text displayed in the box will be 'What is your name?' and the default value presented in the box will be empty, as there is obviously no content between the quotation marks. Once we type in something in the box and either click OK or press the Enter key, the variable receives the value that was entered in the field. Consequently, the name will be displayed on the web page. Otherwise, if we click on Cancel, press the Esc key or close the prompt box, the returned value would be empty and there would be no text on the screen.

Note: Instead of `prompt()` we can also use the more formal `window.prompt` command. Operators

Storing information in a variable is a first step, the beauty of programming is the ability to manipulate this information in many creative ways. For this,

JavaScript provides different operators that allow us to modify data. An operator, represented by a symbol or a word, is something that can change one or more values into something else. The type of operators available are different based on the data type.

Mathematical Operators

The basic mathematical operators like addition (+), subtraction (-), multiplication (*) and division (/) are readily available in JavaScript. They can be used in independent statements or used when declaring variables. For example, by "operating" with the variables `currentYear` and `yearOfBirth`, we can determine the value of the variable `age`.

```
var currentYear = 2015;  
var yearOfBirth = 1979;  
var age = currentYear -  
yearOfBirth;  
document.write(age);
```

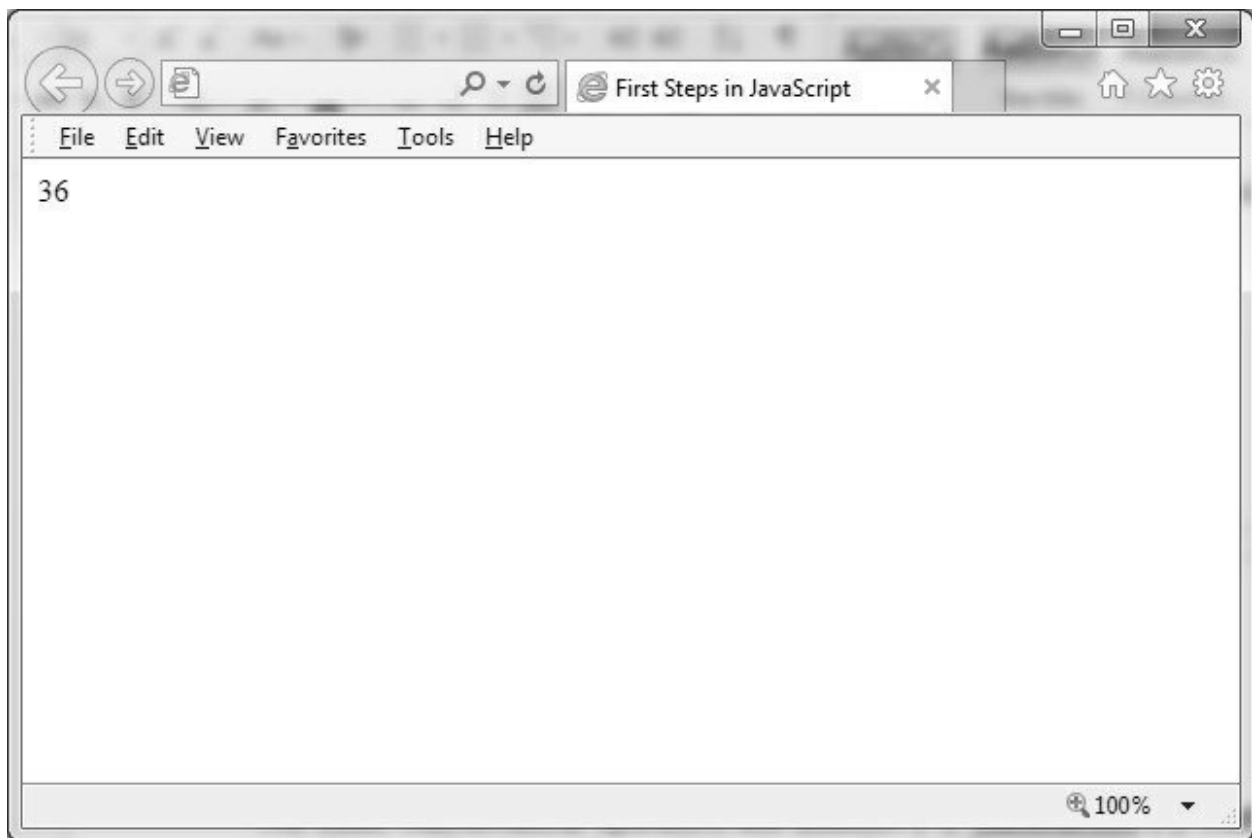


Image 4. Using variables to calculate age

Note: "Calling" a variable to be presented on a web page is easy. Simply use the `document.write()` command.

Mathematical operators, specifically the addition operator, can be used to combine two or more strings. This process of combining strings is called concatenation. In the following example:

```
var firstName =  
"Martwan"; var lastName  
= "Jenkins"; var fullName  
= firstName+lastName;  
document.write(fullName);
```

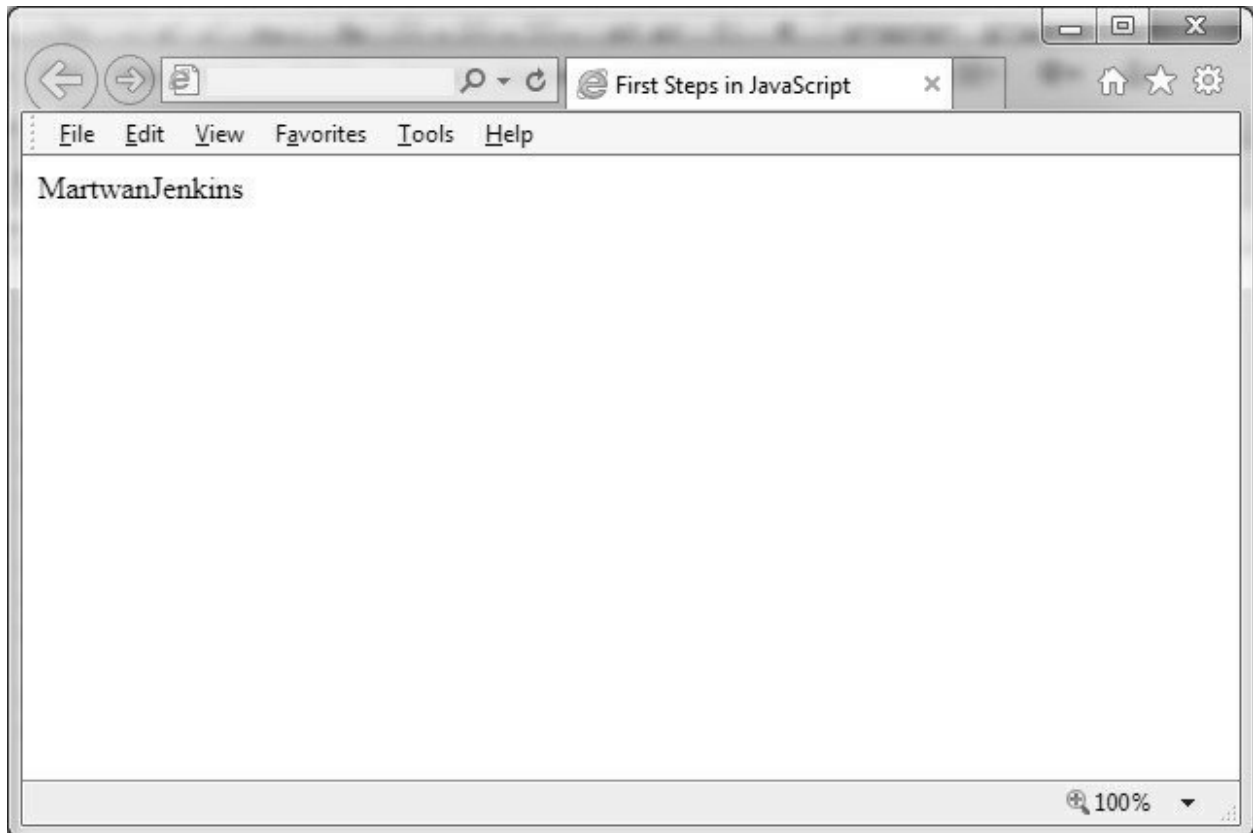


Image 5. Concatenating strings into a full name

The value for `fullName` will end up being `MartwanJenkins`. To make sure that everything is in its proper form we need to include the empty space as a string in quotation marks. For example, we can use the following declaration:

```
var fullName = firstName+" "+lastName;
```

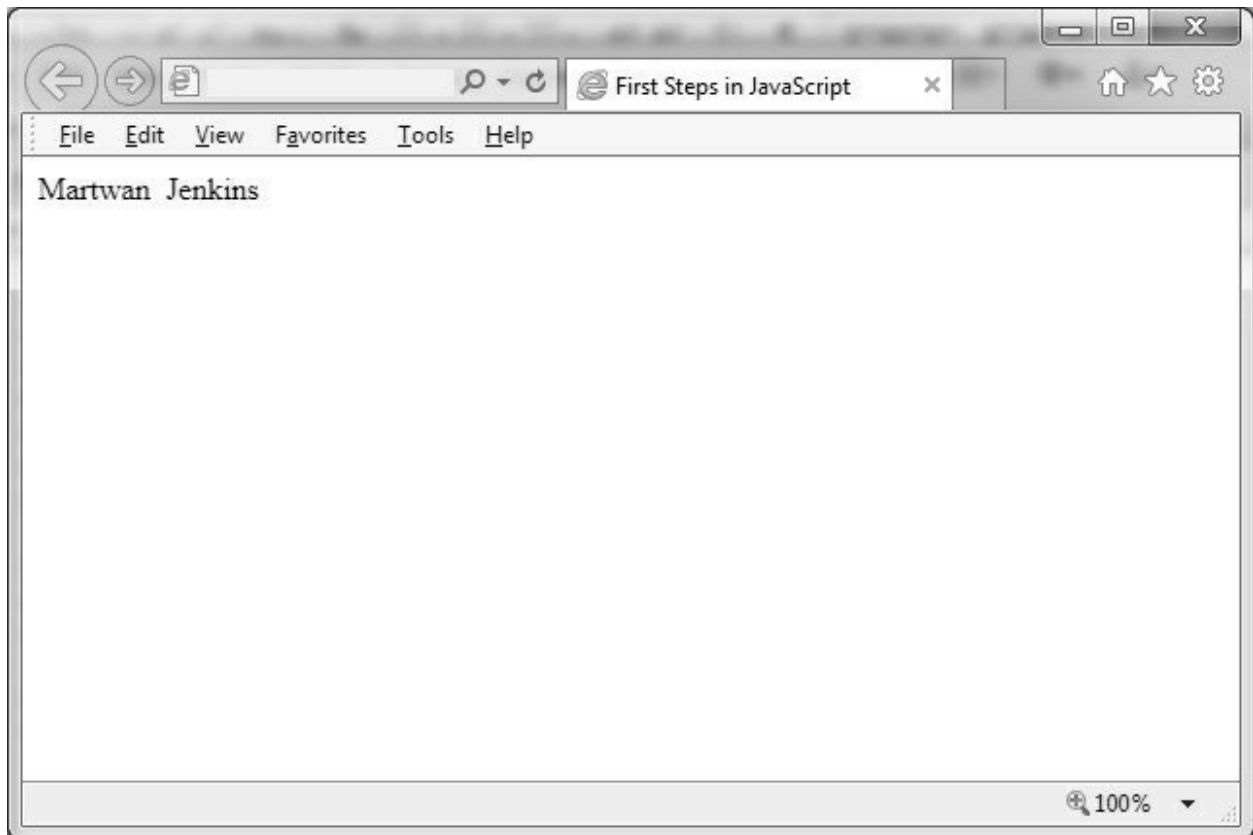


Image 6. Concatenating strings with spaces

We can see that operators are also useful when we want to join text and/or combine variables. As a matter of fact, we can use this to construct more logical sentences. For example, we can combine the "My name is" text with a value from a calculated variable. `var firstName = "Martwan"; var lastName = "Jenkins"; var fullName = firstName+" "+lastName; document.write("My name is " +fullName);`

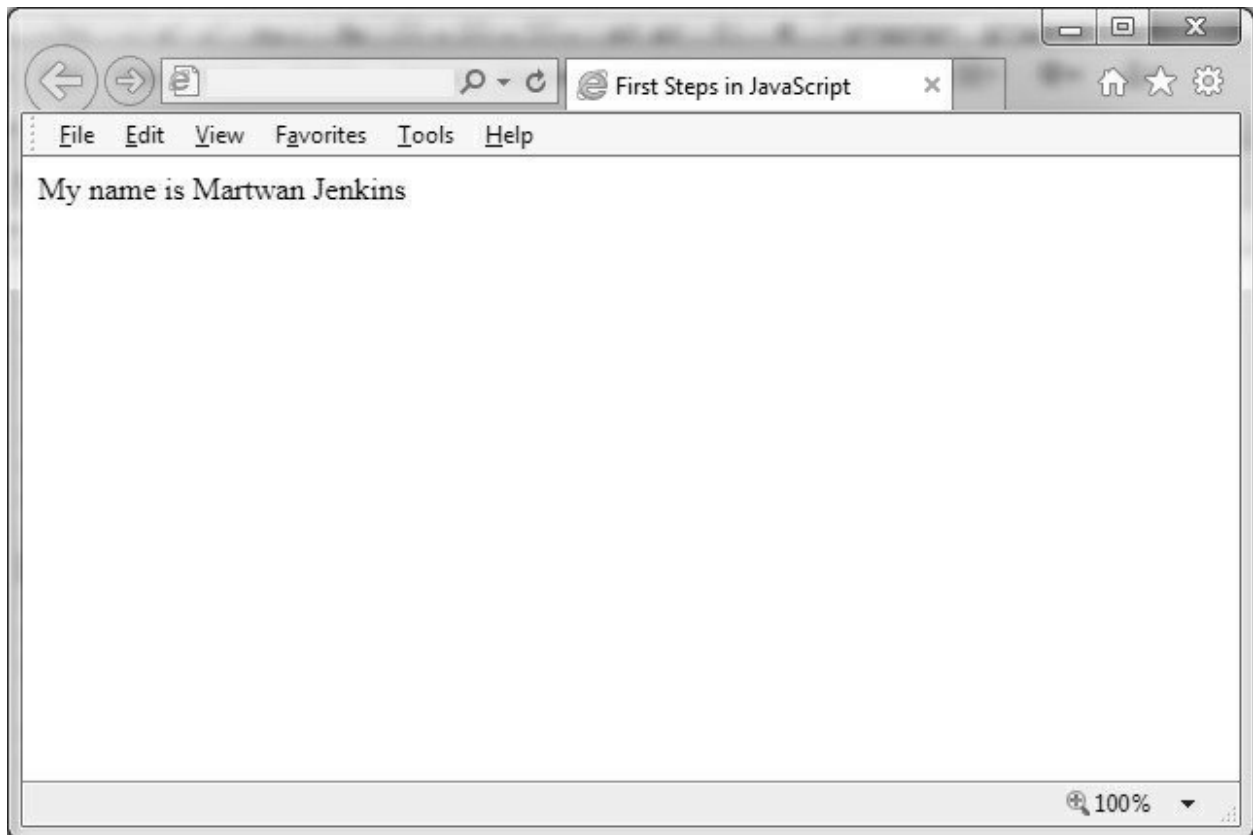


Image 7. Concatenating a full sentence

Note: When performing several mathematical operations in one statement, the rules of precedence apply.

Assignment Operators

Just like things change in real life, so do variables within a JavaScript program. And to change variables within JavaScript we will use assignment operators. We are already familiar with the fundamental assignment operator, the equal sign (=), which is used to give an initial or a new value to a variable. There are other assignment operators that also change the value of a variable, but they do this in a slightly different way.

For example, as the year passes we grow older and our age incrementally changes by one. To make this change in JavaScript there are several different approaches we can take, all with the same results. To play around with the possibilities of changing variables, we can try the following code where after the age variable changes its value is displayed in the browser:

```
var age = 35;
document.write("<p>My age is "+age+"
</p>"); age = age + 1;
document.write("<p>A year has passed, so now I am " +age+"</p>");
age += 10;
document.write("<p>What? Are you telling me that I am a grandad now?
But
I am only " +age+"</p>");
```

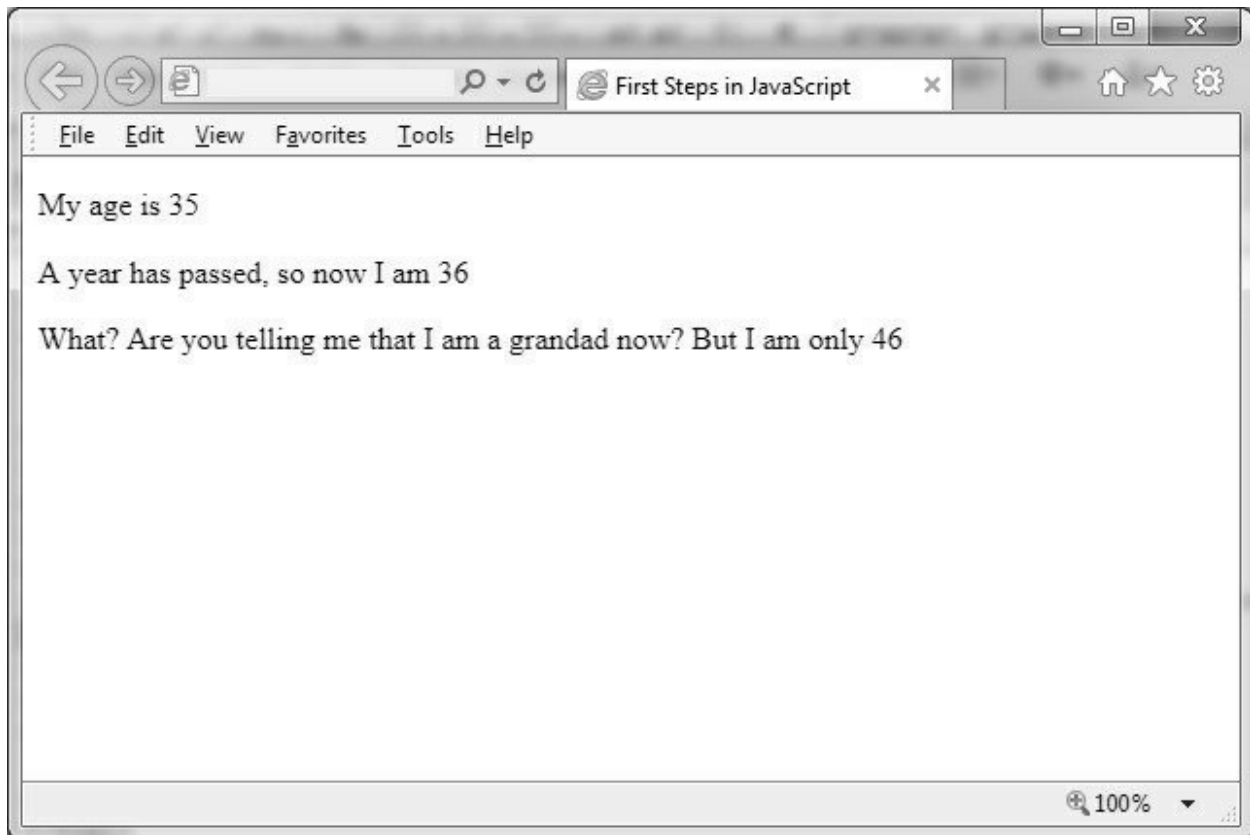


Image 8. Changing variables with assignment operators

While at first these operations might appear slightly confusing, they are still logical if viewed from the programming angle. For example, if we read the statement `age=age+1` backwards what happens is that the value of 1 is added to the current age of 35 which would make 36 the new value of age.

Additionally, we can use a complex assign operator such as `(+=)` in the statement `age+=10`, to increase the value of the variable by 10. We can also use the same logic to other operations like subtraction, division and multiplication.

`age-=5`; is the same as `age = age`

`- 5`; `age*= 5`; is the same as `age`

`= age * 5`; `age/=5`; is the same

as `age = age / 5`;

Additionally, when we want to increase/decrease the value of the variable by 1, we can also use the following assignment operators:

`age++`; is the same as `age = age +`

`1`; `age--`; is the same as `age = age`

- 1;

Comparison Operators

Like the name suggests comparison operators are used to compare two values. After the comparison is made a value of either true or false is returned depending on whether the comparison was exact or not.

The following table summarizes the most common comparison operators.

OPERATOR	MEANING	EXAMPLE	RETURN VALUE
==	equal to	4==4	True
>	greater than	3>4	False
<	less than	3<4	True
>=	greater than or equal to	4>=4	True
<=	less than or equal to	4<=5	True
!=	not equal to	3!=4	True

Table 1. Comparison operators

HTML Overview

HTML, CSS, and Javascript work in tandem to give users a seamless web experience. A lot of budding web developers have these three confused. If you're completely new to the world of web development, this chapter should give you a quick tour of HTML. On the other hand, if you already know HTML, feel free to skip or skim this chapter.

Choosing a Text Editor

Before you start creating interactive webpages, you'll have to download a good text editor. While Macs and PCs usually have built-in text editors that can save files in html format, they lack a lot of useful utilities that make web development a lot easier like syntax highlighting and auto-completion. That being said, here are some of the best text editors for web development:

1. Atom – Atom is a relatively new text editor by Github. It's free, open source, and has plenty of customization options. Basically, Atom is like a programmable text editor; it can do almost anything you want it to with the right plug-ins. Among its many features would be the Fuzzy finder, which allows you to jump to different files with a simple cmd+t command, and the wide range of available themes. It works on Mac OS X, Windows, and Linux and comes at no cost, so try it out and see if suits you.
2. Sublime Text – Sublime Text is quite aptly named, as it offers a sublime writing experience due to its intuitive commands and beautiful interface. Like Atom, Sublime Text is also highly customizable and works on Mac OSX, Windows, and Linux. Its trial version is free, but the full version costs about \$70.
3. Notepad++ - Notepad++ has been a crowd favorite for quite some time because of its unrivaled simplicity and speed. If you just need a no-nonsense text editor that'll let you edit a couple of html files without the need for complex file structures, you may opt to give this text editor a try. Keep in mind, however, that this text editor is only available in Windows.
4. Brackets – Brackets is quite a unique text editor from Adobe, as it's designed to work in tandem with Adobe's other products, specifically Photoshop. If you're interested in creating sleek user interfaces from Photoshop, Brackets allows you to extract information seamlessly from the PSDs you've created and convert them into CSS code. This

text editor works with Windows, Linux, and Mac OSX, so if you're interested in web design, you'll want to have this installed in your system.

If you're not trying to save space on your system, I recommend downloading these four text editors and trying them out for yourself, assuming that you're running a compatible operating system. The exercises and code snippets in this book do not require a specific text editor so feel free to jump in and out of different text editors until you find one that fits your needs.

What is HTML?

HTML basically allows you to present your site's content in a structured manner. Think of it as a blueprint for building a house; it tells you where the living room, bathroom, kitchen, and other rooms will be placed and how they harmoniously interconnect.

Learning the structure of an HTML file is quite simple; it's structured almost like an essay - you provide a title, a skeletal structure, and content that flows coherently within the structure. Whenever you read an article, the title tells you what the article talks about, and the content is structured in a particular way, oftentimes with a sentence that piques your interest, followed by anecdotes, facts and figures, arguments, and a conclusion. When you start coding in HTML, you'll see how similarly structured it is to articles you read or write.

Creating an HTML file, then, is as simple as saving your file with a .html file extension. When you create documents in a word processor like Microsoft Word, you'll notice that when you save your file it has either an .odt, .doc, .docx, or something similar at the end of the filename. This tells the computer to read and format the contents of the file in a very specific way. This is because whenever you save something you've created using a word processor you're not just saving a bunch of words you've typed; you're also saving the font size, font styles, and other formatting styles you've used so that when you open your file, your computer will check the file extension, read the contents according to the file extension specified, and then present you the

file you've created, making sure that it looks exactly the same way when you've saved it.

HTML basically acts the same way, but this time you use tags in order to tell the computer what to do with the data enclosed by the tags; this allows you to easily change the font styles and other aspects of your web page. Think of HTML as a more precise version of a Word Document. For example, take a look at the text below:

This is a sentence.

If you save this as a Word Document, it will save not only the text, but also its font style, size, and structure. The attributes of the document you've just saved is tucked neatly inside the file and accessible only through the word processor menus. However, if you want to display the same sentence in HTML form, the HTML file would look like this:

```
<html>
  <head>
  </head>
  <body>
    <p>This is a sentence.</p>
  </body>
</html>
```

By saving this file with a .html extension, you get a basic webpage that displays "This is a sentence." Notice how it has the same string of text as the previous text box, but it contains additional tags you may or may not be familiar with. When your browser opens this file, it understands the tags and displays only the content on the previous text box; it doesn't actually show the tags to the user. The most basic tags are shown in the code example:

- The <html> tag - contains everything in your web page. It's the alpha and the omega, literally; never forget the <html> tag or else your site won't work properly!
- The <head> tag - keeps the title and other preliminary scripts that the body might use. For instance, if you want to use a set of design protocols (typically in the form of CSS) that

dictate what type of fonts to use, their colors, etc., you can place the code or its reference here.

- The `<body>` tag - this is where your content goes. From text, to sound, to images, to videos, this is where everything happens.

Don't worry too much if you don't yet fully understand how these tags work; you'll learn better how they work by trying them out in future sample codes and exercises.

HTML Essentials

Now that you have a general overview of how HTML works, we're going to have a quick run through the basics, just enough to help you learn how to integrate JavaScript seamlessly into it.

We've discussed how HTML files are like Word Documents with additional tags, so let's start with a simple line of text. Type this into your text editor:

Hi There!

Now create a folder in your computer where you can store all the html files are created and save what you've just typed with a .html extension (e.g., "My First Website.html"). A standard practice for programmers is to create a folder called 'Developer' -- this is where all your files that pertain to programming are stored. You can place this in your home folder. Inside the Developer folder, create a folder called 'Web Development' and for now, place all your html files here. If you're using TextEdit or other text editors that aren't built specifically for html files, remember to check the settings and make sure that "Plain text" format is selected.

Now that you've created your first html file, open it using any web browser you have and see a simple string of text appear before your very eyes!

Notice, however, that what you've typed into the text editor (referred to as "source code") looks just like what the browser displayed; it didn't have the `<html>` tags and other stuff. What gives?

Now, try changing your html file by adding a few tags. It should then look like this:

`<DOCTYPE html`

```
<html>
  <body>
    Hi There!
  </body>
</html>
```

Now reload your browser and see what happens.

Your browser isn't malfunctioning; the web page looks pretty much the same despite the addition of tags in the source code. At this point you may not fully understand why tags are used, but don't worry; you'll see what they're for when you start programming in HTML more.

Conclusion

You've just had your first dive into HTML- nothing too thorough, just enough of the basics to help you use JavaScript with HTML properly. If you're interested in learning more about HTML, you'll want to take a look at online tutorials and references, as well as see how other web developers code their own sites. You can do the latter by right-clicking a site and clicking on 'view page source' or something along those lines. If you're ready to take the plunge into JavaScript, head over to

JavaScript's Control Flow Statements

This language offers a compact group of control flow statements that you can add to your codes. In general, a control flow statement boosts the functionality and interactivity of a JavaScript program. This section of the book focuses on control flow statements.

The Block Statement

Programmers consider this as the most basic statement in JavaScript. Use this to group your JavaScript statements. When writing your programs, enclose the block statement using a pair of brackets.

```
{  
  statement_1;  
  statement_2;  
  .  
  .  
  .  
  statement_n;  
}
```

The image below shows the syntax of a block

statement:

In general, programmers use block statements for control flow statements (i.e. for, if, while, etc.). Here's an example:

```
while (y > 69) {  
    123;
```

In the code snippet given above, "123;" serves as the block statement.

The Conditional Statements

Conditional statements are sets of commands that run if the assigned condition is true. The JavaScript language offers two kinds of conditional statements: (1) if... else and (2) switch.

Let's discuss these conditional statements in detail:

The “if... else” Statement

With this statement, you can run a command as long as the assigned condition is true. It involves an “else” clause (which is optional). The “else” clause holds a statement that will run if the assigned

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

condition is false. The syntax of a basic “if... else”

statement is:

Since your condition must get either “true” or “false,” you must use Boolean expressions when writing a conditional statement. If the condition is true, the first statement will run; otherwise, the second one will run.

JavaScript allows you to set multiple conditions in your conditional statements. You just have to combine several “if” and “else” clauses in a conditional statement. Here’s the syntax that you must

```
if (condition_1) {  
    statement_1;  
} else if (condition_2) {  
    statement_2;  
} else if (condition_n) {  
    statement_n;  
} else {  
    statement_last;  
}
```

use:

To run multiple statements, you must place them inside a block statement (i.e. { ... }). Programming experts claim that it’s best to utilize block statements, particularly when combining if statements.

```
if (condition) {
    statement_1_runs_if_condition_is_true;
    statement_2_runs_if_condition_is_true;
} else {
    statement_3_runs_if_condition_is_false;
    statement_4_runs_if_condition_is_false;
}
```

Check the following example:

If possible, avoid using simple assignments in your conditional expressions. This is because simple assignments are often confused with equality when checking JavaScript codes. For instance, you must not add the following code into your statements:

```
if (a = b) {
```

```
/* sample_statement */
```

If you really need to use a simple assignment in your conditional expressions, place it between a pair of parentheses. Here's an example:

$$\text{if } ((x = y)) \{$$

```
/* sample_statement */
```

The Falsy Values

A “falsy value” is a value that evaluates to false. Here are the falsy values that you will encounter while using JavaScript:

- -
 -
 -
 -
- null
- false
- undefined
- NaN
- (" ") (i.e. an empty string)

Any value not listed above evaluates to true when used in conditional statements.

The “Switch” Statement

This kind of statement allows an application to check an expression and compare it with a predetermined set of cases. If there’s a match, the application runs the assigned command. Here is the syntax of a basic switch statement:

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

First, the application will search for a case with a label that matches the expression’s value. Then, it will pass the control flow to that particular clause, performing the assigned statement/s. If no match is found, the application will check the default clause (which is optional) and pass the control flow to that clause, running the assigned statement/s. If the statement has no default clause, the application will simply run the statement after the switch. Typically, the final clause of a switch statement serves as the default clause, but it isn’t mandatory.

You may also add a break statement in your switches. A break statement makes sure that the application will get out of the switch statement once a match is found. When the application gets out of the switch, the system will run the statement right after the switch statement. If your code doesn’t have a break, the application will simply run all of the commands within your switch statement.

How to Handle Exceptions

JavaScript allows you to throw and manage exceptions. To do these things, you must use the following statements:

the “throw”

- statement the
 - “try... catch”
- statement

The Different Types of Exceptions

In JavaScript, you can throw any type of object. However, thrown objects may possess different characteristics. Although it is usual to throw strings and/or numbers as error messages, it is usually more beneficial to utilize the exception types designed for this purpose. Here are two of the most popular exceptions in JavaScript:

- DOMError – This exception represents a named error object.
- DOMException - This exception indicates an abnormal situation that happens while a property or method is being used. To use this exception, you must add DOMError() into your code.

The “throw” Statement

Obviously, this statement allows you to throw exceptions. While throwing an

```
throw expression;
```

exception, you must indicate the expression holding the data to be thrown. Here’s the syntax:

The syntax given above can throw any type of expression. The following list shows some examples:

- throw “Error1”; // A String type exception
 - throw 1; // A Number type
 - exception throw false; // A Boolean type
- exception

Important Note: You may specify objects while throwing an exception. Then, you may specify the properties of the object/s inside the “catch” section of your statement.

The “try... catch” Statement

This statement looks for statement blocks that you can try. It also specifies a

response that will be used in case an exception is thrown. The try... catch statement catches thrown exceptions.

Try... catch statements have three parts, which are:

1. The “try” section – It holds one or more JavaScript statements. This part is mandatory.
2. The “catch” section – This part holds statements that indicate what the program must do in case an exception gets thrown.

That means if the “try” section doesn’t succeed, the program control will go to the “catch” section. If a statement inside the “try” section throws an exception, the program control will immediately go to the catch section. If no exceptions arise, however, the system will skip the statement’s catch section. In JavaScript, you may use the catch section to manage all exceptions that

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

may be thrown in the try section. Here’s the syntax that you should use:

In the syntax given above, “catchID” represents the identifier that contains the value assigned in your throw statement. You may employ this identifier to acquire data regarding the thrown exception/s. JavaScript generates this identifier whenever the catch section executes. The identifier lasts only while the catch section is active. That means the identifier will cease to exist once the catch section has finished its execution.

For instance, the code below generates an exception. Once the exception is

```
1 try {  
2   throw "myException"; // generates an exception  
3 }  
4 catch (e) {  
5   // statements to handle any exceptions  
6   logMyErrors(e); // pass exception object to error handler  
7 }
```

thrown, program control will go to the catch section.

Important Note: This section is completely optional.

3. The “finally” section – It holds statements that will run after executing the previous sections. The statements placed inside the “finally” section will run whether or not the program encountered an exception. Additionally, statements within this section will still run even if the statement has no “catch” section.

In general, programmers use this section to set graceful exits for their projects whenever an exception happens. For instance, they may need to liberate some resources that are affected by the

```
1 openMyFile();
2 try {
3   writeMyFile(theData); //This may throw a error
4 } catch(e) {
5   handleError(e); // If we got a error we handle it
6 } finally {
7   closeMyFile(); // always close the resource
8 }
```

problematic script. The
image

below will help you understand how this section works.

With the code given above, JavaScript opens a file and runs statements that utilize the file. If JavaScript encounters an error, the “finally” section will close the file before the whole code terminates.

How to Utilize an Error Object

In some cases, you may get more information about an error by checking its properties (i.e. its ‘name’ and ‘message’). This functionality is extremely useful if your code’s catch section doesn’t indicate whether errors are generated by the user or the system itself. To utilize error objects, you

```

1 function doSomethingErrorProne () {
2   if (ourCodeMakesAMistake()) {
3     throw (new Error('The message'));
4   } else {
5     doSomethingToGetAJavascriptError();
6   }
7 }
8 ....
9 try {
10  doSomethingErrorProne();
11 }
12 catch (e) {
13  console.log(e.name); // logs 'Error'
14  console.log(e.message); // logs 'The message' or a JavaScript error
15 }

```

have to use a JavaScript tool called “Error Constructor.” Here’s an example:

JavaScript Promises

The JavaScript language now supports “promises” (i.e. an object that allows programmers to manage the flow of delayed and asynchronous processes).

Promises undergo the following states:

- Pending – This is the initial status of a promise. In this state, a promise may get rejected or fulfilled.
 - • Fulfilled – A promise becomes “fulfilled” if the
 - system successfully completes all of the assigned processes. Settled – A settled promise can be either rejected or fulfilled.

Rejected – In a rejected promise, the system failed to complete the assigned processes.

2 Types of Loops in JavaScript

A loop allows you to repeat code snippets quickly and easily. This chapter will concentrate on the different kinds of loops offered by JavaScript.

General Information

While writing your own codes, you'll find situations where you have to use the same codes over and over again. This kind of task is boring and timeconsuming. However, you can't skip it since your application won't work properly without the proper codes.

Fortunately, JavaScript supports loop statements. A loop statement allows you to repeat code blocks automatically. That means you won't have to spend much time and effort in completing your JavaScript codes.

Loops are designed to repeat codes. They have different mechanisms that can be adjusted to improve their effectiveness in copying codes. Additionally, each loop type has distinct characteristics that make them extremely useful and convenient in certain situations. Here are the loop statements that you'll find in JavaScript:

- “for”
- “while”
- “do... while”
- “label”
- “break”
- “for... of”
- “for... in”
- “continue”

The “for” Loop

Basically, “for” loops repeat a statement until an assigned condition results to

```
for ([initialExpression]; [condition]; [incrementExpression])  
  statement
```

false. The syntax of JavaScript “for” loops is similar to that of C and Java. Here’s the syntax:

The following things happen whenever a “for” loop runs:

1. The loop’s initializing expression (represented by `initialExpression` in the syntax above) performs its function/s. Often, an initializing expression triggers multiple loop counters. That means the syntax given above accepts different levels of complexity. You may also use this expression to declare variables.

Important Note: The initializing expression is optional; thus, you can create effective “for” loops without using any initializing expression.

2. The system evaluates the conditional expression. If the evaluation is true, the statements contained in the loop will run. However, if the evaluation is false, the entire loop will stop.

Important Note: The conditional expression is optional. If you won’t add a conditional expression in your “for” loop, the system will assume that the evaluation is true.

3. The statement runs. If you want to run multiple statements, you must group them into a block (i.e. by enclosing them in curly braces).
4. The final expression (represented by `incrementExpression` in the syntax) updates the entire loop, then the control flow will go back to the second step (see above).

The “while” Loop

“While” loops execute their statement/s while the assigned condition results

```
while (condition)
  statement
```

to “true.” Here is the syntax of a “while” loop:

Once the condition results to “false”, the loop will stop running the assignments assigned to it, then the control flow will go to the statement right after the “while” loop.

The system evaluates the condition prior to running the loop itself. If the condition is true, the statement/s inside the loop will run and the system will check the condition again. If the condition is false, the loop will stop and the control flow will move on to the next statement.

Important Note: Since the condition test happens first, a “while” loop may never run. That means you have to be extremely careful when creating “while” loops in your JavaScript programs.

JavaScript allows you to enclose multiple statements using a pair of curly braces. Use this option if you want to include various statements in your “while” loops.

The “do... while” Loop

This kind of loop runs its statement/s until the assigned condition becomes

```
do
  statement
while (condition);
```

false. When creating a “do... while” loop, you must use the following syntax:

Here, the system executes the statements once before checking the assigned condition. If the condition is true, the system will execute the statements again. This process will continue until the condition becomes false. Once the condition evaluates to false, the control flow will go to the statement right after the “do... while” loop.

In this type of loop, the statements are guaranteed to run at least once even if the condition is false.

The “label” Statement

This kind of statement allows you to “label” (i.e. assign an identifier) your statements. By labeling your JavaScript statements, you can easily repeat them at any part of your application. For instance, you may label a certain loop, and utilize other statements to either continue or disrupt its execution.

```
label :  
    statement
```

Here is the syntax of a “label” statement:

When creating a label, you may use any identifier that isn’t a JavaScript keyword. You may use the syntax given above for any statement.

The “break” Statement

Programmers use this statement to end loop or switch statements. When using a “break” statement, you must remember the following rules:

- If your break statement doesn’t have a label, it will terminate the innermost loop/switch and pass the control flow to the next statement.
- If your break statement has a label, it will terminate the labeled statement.

```
1. break;
```

```
2. break Label;
```

The syntax of a break statement is:

Use the first variant of the syntax if you don’t need to specify a label. This will terminate the innermost loop/switch. Use the second variant if you want to terminate a certain loop/switch. Just enter the label of that particular loop or switch. When using the second variant, only the specified loop/switch will be terminated.

The “for... of” Statement

This statement is a recent addition to the JavaScript language. It creates a loop that can repeat “iterable objects” (e.g. maps, sets, arrays, arguments, etc.). It also invokes an iteration hook that will run for each distinct property of an object. JavaScript allows you to customize the iteration hook of

```
for (variable of object) {  
    statement  
}
```

any “for... of” statement. Here is the syntax that

you must use:

The “for... in” Statement

A “for... in” statement repeats the assigned statement/s over the properties of any object. JavaScript will execute the assigned statement/s for every

```
for (variable in object) {  
    statements  
}
```

property. The syntax of a “for... in” statement is:

The “continue” Statement

You may use this statement to restart other statements (e.g. for, while, do... while, and label). Here are the two rules that you must remember when using a “continue” statement:

- If you won't include a label in your continue statement, it will terminate the current process of the innermost loop and continue the next one. Unlike a break statement, a continue statement cannot terminate an entire loop. If used on “while” loops, the control flow will return to the assigned condition. If used on a “for” loop, on the other hand, the control flow will go back to the increment expression.
- If you'll include a label in your continue statement, it will only affect the statement linked to that label.

1. `continue;`

2. `continue label;`

The syntax of a continue statement has two

forms, which are:

Syntax

As mentioned previously, JavaScript needs to be embedded in the HTML document or has to be referenced from the same. In order to embed the script, the script needs to be placed with the script tag, which looks like –

```
<script>...</script>
```

The script enclosed within these tags must be placed inside the HTML document. Although, this script can appear anywhere inside the HTML document, its placement inside the `<head>...</head>` is recommended. All the JavaScript code written within the `<script>...</script>` block is interpreted by the browser. Two attributes can be provided along with the script tag. These attributes are –

Type

The type attribute is used to specify the scripting language that is being used. In this case, type must be set to the value “text/javascript”.

Language

This attribute has now been phased out. However, earlier versions of HTML still use it. Its value must typically be set to “javascript”. The JavaScript part of the code will now look like –

```
<script language="javascript" type="text/javascript">
```

```
...
```

```
</script>
```

Writing Your First JavaScript

Assuming that you have some experience of coding in HTML, let us work through our first example. This piece of code prints 'Hello World!' As you can see, the code is embedded into an HTML document. The output for the code is shown in the image that appears below it.

```
<html>
<body>
<script language="javascript" type="text/javascript">
<!--
document.write("Hello World..")
//-->
</script>
</body>
```



```
</html>
```

The JavaScript code is enclosed within the `<script></script>` tag. This script is embedded inside the HTML file by placing it inside the `<body></body>` of the `<html></html>` tags. As you can see, the code is also enclosed inside HTML comments. The reason for doing this is that it prevents browsers that do not understand JavaScript to use this code unlawfully.

As part of the main code, `document.write` function is called, which prints the string given to it as parameter on the screen. This function can be used to write HTML as well. In order to execute this code, open a text editor and copy this code to the file. Save the file as `index.htm` and open it in a web browser that supports JavaScript. The result of this can be shown in the image below.

Line Breaks and Whitespaces

You are free to use whitespaces, tabs and newlines in your code to make it readable and well organized. These characters are completely ignored by JavaScript.

Optional Use of Semicolons

Most conventional programming languages like C, Java and C++ require you to place semicolon at the end of each statement to indicate the termination of an statement. However, in Java, if you have written code in such a manner that each statement is on a different line, then you can omit the use of semicolons. Example of this concept is the code shown below –

```
<script language="javascript" type="text/javascript">
<!--
-
variab1
=
15
variab2
=
35
//-->
</script>
```

However, if all the statements are written in continuity, then you must use semicolons to tell JavaScript where a statement ends and a new one starts. Sample code to demonstrate the use of semicolon is given below –

```
<script language="javascript" type="text/javascript">
<!--
variab1
=
15;
variab2
=
35;
//-
-
>
</script>
```

With this said, it is important to mention that the use of semicolon is

recommended as it is considered a good programming practice.

Case Sensitivity

JavaScript is sensitive to capitalization or case of the letters being used. For instance, I and i are two different identifiers for JavaScript. This holds true for any keyword, function name or variable name that you use. Anything you write in JavaScript must have a consistent case for it to be identifiable by JavaScript.

Writing Comments

JavaScript identifies both C and C++ style of comments. The single line comments can be made by writing `//` at the beginning of the line. In this scenario, anything that is written between the `//` characters and end of the line are treated as comments and ignored by JavaScript. The other type of comments supported by JavaScript is the multi-line comment. Such a comment begins with `/*` and ends with `*/`. JavaScript ignores any character and number of lines that lie between these two character sequences.

Besides the above-mentioned, JavaScript also supports HTML style of commenting. However, it only recognizes `<!--` and treats the characters following this character phrase till the end of the line as a single line comment. JavaScript does not recognize the HTML comment closing character sequence, `-->`. Therefore, as and when you require the use of this character phrase, you must precede it with `//`. In order to help you understand this in a better way, sample code has been given below.

```
<script language="javascript" type="text/javascript">
```

```
<!--
```

```
// Single line comment.
```

```
/* Multi-line comment
```

```
Similar to the comments in C and C++ */
```

```
//-->
```

```
</script>
```

3 Enabling JavaScript in The Browsers

Most of the browsers that are being used today support JavaScript. However, you may be required to disable or enable this feature of your browser. You may perform this operation manually. This chapter gives you a quick insight into how manual enabling and disabling of JavaScript can be performed for browsers like Firefox, Google Chrome and Internet Explorer.

Internet Explorer (IE)

In order to enable or disable JavaScript on IE, you need to follow the steps given below –

In the IE panel, go to Tools and from the menu, open Internet Options. Look for Security tab and select the same. Choose the button named Custom Level. Now scroll down and you will see an option of the name, Scripting. Under this heading, select the radio button that says Active Scripting. Click ok and you will be back to the browser.

If you wish to disable JavaScript, you need to follow the same set of steps and at the point where you choose Active Scripting you must choose the Disable button.

Firefox

In order to enable or disable JavaScript on Firefox, you need to follow the steps given below –

Open a new window and in the address bar of the tab, type – `about: config`. This will bring a dialog box in front of you, which shall give you a warning. Choose the option that says ‘I’ll be careful, I promise!’ Once you choose this option, the browser will present you with a list of configure options. Type `javascript.enabled` - in the search bar. If you right click on this value, you will see an option that says select toggle. Choosing this option will simply toggle the value. If JavaScript is enabled, then choosing this option will disable it else it will be enabled.

Chrome

In order to enable or disable JavaScript on Google Chrome, you need to follow the steps given below –

Enter the Chrome menu by clicking on a symbol on the top right hand corner of the window. From the available options, choose Setting and click on ‘Show Advanced Setting’. The window that will option after this will have a section named Privacy. Look for the button that says Content settings. You will be able to locate a JavaScript section under this button. To enable JavaScript, select Allow all sites to run JavaScript (recommended)’ else select ‘Do not allow any site to run JavaScript’.

Programming in JavaScript for Non-JavaScript Browsers

If you are writing a JavaScript and the same is run by a user on a nonJavaScript browser, then you can print a message for the user with the help of the `<noscript>` tag. This concept can be implemented in the following manner.

```
<html>
<body>
<script language="javascript" type="text/javascript">
<!--
document.write("Hello
World...")
//-->
</script>
<noscript>
Warning: JavaScript Required!
</noscript>
</body>
</html>
```

This message ‘Warning: JavaScript Required!’ will be displayed to the user if the user attempts to run the script on a non-JavaScript browser.

Placement of JavaScript in Files

Although, the user is free to include the JavaScript code anywhere inside the HTML document as long as the code is enclosed inside the `<script></script>` tags, there are some preferred ways of its inclusions. These practices are described below.

Including JavaScript Code in `<HEAD></HEAD>` Section

Typically, if you wish to trap user clicks, code related to the same is included in the head section of the HTML document. In order to help you understand this concept, a sample code is given below.

```
<html>
<head>
<script type="text/javascript">
<!--
-
function
pHWorld()
{
alert("HelloWorld!")
}
//-->
</script>
</head>
<body>
<input type="button" onclick=" pHWorld ()" value="Print HelloWorld!"
/>
</body>
</html>
```

Including JavaScript Code in `<BODY></BODY>` Section

If there is some JavaScript functionality that you want to run after the page has loaded, then this script should appear inside the body of the HTML document. The code given below shows how JavaScript code can be embedded into the body of an HTML document.

```
<html>
<head>
</head>
<body>
<script type="text/javascript">
<!--
document.write("HelloWorld!")
//-->
</script>
<p>Body of HTML document</p>
</body>
</html>
```

Including JavaScript Code in <HEAD></HEAD> and <BODY></BODY> Sections

You can have different JavaScript codes embedded in the head and body sections of the same HTML document. Sample code to illustrate how this can be done is given below.

```
<html>
<head>
<script type="text/javascript">
<!--
-
function
printHW()
{
alert("HelloWorld!")
}
//-->
</script>
</head>
<body>
```

```
<script type="text/javascript">
<!--
document.write("HelloWorld!")
//-->
</script>
<input type="button" onclick="printHW()" value="Print Hello World"
/>
</body>
</html>
```

Writing JavaScript into External Files

The more pages you have in your website, the more you will realize that you are writing the same JavaScript code in all these different pages or at least some of these pages. JavaScript allows you to manage identical code used in different HTML documents by writing the JavaScript code in external files, which can then be linked with the page to call the required JavaScript functionality. You can link external files by using the src attribute of the script tag. Sample code to illustrate this functionality has been given below.

```
<html>
<head>
<script type="text/javascript" src="sample.js" ></script>
</head>
<body>
</body>
</html>
```

Here, sample.js is the JavaScript file containing the script. The code within this file can be used by the HTML document just like you used the JavaScript written within the script tag. Please note that the JavaScript code needs to be written into any text file and must be saved with the extension .js. For instance, your sample.js can look like this –

```
function printHW() {
alert("HelloWorld!")
```

Popup Message

JavaScript has numerous built-in methods that can help in displaying popup messages for various purposes. Let us discuss the various popup boxes provided by JavaScript:

Alert Box

This is used to show a message to a user, particular where the emphasis is needed. It comes with an OK button that closes the popup when clicked. It is created by calling the JavaScript's alert() function.

For example:

```
<html>
<body>
  <h1>Popup</h1> <script> alert("Hello
    World!"); // to display a string message
    alert(12); // to display a number alert(true); //
    to display a boolean
  </script>
</body>
</html>
```

When you run the code, it will display the first alert box with the string Hello World! Click OK button and it will display an alert with 12. Again, click the OK button and it will display an alert box with true. This shows that the alert box can be used to display a message of any type.

Confirm Box

Sometimes, a user is expected to give a confirmation so as to proceed. For example, you may want a user to confirm deletion or update of certain details before the process can be completed. This can be done using the `confirm()` function provided by JavaScript. This function will display a popup box to the user with two buttons namely OK and Cancel. The next step is determined by the button that the user clicks.

For example:

```
<html>
<body>
  <h1>confirm()</h1>
  <p id="del"></p>
  <script>
    var
    userChoice;
    if (confirm("Do you really want to delete the data?") == true) {
      userChoice = "Data deleted successfully!";
    } else { userChoice =
      "Delete Canceled!";
    } document.getElementById("del").innerHTML =
    userChoice; </script>
</body>
</html>
```

Prompt Box

In some cases, you may need to receive an input from users and use it to perform further actions on the web page. For example, you may need to calculate the monthly amount for loan repayment based on the number of months within which the user wants to settle the loan. In such a case, you can use the `prompt()` function provided by JavaScript.

The function should take two string parameters. The first one is the message that will be displayed while the second one is the default value to be in the input text once the message has been displayed:

`prompt([string display_message], [string default_Value]);` For example:

```
<html>
<body>
    <h1>prompt()</h1>
    <p id="pro"></p>
    <script> var age = prompt("What is your
        age?", "26");
        document.getElementById("pro").innerHTML = "You are " + age + "
years old";
    </script>
</body>
</html>
```

The code prompts you to enter your age. If you don't and click the OK button, 26 will be used as your default age.

JavaScript Variables

A variable is a name for a storage location. From its name, it can vary. Variables hold values that can vary. In JavaScript, we use the `var` keyword to declare variables. Each variable must be given a unique name. You are allowed to assign a value to a variable during the time of its declaration. This can be done using the equals to symbol (`=`).

The following is the syntax for declaring and initializing variables in JavaScript:

```
var variable-name;  
var variable-name  
= value; Here are  
valid examples of  
this:
```

```
var x = 1; // variable storing a numeric  
value  
var y = 'nicholas'; // variable storing  
a string value  
var z; // declare a variable  
and not assign a value
```

We have used the `var` keyword to declare three variables. Two of the variables have been assigned a value. Note that this has been done at the time of their declaration.

We can also declare more than one variable in a single line and separate them by a comma (`,`). This is demonstrated below:

```
var x = 1, y = 'nicholas', z;
```

Note that in JavaScript, one can declare a variable without using the `var` keyword. However, if you do this, ensure that you assign a value to the variable immediately. This is demonstrated below:

```
x  
=  
1;  
y  
=  
'nicholas';
```

However, I don't recommend that you declare any variable without the use of

var keyword. This is because you may end up overriding a global variable without knowing.

If a variable is declared without the var keyword, it automatically becomes a global variable, regardless of where it has been declared.

Here are the rules that govern the declaration of variable names in JavaScript:

1. The variable name should begin with an underscore (_), a letter (either A to Z or a to z), or the dollar sign (\$).
2. After the first letter, one can use digits in the variable name.
3. JavaScript variable names are case sensitive, example, p is different from P.

The following example demonstrates how to use variables in JavaScript:

```
<script>
var
a =
12;
var
b
=
11;
var
c=
a +
b;
document.write(c);
</script>
```

The code will return the sum of the two variable values, which is 23.

Local Variables

These are variables that are defined within a block or a function. It can only be accessed from within the block or function within which it has been defined.

Here is an example:

```
<script>
function
xyz(){ var
a=12; // a
local
variable
}
</script>
```

The variable a can only be accessed from within the function xyz().

Here is another example of local variable declaration:

```
<script> If(5 < 10){ var a
= 12; //a JavaScript local
variable }
</script>
```

The variable a in the above example can only be accessed from within the block of its declaration.

Global Variables

These are variables that can be accessed from any function. A global variable is usually declared outside a function or with the JavaScript's window object.

For example:

```
<script> var  
val=12; //a  
global variable  
function x(){  
document.writeln(val);  
}  
function  
y(){  
document.writeln(val);  
} x(); //calling a  
JavaScript function  
y(); //calling a  
JavaScript function  
</script>
```

12 12

The code will return the
following:

The code demonstrates that the function can be referenced from different functions and its value will remain the same.

If you need to declare a global variable within a function, you must use the window object. For example:

```
window.val=12;
```

Such a variable can be declared within any function and accessed from any function.

For example:

```
<script> function xy(){ window.val=12; //declaring a global
```

variable using window object

```
} function xz(){ alert(window.val); //accessing a global variable  
from another function  
}  
xy();  
xz();  
</script>
```

We have declared the variable `val` within the function `xy()` and assigned it a value of 12. We have then accessed the same variable from the function `xz()`. This returns a value of 12.

After declaring a variable outside a function, it is internally added to the window object. The object can at the same time be accessed via the window object.

Here is an example:

```
<script> var val = 12; function xy(){  
alert(window.val); //accessing a global  
variable  
}  
</script>
```

Each programming language supports a number of data types. They represent the types of values which can be represented then manipulated in the programming language. JavaScript supports 3 primitive data types including:

1. Numbers such as 1, 8, 90, 345 etc.
2. Strings, which are texts, for example, "My name" etc.
3. Boolean, which can be true or false?

JavaScript is also capable of supporting 2 trivial data types, namely null and undefined. It also has a composite data type called object.

Note that in JavaScript, there is no difference between integers and floats. In JavaScript, all numbers are represented in the form of floating-point values.

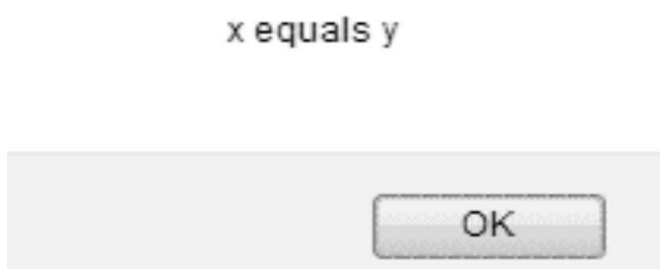
Comparing variables

In JavaScript, variables can be compared in different ways. We can compare whether two variables are equal or not by use of double equals sign.

Example:

```
<script>
x=30;
y=30;
  if (x==y) {
    alert("x equals y")
  };
</script>
```

When executed, the script will show a pop up with the text “x equals y” as shown below:

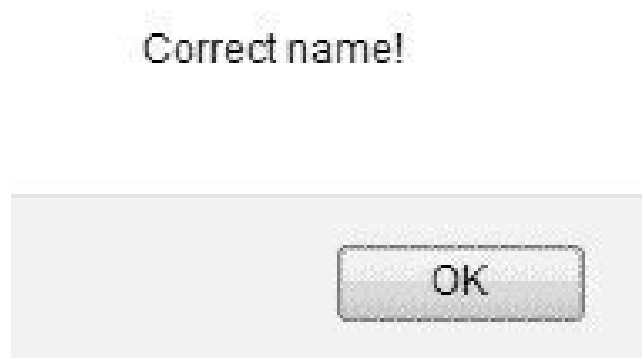


This is the values of the two values are equal, that is, 10. If not, nothing will happen once the script is executed.

Other than numbers, we can also compare strings as shown below:

```
<script>
firstname="Nicholas";
lastname="Samuel";
  if (lastname=="Samuel") {alert("Correct name!")};
</script>
```


The comparison will evaluate to a true, so you will get a popup with the text “Correct name!” as shown below:



Note that when doing a comparison, we use double equals sign (==). If you instead use a single equal' sign, the value on the right side of the sign will become the value of value (variable) on left side. An example is given below:

```
if (lastname="Samuel") {alert("Correct name!");}
```

JavaScript Operators

JavaScript supports the use of operators. An operator performs an operation on one or many operands to produce a result. For example:

2 + 3

In the above example, 2 and 3 are the operands while + is the operator. 2 is the left operand while 3 is the right operand. The + operator will add the two to give a result of 5. Let us discuss the various types of operators supported in JavaScript.

Logical Operators

These types of operators help us in connecting either one or more conditions. The following are the logical operators supported in JavaScript:

- &&- the AND operator. It checks whether the two operands are non-zero. If yes, it returns a 1, otherwise, it returns a 0.
- ||- the OR operator. It checks whether any of the operands is non-zero. If yes, it returns a 1, otherwise a 0.
- !- the NOT operator. For reversing the Boolean result of an operand or condition.

For example:

```
<html>
<body>
  <h1>Logical Operators</h1>
  <p id="pg1"></p>
  <p id="pg2"></p>
  <p id="pg3"></p>
  <p id="pg4"></p>
  <p id="pg5"></p>
  <script> var x = 12, y = 5;
  document.getElementById("pg1").innerHTML = (x != y) && (x <
  y); document.getElementById("pg2").innerHTML = (x > y) || (x ==
  y); document.getElementById("pg3").innerHTML = (x < y) || (x ==
  y); document.getElementById("pg4").innerHTML = !(x < y);
  document.getElementById("pg5").innerHTML = !(x > y); </script>
</body>
</html>
```

The code prints the following result:

Logical Operators

false

true

false

true

false

In the above example, we have used the values of variables x and y to run various logical operators.

Arithmetic Operators

In JavaScript, you can perform arithmetic operations on your variables. These operations include addition (+), subtraction (-), multiplication (*), modulus (%), increment (++), decrement (--) etc. Example:

```
<script
type="text/javascript">
a=1; a=1; a++;
//a=2 a=1; a--;
//a=0 a=1; b=2;
c=a+b; //c=3
a=1; d=a+4; //d=5
First="Edwin";
Last="Peter";
Name=First+" "+Last;
//Name=Edwin Peter a=3*7;
//a=21 b=10/2; //b=5 c=(10/2)*2;
//c=10 d=10/(1*2); //d=5
</script>
```

The results from the various lines have been shown in the comment section, that is, after //. The two forward slashes, that is, //, denote the beginning of a comment in JavaScript.

Assignment Operators

These are operators that help us assign values to variables. They include the following:

- = - to assign the right operand value to the left operand.
- += - to sum up the left and right operand values then assign the obtained result to left operand.
- -= - to subtract the right operand value from the left operand value then assign the obtained result to left operand.
- *= - to multiply the left and right operand values then assign the obtained result to left operand.
- /= - to divide the left operand value by the right operand value then assign the obtained result to left operand.
- %= - to get the modulus of the left operand divided by the right operand then assign the resulting modulus to left operand.

Here is an example:

```
<html>
<body>
  <h1> Assignment Operators</h1>
  <p id="pg1"></p>
  <p id="pg2"></p>
  <p id="pg3"></p>
  <p id="pg4"></p>
  <p id="pg5"></p>
  <p id="pg6"></p>
  <script>
    var
    a =
    10,
    b =
    20;
```

```
    a =  
    b;  
document.getElementById("pg1").innerHTML  
    = a; a += 1;  
document.getElementById("pg2").innerHTML  
    = a; a -= 1;  
document.getElementById("pg3").innerHTML  
    = a; a *= 5;  
document.getElementById("pg4").innerHTML  
    = a; a /= 5;  
document.getElementById("pg5").innerHTML  
    = a; a %= 2;  
document.getElementById("pg6").innerHTML = a;  
</script>  
</body>  
</html>
```

The code will return the following result:

20

21

20

100

20

0

The expression `a = b;` changed the value of variable `a` from 10 to 20. That is why the first statement prints a 20. We then begin to work with a value of `a` being 20. The expression `a += 1;` increases the value of `a` by 1, so it becomes

21. This becomes the new value of variable a. The expression `a -= 1;` subtracts a 1 from the current value of variable a, which is 21, returning a 20. This becomes the new value of variable a. The expression `a *= 5;` multiplies the value of variable a, which is 20 by 5, returning 100. This becomes the new value of variable a. The expression `a /= 5;` divides the value of variable a by 5, returning 20. This becomes the new value of variable a. The expression `a %= 2;` returns the remainder after dividing the value of variable a by 2, returning a 0.

3 INTRODUCTION

When working with data, a key component is the ability to gather data from any device or environment and provide this data in a meaningful way to the data analyst. This chapter will review the process of creating a custom dashboard that provides a workspace for data to be collected and analyzed. This dashboard can be considered the primary portal for the analyst to review data and initiate new data collection processes.

This dashboard consists of a user login portal, allowing users to identify to the system and a register page that allows new users to become active. After a user has logged into the system, a core dashboard allows the user to select the analysis tools and visualizations that have been implemented. To aid the development process Bootstrap¹ is used as the core library during the layout of visual components.

4.2 USER LOGIN

Example 1: User login page

This example consists of six files:

1. create_dashboard.sql — This file is used to create the database table that is used during the login and registration.
2. bootstrap.js — The core Bootstrap JavaScript file. Available from their website^[1].
3. bootstrap.min.css — The core Bootstrap CSS file. Available on the Bootstrap website.
4. index.php — This file is the main user interface for the login process. Bootstrap is used to theme this page.
5. login.php — This page is dynamically called using jQuery. The username and password hash are passed to this page and verified against the credentials stored in the database.

Example code: create dashboard.sql

To begin the login process, a database table that contains the accounts is created titled *dashboard_accounts*, as shown in Sample Code 1. This table contains an *ID* column that is an auto-incremented primary key. The two core columns in this table are the *username* and the *password* columns. When working with passwords, it is always important to remember that the password should never be stored in the database in plain text. In the case where unauthorized access to the database is made, the same username and password combinations could be used on other systems. It is a common practice to store the password in a *hash* based format. Hashing is the process of passing a password through a hashing algorithm that will return a fixed-length string of symbols and letters. However, it is possible to take the hash and work backward to retrieve the password if no additional security measures are implemented, such as adding *salt*.

```
1      CREATE TABLE `dashboard_accounts` (  
2          `id` INT NOT NULL AUTO_INCREMENT,  
3          `username` VARCHAR(100),  
4          `password` VARCHAR(100), 5          `email` VARCHAR(100),
```



```

6          PRIMARY KEY (`id`)
7      );

```

SAMPLE CODE 1: Main *dashboard* *accounts* table for the system.

Example code: [index.php](#)

The user login page is the first page seen by a user before they can access the dashboard. In this section, the complete login page is described. The overall layout for this page is based on Bootstrap. Sample Code 2 begins with a reference to the jQuery^[2] library on line 4. This is included as it will be used to gather the username and password that have been entered by the user for validation later. Popper^[3] is included on line 5. This is a tooltip positioning engine for pop-overs and drop-down menus available from the Bootstrap library. To utilize this functionality, the core Bootstrap library is included on line 8.

The style for the login page is kept very minimal. On lines 10–16, the input fields are styled to stretch to the width of the available space. Padding is added to move the text from the left-hand side of the input field. The margin is set to *8px* and the *display* is set to *inline-block*. A small *1px* border is added to the input fields that is solid light-gray. On lines 17–25, the style for the main login button is added. The color is set to *white* and the *padding* and *margin* is set. As this is a button, when the user places the mouse pointer over the button, the cursor will change to a *pointer*. On line 27, when a mouse is placed over the button, the opacity of the button will change to 0.8 from the initial 1.0.

On line 30, the *viewport* is added as the main visible area for the user. This helps scale the content the user will see with the initial scale set to 1.

The main body of HTML contains a *div* tag with the *class* set to *container* on line 34. Inside this the *nav* bar is added on line 35. This *nav* bar has the *bg-light* colour with rounded edges. More of these colors are available on the Bootstrap website.^[4] On line 36, a single link is added with the *class* set to *navbar-brand*. This is the main focus for the title. The *href* points to the *index.php* page with the

visible text set to *Feedback Dashboard*.

On line 38, the *main* tag is added with the role set to *main*. Inside this main, a single *div* tag is added with the class set to *jumbotron* on line 39. This provides a space for the main content to be displayed. Inside the jumbotron, a *div* tag is added with the class set to *mx-auto* on line 40, allowing us to center the component.

On line 42, a simple piece of text is added to instruct the user to enter their username and password. A form tag is wrapped around two input fields with the *ID* set to *login* on line 43. This is followed by two HTML *input* fields on lines 45 and 47 that were styled earlier using CSS. This *ID* will be used later to attach a listener for the form submission process. A single *span* tag is added with the *ID* set to *message* on line 49 that is used as a default display location for error messages that can occur when the form values are incorrect, e.g., the wrong username or password.

```
1      <head>
2      <link rel="stylesheet" href="Bootstrap.min.css" crossorigin=
3      "anonymous">
4      <script src="jquery.min.js"></script>
5      <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/
6      1.12.3/umd/popper.min.js"
7      crossorigin="anonymous"></script>
8      <script src="Bootstrap.js" crossorigin="anonymous"></script>
9      <style>
10     input[type=text], input[type=password] {
11     width: 100%;
12     padding: 12px 20px;
13     margin: 8px 0;
14     display: inline-block;
15     border: 1px solid #ccc;
16     }
17     button {
18     background-color: grey;
19     color: white;
20     padding: 14px 20px;
21     margin: 8px 0;
22     border: none;
23     cursor: pointer;
24     width: 100%;
25     }
26     button:hover {
27     opacity: 0.8;
```

```

28         }
29     </style>
30     <meta name="viewport" content="width=device-width,
31     initial-scale=1, shrink-to-fit=no">
32     </head>
33     <body>
34     <div class="container">
35         <nav class="navbar navbar-expand-lg navbar-light bg-light rounded"> 36 <a class="navbar-brand"
href="index.php">Feedback Dashboard</a>
37     </nav>
38     <main role="main">
39     <div class="jumbotron">
40     <div class="mx-auto" style="width:300px">
41     <h1>Login</h1>
42     <p> Please enter your username and password to login.</p>
43     <form id="login">
44     <label><b>Username</b></label>
45     <input type="text" id="username"/>
46     <label><b>Password</b></label>
47     <input type="password" id="password"/>
48     <button type="submit">Login</button> 49     <span id="message"></span>
50     </form>
51
52     </div>
53     </div>
54     </main>
55     </div>
56     </body>

```

SAMPLE CODE 2: index.html — Login page for the dashboard created using Bootstrap.

In Sample Code 3, the core functionality of the login page is performed using JavaScript. The first component is a form *submit* listener that is appended to the *login* form on line 58. Instead of a traditional form *POST* that requires the page to be refreshed, the form *POST* is performed using JavaScript. The *preventDefault()* function is called on line 59, during the form submission to prevent any redirection of the page. After this has been performed the jQuery selector is used to select the *username* on line 60 and *password* on line 61. The *val()* function is used to extract the values from the fields. These values are placed into two local variables, *un* and *pw*.

To validate the details that have been entered are correct, a call is made to the jQuery *POST* function to send the *username* and *password* variables to a second PHP page titled *login.php* on line 62.

The *type* of request being sent to this page is described as a *login*. This allows the single PHP page to be re-purposed for other tasks in the future. Once the call has been made to the PHP page the *data* variable is returned on line 64. The variable will contain a flag titled *success* if the login credentials were correct. The location of the web browser is redirected to the *dashboard.php* page on line 67.

In the case where the username or password was incorrect, on line 70, the message *span* tag is used to hold the error message. This is done by passing a string to the *text()* function that can be called on the *span* tag. To aid debugging, two *console.log()* functions are added on lines 68 and 72.

```
57         <script>
58         $( "#login" ).submit(function( event ) {
59             event.preventDefault();
60         var un = $("#username").val(); 61         var pw = $("#password").val();
62             $.post( "login.php", { type:"login", username: un, password:
63                 pw })
64             .done(function( data ) {
65                 alert(data);
66                 if(data == 'success'){
67                     $(location).attr("href", "dashboard.php");
68                     console.log('success');
69                 } else {
70                     $("#message").text('Login failed, please check
71                         your username and password.');
```

SAMPLE CODE 3: index.html — JavaScript form listener with redirection to the correct page on successful login.

Output

shows the completed dashboard login page.

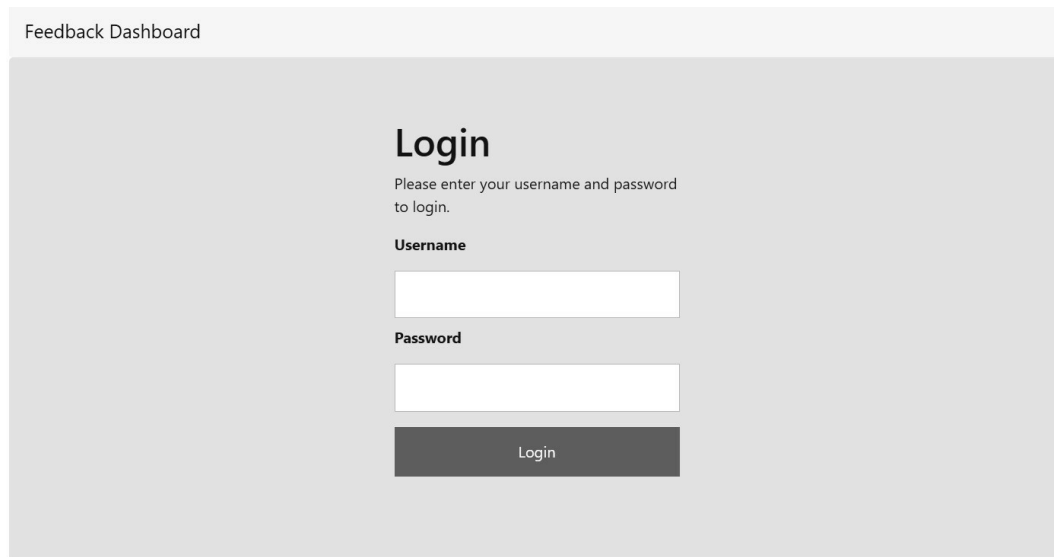


FIGURE 4.1: Custom dashboard for the login page.

Example code: **login.php**

To validate the login details, the *login.php* page is used. When the JavaScript form submission is performed on the login page, the *username* and *password* are sent across via a *POST*. In Sample Code 4, before beginning the login process, the *type* parameter is first checked on line 3 to see what type of request was being made. When the *type* is set to *login*, the *login()* function is called.

Inside the *login()* function on lines 7 and 8, the *username* and *password* are first extracted from the *POST* and placed into local variables. A simple connection is made to the local database to get the record for the *username* that was provided. In the SQL, a *LIMIT 1* clause is appended to only return a single record from the database. This is shown on line 16, where the *username* is left as a question mark, on line 19 the *username* is added using the *bindParam()* function. The statement is executed on line 20.

In the returned record, the *password* is hashed using the *bcrypt* algorithm.^[5] During the registration process of the user, the password that is gathered from the user is passed to the password hashing function that will return a 60 character hash string that is stored in the database. An example of a string for the password

MySamplePassword would be

\$2y\$10\$tLUrTdYnlSI3B/dsYBtga.1JtDXxEKn.Dyqux5LmRvjwHCSArNpEO. *Salt* can be added, which is a small additional piece of text added to the user's password during registration. This salt will prevent a direct match ever being found between two passwords on different systems as the salt is different. The hash generated, therefore, will not be the same each time.

As we are using a hashing algorithm on the password, a direct comparison can not be performed between the password hash and a hash that is stored in the database. A call must be made to the *password verify()* function on line 22 that takes in two parameters. The first is the raw password entered by the user, and the second is the hash that is stored in the record returned from the database at position 2 of the returned row.

When the login is successful, a session is started on line 25, and the *ID* from the user's record is stored in the session variable *ID* on line 26, followed by the username in the *username* variable on line 27. When the login is successful, the word *success* is returned back to the JavaScript function that performed the *POST*. This triggers the original JavaScript that called the *login.php* page to redirect to the main *dashboard.php* page. In the case where the password is not valid, the *else* statement is triggered, and the text *Password is not valid* is returned to the user.

```
1      <?php
2      $type = $_POST['type'];
3      if($type == 'login'){
4          login();
5      }
6      function login(){
7          $username = $_POST['username']; 8          $password = $_POST['password'];
9          try {
10             $host = 'localhost';
11             $dbname = 'bookexamples';
12             $user = 'sampleuser';
13             $pass = '3C9038A509BE6145AAD6827B4568AD918BC3DAC0';
14             $DBH = new PDO("mysql:host=$host;dbname=$dbname", $user,
15                             $pass);
16             $sql = "SELECT * FROM dashboard_accounts WHERE username
17                     =? LIMIT 1";
18             $sth = $DBH->prepare($sql);
19             $sth->bindParam(1, $username, PDO::PARAM_STR);
```

```

20         $sth->execute();
21         $res = $sth->fetch();
22         $result = password_verify ($password,$res[2]);
23         if($result){
24             echo 'success';
25             session_start();
26             $_SESSION['id'] = $res[0];
27             $_SESSION['username'] = $res[1];
28             $_SESSION['valid'] = True;
29         } else {
30             echo 'Password is not valid';
31         }
32         } catch(PDOException $e) {echo $e;}
33     }
34     ?>

```

SAMPLE CODE 4: login.php — Accepts the username and password entered by the user for validation.

USER REGISTRATION

In this section, we will review the process of creating a new user account in the system. The interface for this is similar to that of the login process.

Example 2: User registration page

First, a new file titled *register.php* is created. This file will provide the user with input fields to enter their username, email, and password.

This example contains two files and three libraries:

1. bootstrap.js — The core Bootstrap JavaScript file. Available from their website^[6].
2. bootstrap.min.css — The core Bootstrap CSS file. Available on the Bootstrap website.
3. jquery.min.js — The minified version of the jQuery library. Available on their website^[7].
4. register.php — This is the main registration page for the dashboard.
5. ajax.php — This is used to receive the registration details via jQuery *POST* and save them to the database.

Example code: register.php

Sample Code 5 shows the HTML for the registration page titled *register.php*. This page will contain the Bootstrap HTML to layout the page along with the JavaScript to trigger the registration process.

At the top of this file on line 2, a reference to the Bootstrap CSS file is included. This is followed on line 3 by a reference to the jQuery library. On line 5, a reference to *Popper.js* is included as it is a dependency for Bootstrap. On line 9, the core Bootstrap JavaScript file is included. On line 10, the *meta* tag is added as the main *view port* for the registration page.

The main style for the registration page is added. The input fields take the most styling, on lines 13–19, setting the width to be 100% of the available space. Padding and a margin are added to align the text horizontally across the input field. The method of display is set to *inlineblock* on line 17. A simple border is added of *1px* with a light gray color on line 18. The registration button also has a small amount of styling. The main change visible is the cursor on line 26. This is changed to a pointer when the mouse is placed over the register button. The opacity of the button is also changed on line 30, by setting it to 0.8 from the initial 1.0 when a mouse hovers over it. On line 35, a *div* tag is added with the *class* set to *container*. Inside of this, a *nav* bar tag is added with the theme set to *bg-light* with rounded *edges* on line 36.

After the navigation bar has been added, the next step is to add the main content for the page to allow the user to enter their details. The *main* tag is added on line 40 as a wrapper around all the content. Inside of this, on line 41, a *div* tag is added with the *class* set to *jumbotron*. This component allows for easy scaling. Inside of this, on line 42, another *div* tag is added with the *class* set to *mx-auto* that will automatically stretch horizontally. Inside of this, two simple pieces of text are added.

A *form* tag is added on line 45 as a unique method of identifying this form.

Four input fields are added to gather the *username*, *e-mail address* and *password* from the user on lines 46–53. A second input field is

added, allowing the user to verify their password. The IDs for the password are *password1* and *password2* for simple reference later during comparison. An important element is the *ID* attribute associated with each element, as these will be used later when processing the form with JavaScript. *Label* tags are added that allow a simple piece of text to be specified before each of the input fields. To allow the registration to begin, a *submit* button is added on line 54. A simple *span* tag is added titled *message* on line 55, which is used as a placeholder for error messages that may occur during validation. When the page first loads, this *span* tag will not have any value.

```
1      <head>
2      <link rel="stylesheet" href="Bootstrap.min.css" crossorigin=
3      "anonymous">
4      <script src="jquery.min.js"></script>
5      <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/
6      1.12.3/umd/popper.min.js"
7      crossorigin="anonymous">
8      </script>
9      <script src="Bootstrap.js" crossorigin="anonymous"></script>
10     <meta name="viewport" content="width=device-width,
11     initial-scale=1, shrink-to-fit=no">
12     <style>
13     input[type=text], input[type=password] {
14     width: 100%;
15     padding: 12px 20px;
16     margin: 8px 0;
17     display: inline-block;
18     border: 1px solid #ccc;
19     }
20     button {
21     background-color: grey;
22     color: white;
23     padding: 14px 20px;
24     margin: 8px 0;
25     border: none;
26     cursor: pointer;
27     width: 100%;
28     }
29     button:hover {
30     opacity: 0.8;
31     }
32     </style>
33     </head>
34     <body>
35     <div class="container">
```

```

36      <nav class="navbar navbar-expand-lg navbar-light bg-light
37          rounded">
38      <a class="navbar-brand" href="#">Feedback Dashboard</a>
39      </nav>
40      <main role="main">
41      <div class="jumbotron">
42      <div class="mx-auto" style="width:300px">
43      <h1>Register</h1>
44      <p> Register for a new account</p>
45      <form id="register">
46          <label><b>Username</b></label>
47          <input type="text" id="username"/>
48          <label><b>E-mail Address</b></label>
49          <input type="text" id="email"/>
50          <label><b>Password</b></label>
51          <input type="password" id="password1"/>
52          <label><b>Repeat Password</b></label>
53          <input type="password" id="password2"/>
54          <button type="submit">Register</button>
55          <span id="message"></span>
56      </form>
57      </div>
58  </div>
59      </main>
60      </div>

```

SAMPLE CODE 5: register.html — User registration page HTML part 1.

In Sample Code 6, a custom modal is shown. This modal is hidden when the application is launched and only becomes visible when a new user record has successfully been inserted in the database. On line 61, the unique *ID* provided to the modal is set to *nowRegistered*. On line 66, the title of the modal is set to *Success*. On lines 74-75, the main text visible to the user can be seen. On line 79, a *href* is added with a link to the *index.php* page that contained the original login page.

```

61      <div class="modal" id="nowRegistered" tabindex="-1" role="
62          "dialog">
63          <div class="modal-dialog" role="document">
64          <div class="modal-content">
65          <div class="modal-header">
66      <h5 class="modal-title" style="width:100%">Success</h5>
67          <button type="button" class="close" data-dismiss="modal"
68              aria-label="Close"
69              style="width:100px">
70          <span aria-hidden="true">&times;</span>
71          </button>
72          </div>
73          <div class="modal-body">

```

```

74         <p>You are now registered.</p>
75         <p>Click to login!</a></p>
76     </div>
77     <div class="modal-footer">
78
79         <a href="index.php" class="btn btn-primary"
80             role="button">Login!</a>
81     </div>
82 </div>
83 </div>
84 </div>

```

SAMPLE CODE 6: register.html — User registration page HTML part 2.

After the HTML components have been added, form validation is required. Sample Code 7 outlines this validation. To provide the best user experience when the form is submitted, the natural redirection process of the page will be suppressed. To do this, we will first use the jQuery selector on line 86 to select the *register* form and append a submit handler function. Whenever the form is submitted, this function will be called.

Inside this function, a variable is added titled *valid* on line 87. This is used as a flag to show if the form fields are valid or not before any further processing, e.g., sending the data to the database. The second component is the call to the *event.preventDefault()* on line 88. This will prevent the default redirection process of the page.

The jQuery selector is used to select the *username* field on line 90. The value is extracted using the *val()* function. On line 91, a check is performed to see if the *un* variable is blank. In the case where nothing was

typed into the *username* field, the input field would be blank, allowing the *if statement* to run. Inside this *if statement*, the jQuery selector is used again on line 109 to select the *message* span tag that was created earlier and append the message *Username blank* followed by a HTML *br* tag. This is added as the *message* span tag contains a long string that will be printed directly to the user's browser when an error occurs. By adding in the line break, each error message will be visible on a separate line.

In addition to these basic checks, the values of *password1* on line 102 and *password2* on line 103 are compared with each other and they must match. This is to prevent human error when entering passwords. On line 108, a simple *if statement* can be used with a *!=* between the two values identifying that if they are not equal to each other to show the error message.

```

85         <script>
86         $( "#register" ).submit(function( event ) {
87             var valid = true;
88             event.preventDefault();
89             // Username
90             var un = $("#username").val();
91             if(un == "" ){
92                 $('#message').append('Username blank.<br>');
93                 valid = false;
94             }
95             // Email
96             var email = $("#email").val();
97             if(email == "" ){
98                 $('#message').append('E-mail blank. <br>');
99                 valid = false;
100             }
101             // Password
102             var pw1 = $("#password1").val();
103             var pw2 = $("#password2").val();
104             if(pw1 == "" || pw2 == ""){
105                 $('#message').append('Password blank.<br>');
106                 valid = false;
107             }
108             if(pw1 != pw2){
109                 $('#message').append('Sorry your passwords do not match.
110                 <br>');
111                 valid = false;
112             }
113             // If valid, then perform the post request
114             if(valid == true){

```

```

115             $.post( "ajax.php", { type:"register", username: un,
116             password: pw1, email: email })
117             .done(function( data ) {
118                 if(data == 'registered'){
119                     // Show modal
120                     $("#nowRegistered").modal();
121
122                 } else {
123                     $("#message").text('Sorry registration failed.');
```

SAMPLE CODE 7: register.html — JavaScript registration page validation part 3.

At any stage during the registration process, if any of the fields are not correct the *valid* variable will be set to *false*. This is used to keep track of the form fields and see if the submission process can continue on to the next section. Before we perform the *POST* operation, an *if statement* on line 114 is used to ensure that the *valid* variable is equal to *true*. After this, on line 115, a jQuery *POST* is performed to the *ajax.php* page. The *username*, *password*, and *email* are sent to the second page. On successful *INSERT* on the second page, the Bootstrap modal will be shown by calling the *modal()* function on line 120, otherwise the message *span* tag will be used again to show this error message on line 123.

Example code: [ajax.php](#)

In Sample Code 8, the page titled *ajax.php* that is responsible for inserting new records in the database can be seen. At the top of this page on line 2 the *type* variable is received. On line 3, if the *type* variable is equal to *register* then a call is made to the custom *registerNewUser()* function.

Inside the *registerNewUser()* function on line 7, the three variables that were sent across via the *POST* are set into local variables *username*, *password*, and *email* on lines 7–9. The *password* variable is passed into

the PHP *password hash()* function on line 10. This function takes two parameters. The first one is the password the user has provided, and the second is the type of hash that we would like to produce. In this example, we are working with the *bcrypt* algorithm. It is important to remember that the user's password should never be stored in plain-text in a database.

On line 18, a simple SQL *INSERT* statement is built up to insert into the database that contains the table titled *dashboard_accounts*. The three columns we are inserting into are specified, followed by the *VALUES* keyword and three question marks. These three question marks are used as unnamed placeholders. The values are not put directly into the SQL as this is unsafe. The three values are bound onto the statement using the *bindParam()* function on lines 22–24. The number at the start of each denotes the position to insert. This is followed by the value we want to add, and the data type that is roughly associated with it. As the *username*, *password*, and *email* are all strings, *PARAM_STR* is used as the data type for each.

```

1  <?php
2  $type = $_POST['type']; 3 if($type == 'register'){ 4      registerNewUser();
5
6
7
8
9
10
11
12
13
14
15
16
    }
    function registerNewUser(){
        $username = $_POST['username'];
        $password = $_POST['password'];
        $email = $_POST['email'];
        $hashed_password =
        password_hash($password,
        PASSWORD_BCRYPT);
        try {
            $host = 'localhost:3306';
            $dbname = 'bookexamples';
            $user = 'sampleuser';
            $pass =
            '3C9038A509BE6145AAD6827B4';
            $DBH = new
            PDO("mysql:host=$host;dbname=$

```

```

17         $user,
18         $pass);
19
20         $sql = "INSERT INTO
21         `dashboard_accounts` (`username`,
22         `password`, `email`)
23         VALUES (?, ?, ?)";
24
25         $sth = $DBH->prepare($sql);
26
27         $sth->bindParam(1, $username,
28         PDO::PARAM_STR);
29
30         $sth->bindParam(2,
31         $hashed_password,
32         PDO::PARAM_STR);
33
34         $sth->bindParam(3, $email,
35         PDO::PARAM_STR);
36
37         $sth->execute();
38
39         echo 'registered';
40
41     } catch(PDOException $e) {echo
42     'failed' . $e;}
43
44 }
45
46 ?>

```

SAMPLE CODE 8: ajax.php — Register new user function.

When the *INSERT* runs successfully, the term *registered* will be returned to the browser. This result will be returned to the JavaScript caller on the *register.php* page triggering a Bootstrap modal to open. A *PDOException* is fired if any errors occur during the registration. The error message will be suppressed, and the word *failed* will be returned.

Output

shows the completed register page. describes the modal that is presented to the user once they have successfully registered. This modal provides the user with a link to bring them back to the login page.

Feedback Dashboard

Register

Register for a new account

Username

E-mail Address

Password

Repeat Password

Register

FIGURE 4.2: Completed dashboard register page.

Success

You are now registered.

Click to login!

Login!

Bootstrap modal shown when the registration was successful.

MULTI-COLUMN BAR CHART

In the previous examples, a focus was placed on creating the dashboard to allow for the registration and login of users. Once logged into the system, data that has been collected will be presented to the user in chart format. In these examples, Charts.js is used as the main clientside library for chart rendering.

Example 3: Multi-column bar chart

This example contains two files and two libraries:

1. `feedbackformrecords.sql` — This creates the database table for the sample data to be held for visualization.
2. `chart1.php` — This is the main visualization page.
3. `Chart.bundle.js` — Used to render the bar chart.
4. `utils.js` — A component of `Chart.js`

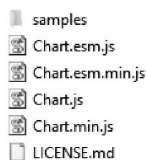
To begin, we first need to download a copy of `Chart.js` from the official repository.^[8] In the GitHub repository for the project, there are a number of variations to choose from. These are available as sometimes a developer may only want the source code while other times, they may

want the minified (or shrunk) version of the code for a deployed application in other times. In our case, we want the code and also the collection of examples to showcase the functionality. This will allow us to view the existing chart samples in context. For this example we download the *Chart.js.zip* file as shown in.

 Chart.esm.js	335 KB
 Chart.esm.min.js	157 KB
 Chart.js	386 KB
 Chart.js.zip	351 KB
 Chart.min.js	173 KB
 Source code (zip)	
 Source code (tar.gz)	

List of download options for Charts.js.

After the zip file has downloaded, unzip the contents of the zip file and place the entire Chart.js folder into the *htdocs* of the web server for convenience. When accessing the examples, this can be done by typing *http://localhost* into our web browser and then navigating into the *Chart.js* folder. Inside the Chart.js folder, there is a number of different JavaScript files with an additional folder titled *samples*, as shown in. The JavaScript files will be kept beside the charts that we will create to make the chart functionality available to us.



Contents of Chart.js zip file.

Inside of this *samples* folder as shown in, there is a file titled *index.html*. When this is opened through a web browser, as shown in complete running examples can be chosen. Unlike working on smaller pieces of code, when generating charts, there are many different elements that are required for the chart to function correctly. When creating a

custom chart, it is best practice to begin with a sam-

ple chart that utilizes sample data. This ensures minimal modifications need to be made for the sample chart to be used with alternative data or parameters.

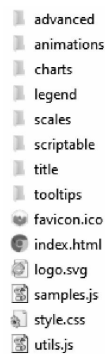


FIGURE 4.6: Contents of Chart.js folder.

 **Chart.js | Samples**
Simple yet flexible JavaScript charting for designers & developers

[Website](#) [Documentation](#) [GitHub](#)

Bar charts	Line charts	Area charts	Other charts
Vertical	Basic	Boundaries (line)	Bubble
Horizontal	Multi axis	Datasets (line)	Scatter
Multi axis	Stepped	Stacked (line)	Scatter - Multi axis
Stacked	Interpolation	Radar	Doughnut
Stacked groups	Line styles		Pie
Floating	Point styles		Multi Series Pie
	Point sizes		Polar area
			Radar
			Combo bar/line

index.html containing a list of working samples.

Example code: feedbackformrecords.sql

Before generating a chart, a new database table is set up. In Sample Code

9, a sample database table can be seen. The *FeedbackFormId* contains an integer used to identify what form the data has come from. The *FeedbackElementID* is a second attribute used to further narrow down exactly what component a user has used, e.g., input field or clicked a button. To record a user's rating between one and ten, an attribute titled *Score* is added. This will contain the numeric rating a user has given.

```
1      CREATE TABLE `feedbackformrecords` (  
2      `id` INT NOT NULL AUTO_INCREMENT,  
3      `FeedbackFormId` INT,  
4      `FeedbackElementID` INT, 5      `Score` INT,  
6      PRIMARY KEY (`id`)  
7      );
```

SAMPLE CODE 9: Sample feedback records with score attribute.

Example code: [chart1.php](#)

Sample Code 10 begins on line 5 with reference to the *Chart.bundle.js* file that contains the core JavaScript to render the graph. After this on line 6, a reference is made to the *utils.js* file that contains additional utilities for drawing the graph. This graph is a Chart.js demo with additional PHP to gather the data to be rendered.

On line 16 and 17 a *div* with the *ID* set to *container* is added and a canvas with the *ID* set to *canvas*. The *canvas* is where the drawing operations will take place, and the *container* is the single identifiable *div* tag for the entire bar chart. The width of the *container div* tag can be changed to suit the application needs. In the code below, the *container div* is taking up 50% of the page width.

Following this code, on line 20, the color helper is included. On line 21, the main object used to hold the data for the chart is created. Inside this on line 22, the labels attribute is created. This is an array of different names that will appear across the horizontal axis of the bar chart.

On line 24, the *datasets* array is created. This array contains the values that will appear for each label. This bar chart expects a commaseparated values inside the array. A sample data point array would be:

data [6, 40, 10, 11, 12, 1, 90, 9, 3, 10]. The first element inside of this array, 6, would be the value then associated with the *Score 1* label on the bar chart.

As hard-coding the values into the bar chart would not be dynamic, we will instead replace the values inside the *data[]* array with a call to a PHP function on line 31 that will dynamically call the database to return the values. This PHP function that takes a single parameter in the form of the *ID* of the records we are interested in plotting. In this example, 1 is passed.

```
1          <!doctype html>
2          <html>
3          <head>
4          <title>Bar Chart</title>
5          <script src="Chart.bundle.js"></script>
6          <script src="utils.js"></script>
7          <style>
8          canvas {
9          -moz-user-select: none;
10         -webkit-user-select: none;
11         -ms-user-select: none;
12         }
13         </style>
14         </head>
15         <body>
16         <div id="container" style="width: 50%;">
17         <canvas id="canvas"></canvas>
18         </div> 19 <script>
20             var color = Chart.helpers.color;
21             var barChartData = {
22                 labels: ["Score 1", "Score 2", "Score 3", "Score 4", "Score 5",
23                     "Score 6", "Score 7", "Score 8", "Score 9", "Score 10"],
24                 datasets: [{
25                     label: 'Customer Satisfaction',
26                     backgroundColor: color(window.chartColors.blue).alpha(0.5).
27                         rgbString(),
28                     borderColor: window.chartColors.blue,
29                     borderWidth: 1,
30                     data: [
31                         <?php getSatisfactionScores(1); ?>
32                     ]
33                 }]
34             };
35
36             window.onload = function() {
```

```

36                                     var ctx = document.getElementById("canvas").getContext("2d");

                                     window.myBar = new Chart(ctx, {
37                                         type: 'bar',
38                                         data: barChartData,
39                                         options: {
40                                             responsive: true,
41                                             legend: {
42                                                 position: 'top',
43                                             },
44                                             title: {
45                                                 display: true,
46                                                 text: 'Customer Satisfaction'
47                                             }
48                                         }
49                                     });
50
51
52     };
53 </script>
54 </body> 55 </html>
```

SAMPLE CODE 10: chart1.php — Simple Chart.js bar chart part 1.

Inside of the *getSatisfactionScores()* function shown in Sample Code 11, on line 57, we are taking in the *formID* and then passing this to the database query in the form of an unnamed placeholder. On line 65, the SELECT statement will gather the unique *score* labels, e.g., 1–10 followed by the *count* for each *score*. This is done by making a call to the SQL *count()* function and passing the *score* column.

A question mark is used to mark where the *formID* value will be placed. The value is passed to this position using the *bindParam()* function on line 69. Once the query has run, ten results will be returned. The *foreach()* statement on line 73 is used to loop over these results and print out the *score* label and the associated *count* number. This is done by running a *echo* statement on the current row at position 1 as shown on line 74.

```

56      <?php
57      function getSatisfactionScores($formID){
58          try {
59              $host = 'localhost:3306'; 60              $dbname = 'bookexamples'; 61              $user = 'sampleuser';
62                  $pass =
63                      '3C9038A509BE6145AAD6827B4568AD918BC3DAC0';
64                      $DBH = new
65                          PDO("mysql:host=$host;dbname=$dbname", $user,
66                              $pass);
67                          $sql = "SELECT score, count(score) FROM
68                              feedbackformrecords

```

```

67                                     WHERE FeedbackFormId = ? GROUP BY
                                     score";
68                                     $sth = $DBH->prepare($sql);
69                                     $sth->bindParam(1, $formID,
70                                     PDO::PARAM_INT);
71                                     $sth->execute();
72                                     $res = $sth->fetchAll();
73
74                                     foreach ($res as $row){
75                                     echo $row[1] . ' ';
76                                     } catch(PDOException $e) {echo $e;}
77                                     }
78                                     ?>

```

SAMPLE CODE 11: chart1.php — Simple Chart.js bar chart part 2.

Output

The final resulting bar chart can be seen in [Figure 4.8](#). As this bar chart is pulling data directly from the database, and not dependent on a static file, so whenever a user submits data into the database the result data will be visible when the chart is reloaded.

4.5BINARY BAR CHART

A simple form of data is binary feedback, whereby, a question is asked, and the answer is stored in the form of 1 for a *yes* and 0 for a *no*. Typically a user will be asked questions, and they will respond by ticking a box or clicking a radio button. Unlike the previous example that showed data on a range between one and ten, this data has only two variations. Although it may seem limited, it is the only way of accurately gathering information about specific questions such as:

- Do you earn over 50k?
- Are you married?
- Do you have kids?

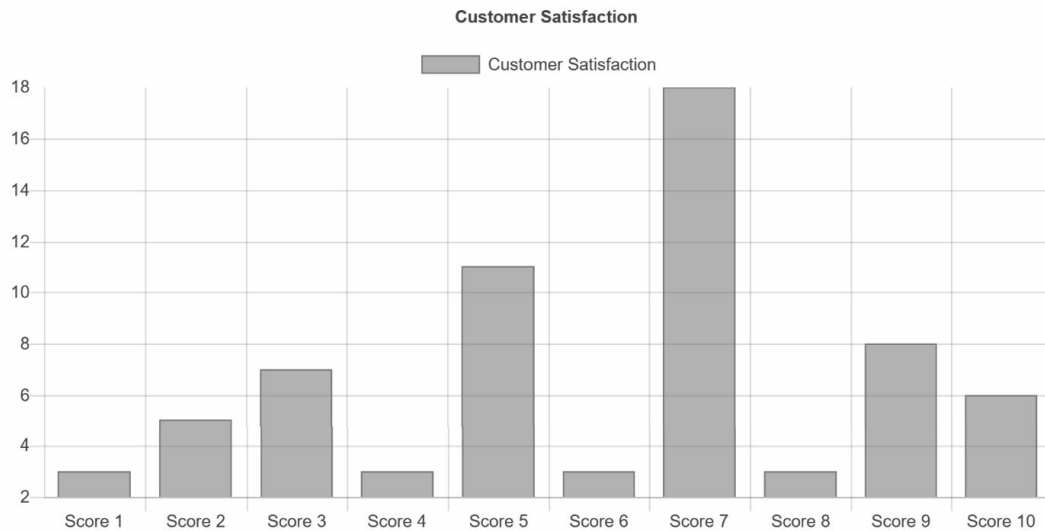


FIGURE 4.8: Bar chart showing the count of each score.

These questions can have a definitive *yes* or *no* answer attached to them. When graphing this type of data, both data collections for the *yes* and the *no* are shown. This helps derive quick insights from the data such as checking whether three-quarters of the people said *yes*, then there would likely be one quarter who said *no*.

Example 4: Binary bar chart

This example contains two files:

1. `survey.sql` — This file contains the SQL to create the `surveydatabase` table.
2. `chart2.php` — This file contains the main visualization.
3. `Chart.bundle.js` — This provides the ability to create the bar chart.
4. `utils.js` — An additional file that is used by `Chart.js`.

Example code: [survey.sql](#)

Sample Code 12 shows a sample database table titled *survey*. Like previous examples, an *ID* is added which is the primary key and also autoincremented. Every time a new survey has been completed, the data will be entered into this table. However, it is possible to have additional

descriptive data about who took the survey and when, it will be left out of this example.

The *FeedbackFormId* column references the feedback campaign attached to this data and four additional data columns: *married*, *haskids*, *ownscar*, and *makesover50k*.

```
1      CREATE TABLE `survey` (  
2      `id` INT NOT NULL AUTO_INCREMENT,  
3      `FeedbackFormId` INT,  
4      `married` INT, 5      `haskids` INT, 6      `ownscar` INT,  
7      `makesover50k` INT,  
8      PRIMARY KEY (`id`)  
9  );
```

SAMPLE CODE 12: Example *survey* database table SQL.

Example code: [chart2.php](#)

In Sample Code 13, on lines 5 and 6, the additional JavaScript files that allow us to create the bar chart are imported. On lines 8–12, the browser is set not to allow selections. On line 16, the *container div* tag is added to hold the new chart.

Three different arrays are set up to hold the data for the chart. On line 21, The first of these titled *lab* array is to be used to hold each of the labels which will be added to the chart. On line 22, the *Data1* is added, which is responsible for holding all of the 0 data, e.g., *no*. On line 23, the *Data2* is added which is responsible for holding all the 1 data, e.g., *yes*.

After the arrays have been set up in JavaScript, they next have to be populated with data. To do this a custom PHP function titled *getData()* is called on line 24. This function takes in the name of the column we would like to add to the chart. On line 29, the color of the *No* bar which will be added is set to *red*. This is followed by the *Yes* bar being set to *blue* on line 35. On line 46, the graph is initialized by passing a reference to the canvas in the form of a reference titled *ctx*. On line 48, the data is passed into the graph.

```

1  <!doctype html>
2  <html> 3 <head>
    4          <title>Bar Chart</title>
    5          <script src="Chart.bundle.js"></script>
    6          <script src="utils.js"></script>
    7          <style>
    8          canvas {
    9              -moz-user-select: none;
10             -webkit-user-select: none; 11             -ms-user-select: none;
12             }
13         </style>
14     </head>
15     <body>
16         <div id="container" style="width:
17             75%;">
18             <canvas id="canvas"></canvas>
19             </div>
20             <script>
21                 var color = Chart.helpers.color;
22                 var lab = new Array();
23                 var data1 = [];
24                 var data2 = [];
25                 <?php getData('married'); ?>
26                 var barChartData = {
27                     labels: lab,
28                     datasets: [{
29                         label: 'No',
30                         backgroundColor:
31                             color(window.chartColors.red).
32                             alpha(0.5).rgbString(),
33                         borderColor: window.chartColors.red,
34                         borderWidth: 1,
35                         data: data1,
36                     }, {
37                         label: 'Yes',
38                         backgroundColor:
39                             color(window.chartColors.blue).
40                             alpha(0.5).rgbString(),
41                         borderColor:
42                             window.chartColors.blue,
43                         borderWidth: 1,
44                         data: data2,
45                     }
46                 ];
47                 window.onload = function() {
48                     var ctx = document.getElementById("canvas").
49                     getContext("2d");
50                     window.myBar = new Chart(ctx, {
51                         type: 'bar',
52                         data: barChartData,
53                         options: {
54                             responsive: true,
55                             legend: {
56                                 position: 'top',
57                             },

```

```

54                                     title: {
55                                     display: true,
56                                     text: 'Chart.js Bar Chart'
57                                     }
58                                     }
59                                     }); };
60                                </script>
61                                </body> 62 </html>

```

SAMPLE CODE 13: chart2.php — Sample binary data bar chart.

In Sample Code 14, the core of this bar chart generation process is the custom PHP function that is used to gather the data. This function begins on line 64 as a simple function definition with one variable being passed into it titled *column*. This is the name of the database column that we wish to gather the data from.

Inside of this function, on line 70, a connection to the database is first made, and the SQL statement is created on line 72. In this query, the *SELECT* statement selects the *column* that was passed to the function. The SQL *count()* function is then used to count the number of records for the defined column. This allows the one function to be used to return information about any column in the database table.

A *for each* statement on line 80 is used to iterate through each of the rows returned from the database. Once the query has run, a collection of counted records will be returned. Before the data is used by the chart, on line 83 and 86, a PHP *echo* is used to place modified JavaScript code on the page for both the count of *Yes* records if we are viewing 1 records and 0 if we are working with the count of *No* records. These *echo* statements generate JavaScript array values that are read in by the chart.

```

63                                     <?php
64                                     function
                                getData($column){
65                                     try {
66                                     $host =
                                'localhost:3306';
67                                     $dbname =
                                'bookexamples';
68                                     $user = 'sampleuser';
69                                     $pass =
                                '3C9038A509BE6145AAD6827B4568AD918BC3DAC0';
70                                     $DBH = new
                                PDO("mysql:host=$host;dbname=$dbname",
71                                     $user, $pass);
72                                     $sql = "SELECT

```

```

73         $column, count($column) FROM survey WHERE
74                                     FeedbackFormId = 1
75                                     GROUP BY
76                                     $column";
77                                     $sth = $DBH-
78                                     >prepare($sql);
79                                     $sth->execute();
80                                     $res = $sth-
81                                     >fetchAll();
82                                     // add the label
83                                     echo 'lab.push("'.$column.'");';
84                                     foreach ($res as
85                                     $row){
86                                     // its is 0 add to first
87                                     if($row[0] == 0){
88                                     echo " data1.push("'.$row[1].'");";
89                                     } //if it is 1 add to
90                                     the second column
91                                     else if($row[0] == 1)
92                                     {
93                                     echo "data2.push("'.$row[1].'");";
94                                     }
95                                     }
96                                     }
97                                     catch(PDOException $e) {echo $e;}
98                                     }
99                                     ?>

```

SAMPLE CODE 14: chart2.php — Example PHP to retrieve the records for a single column by name.

Output

In we can see the result of calling the *getData()* function to gather the values for the *married* column. Both have a count of Yes and a count for No.

DYNAMIC DASHBOARD

Although we looked at gathering data and placing results into individual database tables, there was no complete solution. In the following sections, we will introduce the concept of *campaigns*. Each campaign is

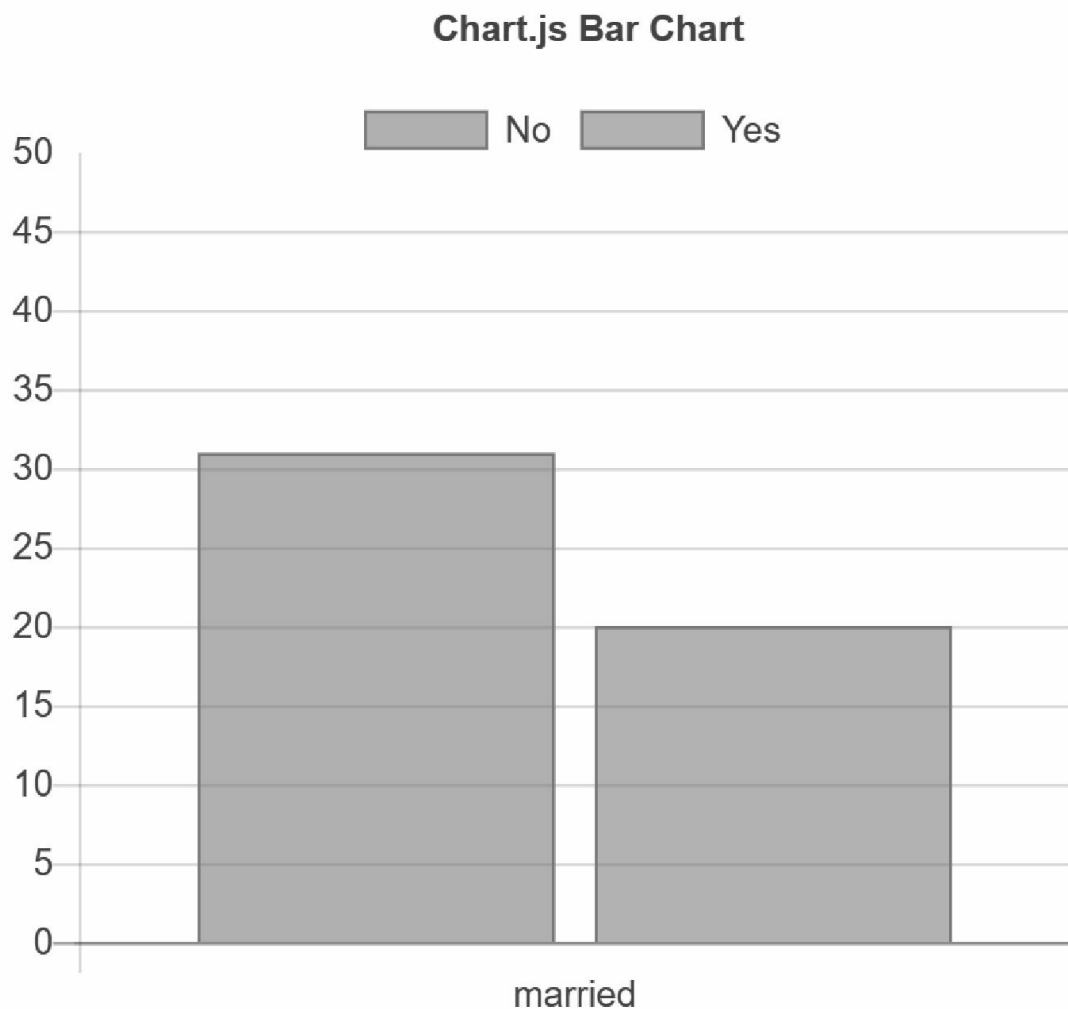


FIGURE 4.9: Two-column bar chart for binary data.

considered one complete data gathering and analysis solution that will be presented on a dashboard.

Example 5: Basic dashboard

To view data, a dashboard is designed to allow users to dynamically select different data sources and render visualizations. This example includes Bootstrap, Popper.js, jQuery, and jQueryUI.

1. `campus.sql` — This contains the *dashboard camps* table and the *dashboard camp elements* table that are used for storing data about

an individual campaigns.

2. dashboard1.php — A sample dashboard that is styled using Bootstrap.
3. Bootstrap — The JavaScript and CSS is used to provide a layout to the dashboard.
4. dashboard.css — This contains additional CSS for the dashboard layout.
5. Popper.js — This is used as a tooltip positioning engine for Bootstrap.
6. jQuery/jQueryUI — This library is used to provide modals and for selecting individual tags using the jQuery selector.

Example code: **camp.sql**

Before beginning with the process of editing the HTML for the dashboard, a new database table is created titled *dashboard camps*, as shown in Sample Code 15. This will be used as the main location for storing an individual campaign in the system. Each campaign will contain a date when it was first created and a brief description that is used as the method of describing the campaign to the user. The core identifier used for an individual campaign is the *campid*. Unlike other fields, the *campid* is not auto-incremented as legacy labels and IDs may have already been created. In the case where they are not, a simple integer can be used for each unique campaign.

```
1          CREATE TABLE `dashboard_camps` (  
2          `id` INT NOT NULL AUTO_INCREMENT,  
3          `datemade` DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,  
4          `campname` VARCHAR(100), 5          `campid` INT, 6          PRIMARY KEY (`id`)  
7);
```

SAMPLE CODE 15: Example *dashboard camps* database table.

Each individual campaign can contain a number of different campaign elements, e.g., total sales data that is an integer that is stored in a database table X with column name Y. As the data may not be stored in a database, the method used to store these data sources should remain

flexible. To achieve this a second database table titled *dashboard camp elements* is created. This table will contain all the data about the individual data sources a single campaign may have. The table consists of a unique primary key titled *ID*, followed by the *campID* that is linked to the *campID* in the previous table. The *description* column is used to provide a brief description to the user as to what the element is used for.

The *datatype* column is used to define the data type, e.g., int, string, boolean, etc. This is used as a flag for the developer in future steps when this data is going to be used. The final column titled *datasource* contains a reference to where the element being visualized is stored, e.g., in a MySQL database table or a CSV file that was uploaded to the server. The *options* column is used to hold specific data about the columns that should be retrieved from the data source. The final column is the *chart* that contains a description of the type of chart that the data source is associated with, e.g., a bar chart or line chart. Although these two examples are used for the dashboard, additional chart types can easily be added.

```
1      CREATE TABLE `dashboard_camp_elements` (  
2      `id` INT NOT NULL AUTO_INCREMENT,  
3      `campID` INT,  
4      `desc` VARCHAR(500),  
5      `datatype` VARCHAR(100),  
6      `datasource` VARCHAR(100),  
7      `options` VARCHAR(500),  
8      `chart` VARCHAR(50),  
9      PRIMARY KEY (`id`)  
10     );
```

Output

describes a sample database table containing values for each of the attributes.

To create a dashboard, Bootstrap offers a sample dashboard layout that can be used for the basic configuration shows an example of the Bootstrap theme with very minimal modifications to remove unnecessary links and data. This sample can be seen live at.

 id	campID	desc	 datatype	datasource	options	chart
14	1	sales	int	mysql	select Score, count(Score) from feedbackform...	barchart
20	1	products	int	csv	products.csv,0,1	linechart
21	2	products	int	csv	products.csv,0,1	barchart

FIGURE 4.10: Database table with sample data sources and configuration options.

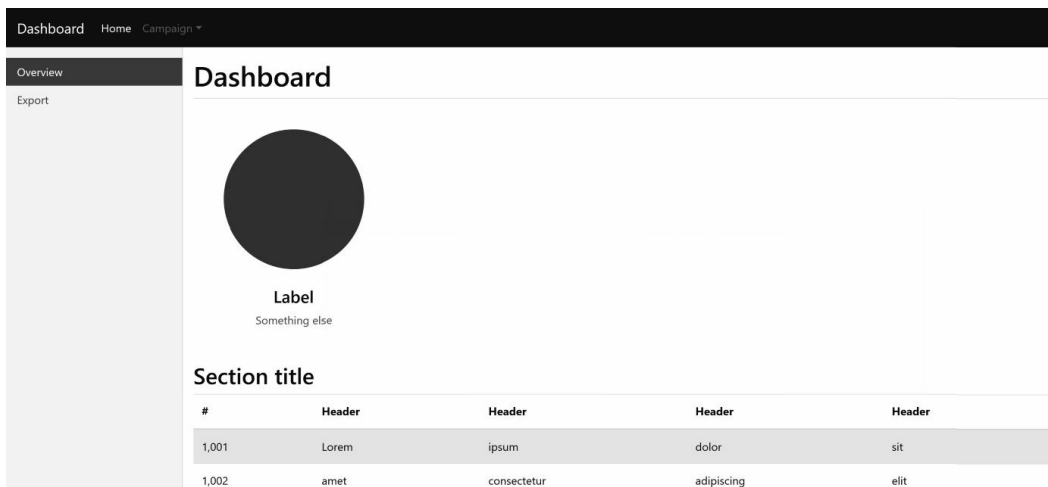


FIGURE 4.11: Sample Bootstrap dashboard.

Example code: dashboard1.php

In Sample Code 16, a new file titled *dashboard1.php* is created. This will be the main portal when a user has logged into the system. At the top of this page inside the *head* tag on lines 4–8, the default tags are added, including the *viewport*, *description*, *author*, and *title* for the page. This is followed by the CSS that is used for the core Bootstrap theme on line 9. A custom CSS file titled *dashboard.css* is also added on line 12 that can be modified to suit any application theme. On lines 12–15, jQuery, Popper.js, Bootstrap, and jQueryUI are included.

Once the libraries have been imported, the *header* for the dashboard is included on line 20. This header contains a *nav* tag that has the *class* set to *navbar* including the *navbar-dark* theme that is fixed to the top of the page.

On line 22, a link is included with the *class* set to *navbar-brand* which is a reference to the main text focused on the page. In this example, the word *Dashboard* is used as the title. A *div* tag is added with the *class* set to *collapse navbar-collapse* on line 23. This adds an

additional item to the nav bar that can be used as a drop-down menu. In this example it is used to display the word *Home*. On line 26, a link to the homepage or the start page of the dashboard is placed in the *href* tag.

On line 28, a navbar item titled *Campaign* is added that will be used as the root of the campaign. This contains three options on lines 32–34, *Add New Visualization*, *Build New Campaign* and *Add Campaign Elements*. As each of these items are children of the *Campaign* navbar item, the *class* of each is set to *dropdown-item*. Three separate modals will open when each item is clicked. To trigger this process a unique *href* value is given to each.

On line 38 and 41, two additional links are added for the custom *Text Analysis* feature and the *View Red Flags* feature that will be added in a later chapter.

```

1          <!doctype html>
2          <html lang="en">
3          <head>
4          <meta charset="utf-8">
5          <meta name="viewport" content="width=device-width,
      , _ initial-scale=1, shrink-to-fit=no">
6          <meta name="description" content="">
7          <meta name="author" content="">
8          <title>Dashboard</title>
9          <link rel="stylesheet" href="Bootstrap.min.css">
10         <link rel="stylesheet"
      , _ href="https://code.jquery.com/ui/1.12.1/themes/base/
11         jquery-ui.css">
12         <link href="dashboard.css" rel="stylesheet">
13         <script src="jquery-3.1.0.min.js"></script>
14         <script src="popper.min.js"></script>
15         <script src="Bootstrap.min.js"></script>
16         <script
      , _ src="https://code.jquery.com/ui/1.12.1/jquery-ui.js">
17         </script>
18         </head>
19         <body>
20         <header>
21         <nav class="navbar navbar-expand-md navbar-dark
      , _ fixed-top bg-dark">
22         <a class="navbar-brand" href="#">Dashboard</a>
23         <div class="collapse navbar-collapse"
      , _ id="navbarsExampleDefault">
24         <ul class="navbar-nav mr-auto"> 25         <li class="nav-item active">
      <a class="nav-link" href="#">Home

```

```

27         , _ class="sr-only">(current)</span></a>
28     </li>
29     <li class="nav-item dropdown">
30         <a class="nav-link dropdown-toggle"
31             , _ href="#" id="dropdown01"
32             , _ data-toggle="dropdown"
33             , _ aria-expanded="false">Campaign</a>
34         <div class="dropdown-menu"
35             , _ aria-labelledby="dropdown01">
36             <a class="dropdown-item" id="addViz"
37                 , _ href="#addAtt" data-toggle="modal"
38                 , _ >Add New Visualisation</a>
39             <a class="dropdown-item"
40                 , _ href="#buildCamp"
41                 , _ data-toggle="modal" >Build New
42                 , _ Campaign</a>
43             <a class="dropdown-item"
44                 , _ href="#addElements"
45                 , _ data-toggle="modal" >Add Campaign
46                 , _ Elements</a>
47             </div>
48         </li>
49     <li class="nav-item dropdown">
50         <a class="nav-link dropdown-toggle"
51             , _ href="#" id="dropdown01"
52             , _ data-toggle="dropdown"
53             , _ aria-expanded="false">Text
54             , _ Analysis</a>
55         <div class="dropdown-menu"
56             , _ aria-labelledby="dropdown02">
57             <a class="dropdown-item"
58                 , _ href="#selectRanges" , _ data-toggle="modal" >View
59                 Red
60                 , _ Flags</a>
61             </div>
62         </li>
63     </ul>
64 </div>
65 </nav>
66 </header>

```

SAMPLE CODE 16: dashboard1.php — Simple Bootstrap dashboard — part 1.

To add extra options, on line 48, an additional *div* is added with the class set to *container-fluid*. Inside this, a *nav* tag is added with the *class* set to *small*, with a *light* theme. Two options are added on lines 53 and

58 in the form of two *li* tags with the *class* set to *nav-item*. Although currently unused, a *href* can be set here.

In Sample Code 17, a tag titled *main* is added that represents the center of the dashboard. Inside of this tag a *div* tag is added with the *ID* set to *output* on line 66. As visualizations are created, they will be appended into this *div* tag.

```

48                                     <div class="container-fluid">
49                                     <div class="row">
50                                     <nav class="col-sm-3 col-md-2 d-none d-sm-block bg-
light
51                                     sidebar">
52                                     <ul class="nav nav-pills flex-column">
53                                     <li class="nav-item">
54                                     <a class="nav-link active" href="#">Overview
55                                     <span class="sr-only">(current)
56                                     </a>
57                                     </li>
58                                     <li class="nav-item">
59                                     <a class="nav-link" href="#">Export</a>
60                                     </li>
61                                     </ul>
62                                     </nav>
63                                     <main role="main" class="col-sm-9 ml-sm-auto col-
md-10
64                                     pt-3">
65                                     <h1>Dashboard</h1>
66                                     <div id="output">
67                                     </div>
68                                     <section class="row text-center
placeholders">
69                                     </section>
70                                     </div>
71                                     </div>

```

SAMPLE CODE 17: dashboard1.php — Simple Bootstrap dashboard — part 2.

Output

Figure 4.12 shows the dashboard with a custom navigation bar.

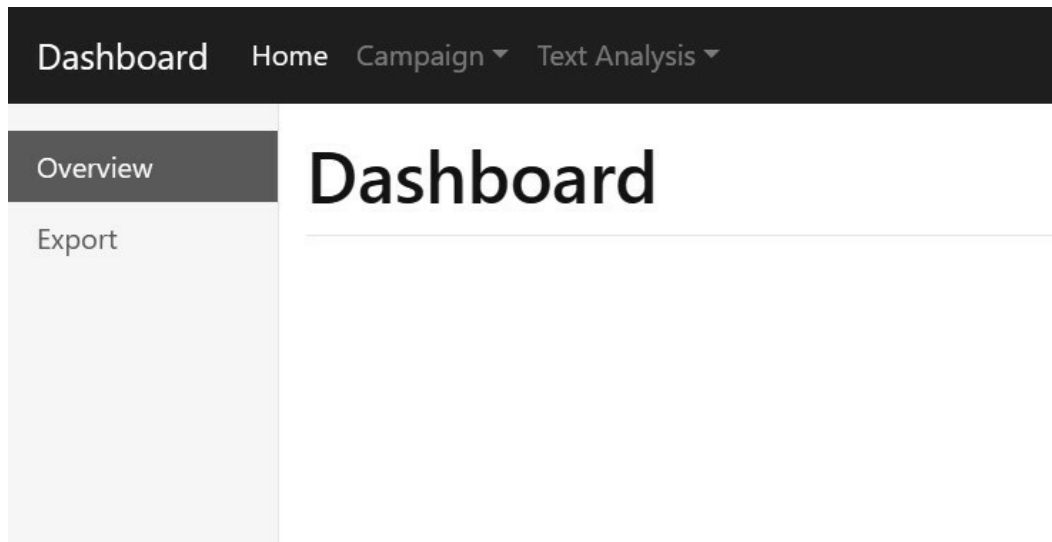


FIGURE 4.12: Bootstrap default dashboard with a nav bar.

Example 6: Select campaign elements

When the Campaign menu on the dashboard is clicked, followed by the *Add New Visualization*, a modal is required to present the user with a list of campaigns and campaign attributes.

This example includes Bootstrap, Popper.js, jQuery, and jQueryUI.

1. `dashboard2.php` — This is a modification to the original dashboard to present the user with campaign options.
2. `ajax.php` — This file is used to retrieve campaign elements and data from the database.
3. Bootstrap — The JavaScript and CSS is used to provide a layout to the dashboard.
4. `dashboard.css` — This contains additional CSS for the dashboard layout.
5. Popper.js — This is used as a tooltip positioning engine for Bootstrap.
6. jQuery/jQueryUI — This library is used to provide modals and for selecting individual tags using the jQuery selector.

Example code: `dashboard2.php`

In Sample Code 18, to create the modal, a *div* tag is added with the

class set to *modal fade* on line 2. This modal is uniquely identified by the *ID* that is set to *addAtt*. By default, the modal is hidden on the page until it has been opened by the user. The *role* of the *tag* is set to *dialog*. The *aria-labelledby* is set to *addAtt*, that is similar to creating a name to link a label to the *div*. Inside this *div*, a second *div* on line 4 is added with the *class* set to *modal-dialog*. This contains a *div* tag on line 6 with the *class* set to *modal-header* that is displayed at the top of the modal. In this header, on line 7, a *title* is added with the text *Select Campaign*. On line 8, a *button* tag is used on the right-hand side of the modal to provide the user an option to close the modal if required.

In the body of the modal, a row is added on line 14 that contains a *div* tag with the *class* set to *col-4* on line 15. Inside this, a *list-group* is added for structure on line 16. After this, on line 19, a second *div* is added with the *class* set to *col-8* that provides a wider window for the content to be displayed. A *list-group* is again added for structure on line 20. On lines 27 and 28, two buttons are added to the end of the modal. The first of these is the *Close* button, allowing the user to dismiss the modal followed by the *Add Vizualization* button. When this button is clicked, the selected element is used as the starting point for the call that is made to the *addViz()* JavaScript function that is responsible for adding the visualization to the dashboard. The modal does not have any content specified in this list. The list of campaigns and campaign attributes is added later using JavaScript.

```

1          <!-- Select Campaign Modal -->
2      <div class="modal fade" id="addAtt" tabindex="-1" role="dialog"
3          aria-labelledby="addAtt" aria-hidden="true">
4      <div class="modal-dialog" role="document">
5          <div class="modal-content">
6              <div class="modal-header">
7                  <h5 class="modal-title" id="addAtt">Select Campaign</h5>
8                  <button type="button" class="close" data-dismiss="modal"
9                      ,_ aria-label="Close">
10                     <span aria-hidden="true">&times;</span>
11                 </button>
12             </div>
13             <div class="modal-body"> 13          <!-- List of camps-->
14                 <div class="row">
15                     <div class="col-4">
16                         <div class="list-group" id="list-tab-camp"
17                             ,_ role="tablist">

```

```

17                                </div>
18                                </div>
19                                <div class="col-8">
20                                <div class="list-group" id="list-tab-att"
, _ role="tablist">
21                                Click on a campaign for attributes
22                                </div>
23                                </div>
24                                </div>
25                                </div>
26                                <div class="modal-footer">
27                                <button type="button" class="btn btn-secondary"
, _ data-dismiss="modal">Close</button>
28                                <button onclick="addViz()" type="button" class="btn
, _ btn-primary">Add Visualisation</button>
29                                </div>
30                                </div>
31                                </div>
32                                </div>

```

SAMPLE CODE 18: dashboard2.php — List of campaigns modal.

In Sample Code 19, on line 35, the jQuery selector is used to select the menu item with the *ID* set to *addViz*. When this is clicked, on line 36, a jQuery *POST* is performed to the *ajax.php* page. The *type* being sent to the page is set to *getcamp*. This function returns a variable titled *data* on line 37 that contains the list of campaigns marked up in HTML. On line 37 the *list-tab-camp* element is selected and the HTML is set using the *html()* function and passing in the *data* variable that was returned.

When each campaign is clicked, a list of elements is available for visualization. On line 41 to perform this action, a listener is bound to the *div* tag with the *ID* set to *list-tab-camp*. On *shown.bs.tab*, e.g., a person clicking on one of the campaigns, this function will then trigger. On line 42, a variable title *data* is added that will contain the HTML for the selected tab element. On line 43, the campaign *ID* is extracted from the data variable using the *attr()* function and passing in the name of the element of interest. In this case, the *cid*.

To retrieve a list of elements that make up the individual campaign, on line 45, a *POST* is made to the *ajax.php* passing the *type* of request as the *getformfields*. The *cid* for the individual campaign is also passed

to return only the relevant elements. On line 46, when the function is complete, the result will be returned inside a function with a variable titled *data* that is appended to the *list=tab-att* using the *html()* function on line 47.

```

33          <script>
34          // Add list of camp elements to the modal
35          $("#addViz").click(function() { console.log("Called add viz");
36          $.post( "ajax.php", { type: "getcamp"} )
37          .done(function( data ) {
38          $('#list-tab-camp').html(data);
39          });
40          });
41          $('#div[id="list-tab-camp"]').on('shown.bs.tab', function (e) {
42      var data = e.target; // HTML of just selected tab
43      var cid = $(data).attr('cid'); // get the ID
44          // get list of attributes for campaign
45          $.post( "ajax.php", { type:"getformfields", cid: cid})
46          .done(function( data ) {
47          $('#list-tab-att').html(data);
48          });
49          })
50          </script>

```

SAMPLE CODE 19: dashboard2.php — JavaScript for listing campaigns on the dashboard modal.

Example code: **ajax.php**

To retrieve a list of campaigns and campaign elements, the *ajax.php* page is modified. In Sample Code 20, on line 3, if the *type* is set to *getcamp* a list of campaigns are returned by calling the *getCamp()* PHP function on line 4. A second *type* attribute titled *getformfields* is added on line 6, for retrieving a campaign elements based on the *cid*.

When the campaign modal first opens, a list of campaigns is retrieved from the *dashboard_camps* table. The two pieces of information of interest are the *campid* and the *campname*. During each iteration of the *foreach* statement on line 22, a new *href* link is created. The *class* is set to *list-group-item list-group-item-action*. Custom attributes are added into the link to aid the movement of basic campaign data such as the *cid* attribute. The *list-camp* is set to be the *ID* of the campaign. The *datatoggle* attribute is set as a *list* on line 24. For the user, the text shown is the *campid* followed by a single dash

followed by the *campname* that is extracted from the row.

For each campaign in the modal that has been selected, an AJAX request is made to the *ajax.php* page making a call to the *getFormFields()* function.

```
1          <?php
2          $type = $_POST['type'];
3          if($type == 'getcamp'){
4              getCamp();
5          }
6          else if($type == 'getformfields'){
7              getFormFields();
8          }
9          function getCamp(){
10             try {
11                 $host = 'localhost:3306';
12                 $dbname = 'bookexamples';
13                 $user = 'sampleuser';
14                 $pass = '3C9038A509BE6145AAD6827B4568AD918BC3DAC0';
15                 $DBH = new PDO("mysql:host=$host;dbname=$dbname",
16                 $user, $pass);
17                 $sql = "select campid, campname from dashboard_camps;";
18                 $sth = $DBH->prepare($sql);
19                 $sth->execute();
20                 $res = $sth->fetchAll();
21
22             foreach($res as $row){ 23         $id = $row['campid'];
24                 echo '<a class="list-group-item list-group-item-action"
25                     cid="'. $id.'"
26                     id="list-camp-' . $id.'" data-toggle="list" href= 27 "#list-camp-' . $id.'"
28                     role="tab" aria-controls="camp-' . $id.'">CID:' .
29                     $row['campid'] . ' -
30                     ' . $row['campname'] . '</a>';
31             }
32
33         } catch(PDOException $e) {echo $e;}
34     }
```

SAMPLE CODE 20: ajax.php — Retrieving a list of campaigns — part 1

In Sample Code 21, on line 37, the function is defined. This function is responsible for returning a list of data about the elements that make up a single campaign. These Include the text-based description that was given to the element, the *dataType* that was being processed, information about the data source, e.g., if it was a CSV file or SQL and the type of chart that will be used to render the data on the dashboard.

On line 38, the *cid* that has been passed is stored in a local variable. On line 46, the SQL to select the campaign elements is added. A question mark is used for the *campID* attribute as it is dynamically passed to the function and then appended using the *bindParam()* function on line 49. On line 53, a *for each* statement is added that iterates over the results. On lines 54–58, for each record, the *ID*, description, data type, data source, and chart type are gathered and placed into local variables. On line 61, for each record a complete HTML link is created. Each of the attributes is placed into the link to aid future retrieval during the graph generation process.

```

35         // return a list of form fields
36         // for a given cid
37         function getFormFields(){
38             $cid = $_POST['cid'];
39             try {
40                 $host = 'localhost:3306';
41                 $dbname = 'bookexamples';
42                 $user = 'sampleuser';
43                 $pass = '3C9038A509BE6145AAD6827B4568AD918BC3DAC0';
44                 $DBH = new PDO("mysql:host=$host;dbname=$dbname",
45                     $user, $pass);
46                 $sql = "select * from dashboard_camp_elements where
47                     campID=?;";
48                 $sth = $DBH->prepare($sql);
49                 $sth->bindParam(1, $cid, PDO::PARAM_STR);
50                 $sth->execute();
51                 $res = $sth->fetchAll();

52
53                 foreach($res as $row){
54                     $elementID = $row['id'];
55                     $desc = $row['desc'];
56                     $dataType = $row['datatype'];
57                     $src = $row['datasource'];
58                     $chart = $row['chart'];
59                     // print out each individual attribute as a new button 60 // inside of the tab
61                     echo '<a chart="'.$chart.'" datasource="'.$src.'"
62                         elementid="'.$elementID.'"
63                         elementtitle="'.$desc.'" cid="'.$cid.'"
64                         class="list-group-item list-group-item-action"
65                         id="list-camp-e" data-toggle="list" href="#"
66                         role="tab"
67                         aria-controls="camp">'.$desc.' - ( '.$dataType.' ) -
68                         '.$src.'</a>';
69                 }
70                 catch(PDOException $e) {echo $e;}
71             }
72             ?>

```

SAMPLE CODE 21: ajax.php - Retrieving a list of form fields - part 2

Output

shows the *Select Campaign* modal. This modal has a list of the active campaigns on the left. When the campaign is selected, a list of the associated elements is shown on the right of the modal. This contains information about the element description, the format of the data, and the source of the data.

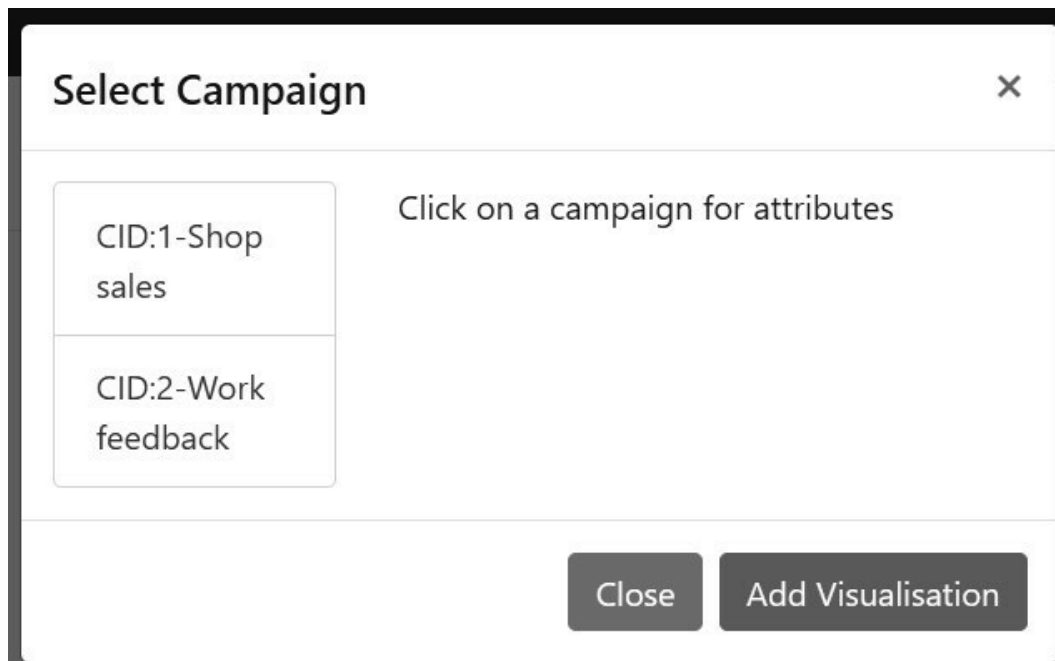


FIGURE 4.13: Select campaign modal with two available campaigns.

Figure 4.14 describes a list of campaigns on the left followed by a list of campaign elements that can be visualized on the right. Once an item has been selected in the right menu, the *Add Visualization* option can be selected.

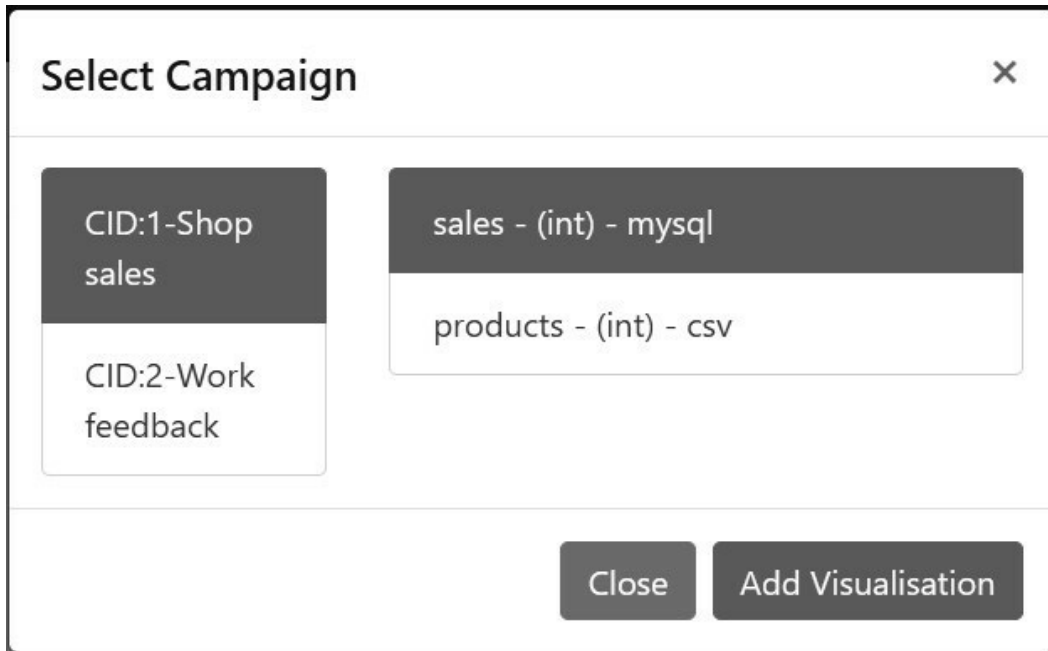


FIGURE 4.14: Viewing a list of campaigns with a single campaign element selected.

Example 7: Adding a visualization

When any of the campaign elements are clicked on the campaign modal they will become blue, indicating they have been selected. To add a visualization based on this selected campaign element, a call is made to the *addViz()* function when the *Add Visualization* button is clicked. This example contains four files:

1. `dashboard3.php` — This file contains the original dashboard HTML/JavaScript with an additional function to add visualizations.
2. `data_processor.php` — This file contains the PHP to access the database or file depending on the data source chosen.
3. `line.php` — This contains the HTML/JavaScript to generate a `Charts.js` line chart.
4. `barchart.php` — This contains the HTML/JavaScript to generate a `Chart.js` bar chart.

Example code: `dashboard3.php`

In Sample Code 22, the *addViz()* function is described that is added

inside the *script* tag at the bottom of the dashboard3.php file. On line 2, a simple *console.log()* is performed to indicate to us that the function has correctly been called. On lines 3–6, the *list-tab-att* items are searched for where the status is set to *a.active* indicating that it has been selected. For the selected campaign item, the *elementID*, *chart*, *datasource*, and *elementtitle* are extracted. These were placed in the individual items earlier during generation.

On line 11, a check is performed to see what type of chart is required. On line 11, if the *type* is equal to *barchart*, the jQuery selector is used to select the *output div* tag and append a new *div* tag. The contents of this *div* tag are gathered by dynamically calling the *barchart.php* page via the *load()* function and passing in the *source*, *ID*, and *title* variables. On line 16, the same process is repeated for the line chart.

```
1      function addViz(){
2          console.log('add viz called');
3      var elementid = $("#list-tab-att a.active").
4          attr('elementid');
5      var chartType = $("#list-tab-att a.active").
6          attr('chart');
7      var dataSource = $("#list-tab-att a.active").
8          attr('datasource');
9      var vizTitle = $("#list-tab-att a.active").
10         attr('elementtitle'); 11      if(chartType == 'barchart'){
12         $('#output').append(($('<div>').load("bar.php?source="
13             +dataSource+"&id="
14             +elementid + "&title="+escape(vizTitle))));
15         }
16         else if(chartType == 'linechart'){
17             $('#output').append(($('<div>').load("line.php?source="
18                 +dataSource+"&id="
19                 +elementid + "&title="+escape(vizTitle))));
20             }
21         }
```

SAMPLE CODE 22: dashboard3.php — addViz() function to add a new visualization based on the current selected campaign element.

Example code: data processor.php

Both the *barchart.php* and *line.php* file require use of a file titled *data processor.php*. This file is used as the main method of gathering data

from different data sources.

In Sample Code 23, on line 2, the *getData()* function is defined that takes in the *ID* of the element to chart along with the *sourceType* that references where the data is currently stored. On line 3, if the source type is a MySQL database, then a query to the database is made. The query is gathered by making a call to the *getDataSourceOptions()* function on line 11 and passing the *ID* of the element to be visualized. For each row that is returned the *row[0]* that contains the X value is added to the JavaScript *lab* array by calling the *push()* function on line 16 and passing the term. On line 17, the Y value to be displayed to modify the height of the bar is gathered from the *row[1]* and pushed to the *data1* JavaScript array.

On line 23, if the *sourceType* variable is set to *csv*, then a call to the *getDataSourceOptions* is made on line 24. This function takes in the *ID* of the elements to be graphed. The data that will be plotted must follow the same structure of X and Y. Where X is the label and Y is the value to be plotted. During the process of adding a CSV to the system, the two columns are defined, and then the file name is described. In the database table, these values take the form of *filename.csv, 1, 2* where 1 is the X column and 2 is the Y column of the CSV file to be plotted.

To break the values apart for processing, a call is made to the *explode()* function on line 25 that takes in a delimiter followed by the string to process. In this example, the delimiter is a comma. A call to *fgetcsv()* is made on line 27 to open a connection to the file and gather the contents. As the file is a CSV, the delimiter is once again specified as a comma here. The *label* for the chart is stored in the *lab* JavaScript array.

The values for this are extracted from the *data* variable that contains the individual columns in the CSV file. On lines 30 and 31, a reference to *col1* and *col2* is provided to identify the two columns of interest and extract their values. The first of these is the label and the second is the value associated to the label. These are printed using an *echo* statement that contains the JavaScript to add to the *lab* and *data1* JavaScript arrays. The JavaScript *push()* function is used to add the values to the associated arrays.

```

1          <?php
2          function getData($id, $sourceType){
3              if($sourceType == 'mysql'){
4                  try {
5                      $host = 'localhost:3306';
6                      $dbname = 'bookexamples';
7                      $user = 'sampleuser';
8                      $pass =
39          '3C9038A509BE6145AAD6827B4568AD918BC3DAC0';
9                      $DBH = new PDO("mysql:host=$host;dbname=$dbname",
10                         $user, $pass);
11                      $sql = getSourceOptions($id);
12                      $sth = $DBH->prepare($sql);
13                      $sth->execute();
14                      $res = $sth->fetchAll();
15                      foreach ($res as $row){
16                          echo " lab.push('".$row[0]."',");
17                          echo " data1.push('".$row[1]."',");
18                      }
19
20                      } catch(PDOException $e) {echo $e;}
21
22                      }
23                      else if($sourceType == 'csv'){
24                          $dataOptions = getSourceOptions($id);
25                          $op = explode(',', $dataOptions);
26                          if (($file = fopen('samplefiles/'.$op[0], "r")) !== FALSE)
27                              {while (($data = fgetcsv($file, 1000, ",")) !== FALSE)
28                                  {$col1 = $op[1];
29                                  $col2 = $op[2];
30                                  echo " lab.push('".$data[$col1]."',");
31                                  echo " data1.push('".$data[$col2]."',");
32                              }
33                              fclose($file);
34                          }
35                      }
36
37
38          }
39          function getSourceOptions($id){
40              try {
41                  $host = 'localhost:3306';
42                  $dbname = 'bookexamples';
43                  $user = 'sampleuser';
44                  $pass = '3C9038A509BE6145AAD6827B4568AD918BC3DAC0';
45                  $DBH = new PDO("mysql:host=$host;dbname=$dbname",
46                     $user, $pass);
47                  $sql = "SELECT options FROM dashboard_camp_elements
48                     WHERE id = ?";
49                  $sth = $DBH->prepare($sql);
50                  $sth->bindParam(1, $id, PDO::PARAM_INT);
51                  $sth->execute();
52                  $res = $sth->fetch();
53                  return $res[0];

```

```

54         } catch(PDOException $e) {echo $e;}
55     }

```

SAMPLE CODE 23: `data_processor.php` — Get data from a data source depending on the data source type.

Example code: **line.php**

A file titled *line.php* is created to visualize line charts. On line 2, the *data processor.php* file is included that will provide access to the data source as described earlier. On lines 11 and 12, the two JavaScript files required by Charts.js are included. Both these files must be in the current directory. On lines 14–18, the *canvas* is set to not allow selections. On line 22, a *div* is added that will be used for holding the visualization. On line 27, a random number is generated and placed into a local variable titled *uniqueID*. This is used as the *ID* for the graph. On lines 38 and 30, the *ID* is set into the *container* and the *canvas*. On line 36 a call is made to the PHP function titled *getData()* that was described earlier. The *ID* and the *sourceType* are passed to this function.

On line 38, the *config* variable is added that contains the *type* of graph to be rendered. the *labels* is set to be the *lab* array that was generated. Inside the *datasets* array on line 43, the *data* for the graph is set to be the *data1* array that was generated.

```

1  <?php
2  include('data_processor.php');
3  $id = $_GET['id'];
4  $title = $_GET['title'];
5  $sourceType = $_GET['source'];
6  ?>
7  <!doctype html>
8  <html> 9 <head>
10         <title>Line Chart</title>
11         <script src="Chart.bundle.js"></script>
12         <script src="utils.js"></script>
13         <style> 14     canvas{
15                 -moz-user-select: none;
16                 -webkit-user-select: none; 17     -ms-user-select: none;
18         }
19     </style>
20 </head>
21 <body>
22     <div id="container" style="width:50%"; class="ui-widget-
23     content">

```

```

24         <canvas id="canvas"></canvas>
25     </div>
26     <script>
27     var uniqueID = Math.floor((Math.random() * 1000) + 1);
28     document.getElementById("container").setAttribute("id",
29     "container"+uniqueID);
30     document.getElementById("canvas").setAttribute("id",
31     "canvas"+uniqueID);
32     var lab = new Array();
33     var data1 = new Array();
34     <?php
35     // Get the data for the graph
36     getData($id,$sourceType);
37     ?>
38     var config = {
39     type: 'line',
40     data: {
41     labels: lab, 42     datasets: [{
43         label: "",
44         backgroundColor: window.chartColors.red,
45         borderColor: window.chartColors.red,
46         data: data1,
47         fill: false,
48     }]
49     },

```

SAMPLE CODE 24: line.php — Used when adding a visualization — part 1.

In Sample Code 25, on line 84, the chart reference *ctx* and the *config* are passed to to create a new *chart* object. On lines 85 and 86, the *draggable()* and the *resizable()* functions are called on the container allowing it to be moved on the screen. These functions are components of the jQueryUI library that is added as part of the original dashboard.

```

50     options: {
51     responsive: true, 52     title: {
53     display: true,
54     text: unescape('<?php echo $title;?>')
55     },
56     tooltips: {
57     mode: 'index',
58     intersect: false,
59     },
60     hover: {
61     mode: 'nearest',
62     intersect: true
63     },
64     scales: {
65     xAxes: [{
66     display: true, 67     scaleLabel: {
68         display: true,

```



```

69         labelString: 'Month'
70     }
71 },
72     yAxes: [{
73         display: true,74         scaleLabel: {
75             display: true,
76             labelString: 'Value'
77         }
78     }]
79 }
80 }
81 };
82
var ctx = document.getElementById("canvas"+uniqueID).
    getContext("2d");
83
84 window.myLine = new Chart(ctx, config);
85 $("#container"+uniqueID).resizable();
86 $("#container"+uniqueID).draggable();
87 </script>
88 </body>89 </html>

```

SAMPLE CODE 25: line.php — Used when adding a visualization — part 2.

Example code: **bar.php**

In Sample Code 26 and 27, the *bar.php* file can be seen. This file has the exact same JavaScript as the *line.php* file. The only modification that needs to be made is on line 52, the *type* is set to *bar*.

```

1  <?php
2  include('data_processor.php');
3  $id = $_GET['id'];
4  $title = $_GET['title'];
5  $sourceType = $_GET['source'];
6  ?>
7  <!doctype html>
8  <html> 9 <head>
10     <script src="Chart.bundle.js"></script>
11     <script src="utils.js"></script>
12     <style>
13     canvas {
14         -moz-user-select: none;
15         -webkit-user-select: none;
16         -ms-user-select: none;
17     }
18     </style>
19     </head>
20     <body>
21     <div id="container" style="width:50%"; class="ui-widget-content">
22         <canvas id="canvas"></canvas>
23     </div>
24     <script>

```

```

25     var uniqueID = Math.floor((Math.random() * 1000) + 1);
26     document.getElementById("container").setAttribute("id",
27     "container"+uniqueID);
28     document.getElementById("canvas").setAttribute("id",
29     "canvas"+uniqueID);
30     var color = Chart.helpers.color;
31     var lab = new Array();
32     var data1 = new Array();
33     <?php
34     // Get the data for the graph
35     getData($id, $sourceType);
36     ?>
37     var barChartData = {
38         labels: lab, 39         datasets: [{ 40 label: ",
41             backgroundColor: color(window.chartColors.red).
42             alpha(0.5).rgbString(),
43             borderColor: window.chartColors.red,
44             borderWidth: 1,
45             data: data1 46         }]
47
48     };
49     var ctx = document.getElementById("canvas"+uniqueID).
50         getContext("2d");
51     window.myBar = new Chart(ctx, {
52         type: 'bar',
53         data: barChartData,

```

SAMPLE CODE 26: bar.php — Used when adding a visualization — part 1.

```

54         options: {
55             responsive: true,
56             scales: {
57                 yAxes: [{
58                     ticks: {
59                         beginAtZero: true
60                     }
61                 }]
62             },
63             legend: {
64                 display:false,
65             },
66             title: {
67                 display: true,
68                 text: unescape('<?php echo $title;?>')
69             }
70         }
71     });
72     $("#container"+uniqueID).resizable();
73     $("#container"+uniqueID).draggable();
74     </script>
75 </body> 76 </html>

```

SAMPLE CODE 27: bar.php — Used when adding a visualization — part 2.

Output

outlines the final dashboard that contains two different visualizations. The first of these is a bar chart and the second is a line chart. As the setup of the chart was dynamic, to add in further chart types requires minimal modification.

Dashboard

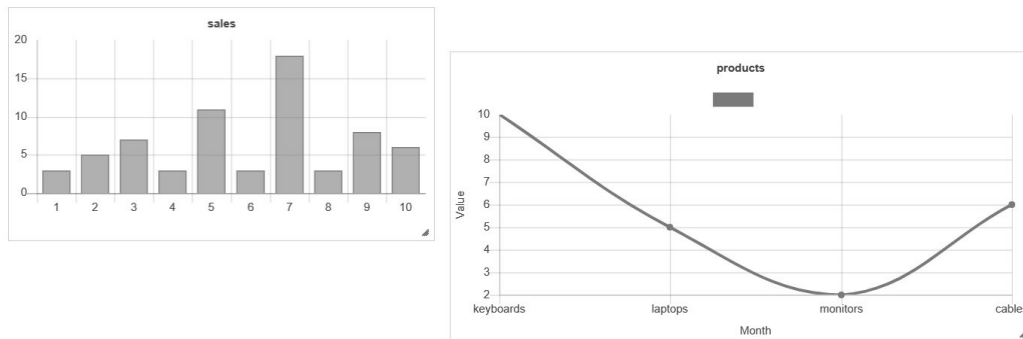


FIGURE 4.15: Example dashboard with two visualizations added.

Example 8: Adding a new campaign

The backbone of each visualization is the campaign. To add new campaigns into the system, a menu item titled *Build New Campaign* is available in the *Campaign* drop-down menu. When this is selected, a new modal is created that allows the user to enter the unique *ID* for the campaign, followed by a description. To do this, a new modal is first created.

This example consists of two files:

1. dashboard4.php — This is the original dashboard with an additional Bootstrap modal.
2. ajax.php — This file is used to dynamically insert a new campaign into the database.

Example code: dashboard4.php

In Sample Code 28, on line 2, a *div* tag is added with the *ID* set to *buildCamp*. On line 9, the title for the modal is set to *Create new*

campaign. On lines 21 and 22, two input fields are added with the IDs set to *newcid* and *newdesc*, respectively. On line 28, a close button is added, followed by another button on line 30 that is responsible for triggering the campaign to be added to the database. The *ID* of this button is set to *addNewCamp*.

```

1                                <!-- Build new camp modal -->
2                                <div class="modal fade" id="buildCamp" tabindex="-1"
3                                role="dialog"
4                                aria-labelledby="buildCamp"
5                                aria-hidden="true"> 6                                <div class="modal-dialog"
6                                role="document">
7                                <div class="modal-content">
8                                <div class="modal-header">
9                                <h5 class="modal-title" id="buildCamp">Create new
10                               Campaign</h5>
11                               <button type="button" class="close" data-dismiss="modal"
12                               aria-label="Close">
13                               <span aria-hidden="true">&times;</span>
14                               </button>
15                               </div>
16                               <div class="modal-body"> 17                               <!-- List of camps-->
18                               <div class="row">
19                               <div class="col-5">
20                               <div class="list-group" id="list-tab-camp" role="tablist">
21                               Campaign ID <input type="text" id="newcid"><br>
22                               Description <input type="text" id="newdesc"><br>
23                               </div>
24                               </div>
25                               </div>
26                               </div>
27                               <div class="modal-footer">
28                               <button type="button" class="btn btn-secondary"
29                               data-dismiss="modal">Close</button>
30                               <button id="addNewCamp" type="button" class=
31                               "btn btn-primary"
32                               data-dismiss="modal">Add New Campaign</button>
33                               </div>
34                               </div>
35                               </div>
36                               </div>
37                               <!--- END NEW CAMP MODAL -->

```

SAMPLE CODE 28: dashboard4.php — Additional add new campaign modal.

In Sample Code 29, on line 3, a jQuery listener is bound to the *addNewCamp* button. On line 5 and 6, when the button is clicked, the

jQuery selector is used to grab the *newcid* and the *newdesc* values from the modal. On line 7, these variables are *POST* to the *ajax.php* page under the *type* titled *addnewcamp*. A simple *alert()* is used to output the response after calling the *ajax.php* page.

```

1          <script>
2          // example 3 changes
3          $("#addNewCamp").click(function() {
4              alert("called");
5          var newcid = $("#newcid").val();
6          var newdesc = $("#newdesc").val();
7              $.post( "ajax.php", { type: "addnewcamp", newcid:newcid,
8                  newdesc:newdesc } )
9              .done(function( data ) {
10                  alert(data); 11          });
12
13      });
14      </script>

```

SAMPLE CODE 29: dashboard4.php — Add new campaign button click listener.

Example code: **ajax.php**

Sample Code 30 outlines the *ajax.php* page. On line 1, the *type* is first caught. On line 2, if the *type* is equal to *addnewcamp*, a call is made to the *addNewCamp()* function. On lines 6 and 7, the new *ID* and description are first extracted from the *POST* and placed in local variables. On line 13, a new connection to the database is created. On line 15, the SQL *INSERT* statement is created placing two question marks for the values. The values are bound by calling the *bindParam()* function on lines 18 and 19. On line 20, the *INSERT* statement is executed.

```

1      $type = $_POST['type'];
2      if($type == 'addnewcamp'){
3          addNewCamp();
4      }
5      function addNewCamp(){
6          $newcid = $_POST['newcid'];
7          $newdesc = $_POST['newdesc'];
8          try {
9              $host = 'localhost:3306'; 10          $dbname = 'bookexamples';
11              $user = 'sampleuser';
12              $pass = '3C9038A509BE6145AAD6827B4568AD918BC3DAC0';
13              $DBH = new PDO("mysql:host=$host;dbname=$dbname",

```

```

14         $user, $pass);
15         $sql = "INSERT INTO `dashboard_camps` (`campname`,
16         `campid`) VALUES (?,?);";
17         $sth = $DBH->prepare($sql);
18         $sth->bindParam(1, $newdesc, PDO::PARAM_STR);
19         $sth->bindParam(2, $newcid, PDO::PARAM_INT);
20         $sth->execute();
21         echo 'Added';
22     } catch(PDOException $e) {echo 'failed' . $e;}
23     }

```

SAMPLE CODE 30: ajax.php — Add a new campaign PHP.

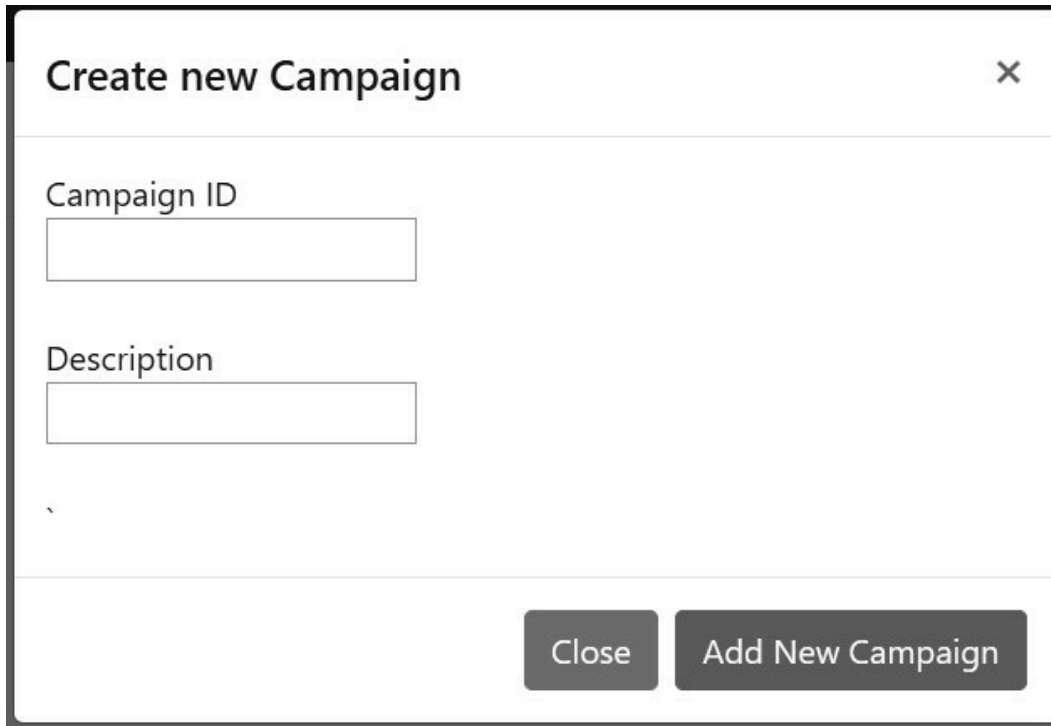
Output

Figure 4.16 shows the output after clicking *Add New Campaign*.

Example 9: Add new campaign element

The most complicated modal that is shown to the user is the modal that is designed to allow the user to specify a new campaign element. A campaign element is a single field in a campaign. This example consists of three files:

1. dashboard5.php — The original dashboard with a new modal for adding campaign elements.
2. ajax.php — This file is responsible for inserting a new campaign element record into the database.



The image shows a modal dialog box with the title "Create new Campaign" and a close button (X) in the top right corner. Inside the dialog, there are two text input fields: "Campaign ID" and "Description". At the bottom right of the dialog, there are two buttons: "Close" and "Add New Campaign".

FIGURE 4.16: Two text fields allowing the user to create a new campaign.

3. `listOfFiles.php` — This file is used to retrieve a list of files that can be used as a data source.

Example code: `dashboard.php`

In Sample Code 31, on line 2 a new *div* tag is created. The *ID* for this *div* is set to *addElement*. The *role* is set to *dialog*.

On line 19, an input field is added, allowing the user to enter the campaign *ID*. On line 21, another input field is added, allowing the user to enter a column name. As data can come in any format, on line 24, a *select* field is added, allowing the user to specify if the content is int, text, or decimal. On line 30, another field is added, allowing the user to determine if the content is coming from a CSV file or a MySQL database.

On line 35, a *select* is added, allowing the user to select if they want a bar chart or a line chart to appear on the dashboard. A *div* titled *dataSourceOptions* is added. When this modal loads, no content is

shown in this *div* tag. When the user selects MySQL, a text area will be shown in this *div*. When the user selects CSV, then a list of files will be shown to select from.

```

1          <!-- ADD NEW CAMP ELEMENT -->
2      <div class="modal fade" id="addElements" tabindex="1"
3          role="dialog"
4          aria-labelledby="addElements" aria-hidden="true">
5          <div class="modal-dialog">
6      <div class="modal-content" style="width:900px">
7          <div class="modal-header">
8      <h5 class="modal-title" id="addElements">Create New
9          Campaign Element</h5>
10         <button type="button" class="close" data-dismiss="modal"
11             aria-label="Close">
12         <span aria-hidden="true">&times;</span>
13         </button>
14         </div>
15         <div class="modal-body">
16         <div class="container">
17         <div class="row">
18         <div class="col-sm">
19         <b> Campaign ID</b><br> <input type="text"
20             id="addElementCid"><br>
21         <b> Custom Column Name </b><br><input type="text"
22             id="addElementCol"><br>
23         <b>Data type</b><br>
24         <select id="addElementDataType">
25         <option>int</option>
26         <option>text</option>
27         <option>decimal</option>
28         </select><br>
29         <b>Data Source</b><br>
30         <select id="addElementSource">
31         <option value="csv">CSV File</option>
32         <option value="mysql">MySQL Column</option>
33         </select><br>
34         <b>Chart Type</b><br>
35         <select id="chartType">
36         <option value="barchart">Bar Chart (X and y)
37         </option>
38         <option value="linechart">Line Chart (X and Y)
39         </option>
40         </select><br>
41         </div>
42         <div class="col-sm">
43         <b> Select data source</b><br>
44     <div id="dataSourceOptions" style="width:300px;
45         height:200px; overflow: auto">
46         </div>
47         <div id="dataSourceAdditional">

```


50

48

</div> 49

</div>

51

</div>

52

<div class="modal-footer">

53

<button type="button" class="btn btn-

secondary"

data-dismiss="modal">Close</button>

54

<button id="addNewCampElement"

55

type="button"

class="btn btn-primary"

56

data-dismiss="modal">Add New

57

Element

</button>

58

</div>

59

</div>

60

</div>

61

</div>

62

</div> 64 </div>

63

SAMPLE CODE 31: dashboard5.php — Add new campaign element modal.

Sample Code 32 describes the Javascript that is used to track the data source the user is adding. On line 65, when the source, e.g., CSV or MySQL is selected on the form, the *on()* function will trigger. On line 66, if the selected source is equal to *mysql*, on line 67, the *dataSourceOptions* div tag is selected, and the content is updated to be a sample SQL query. On line, 71, if a CSV is selected, then the *dataSourceOptions* div tag is selected and the contents are set to be the output of loading the *listOfFiles.php* file. This PHP file is used for listing the contents of a directory.

72

```

65      $('#addElementSource').on('change', function() {
66          if(this.value == 'mysql'){
67              $('#dataSourceOptions').html('<textarea rows="6"
, _ cols="35" id="querybox">select Score, count(Score) ,_ from feedbackformrecords where
FeedbackFormId=1 and
, _ FeedbackElementID = 1 group by Score;</textarea>');
68          $('#dataSourceAdditional').html("");
69          }
70          else if(this.value == 'csv'){
71              $('#dataSourceOptions').load('listOfFiles.php');

73          $('#dataSourceAdditional').html('Column X: <input
, _ type="text" id="selectedColX" size="2"/>'
74          ,_ id="selectedColY"
+ 'Column Y:

```

```

75         ,_ size="2"/><br>');
76     }
77     else
78         if(this.value == 'textfile'){
79             $('#dataSourceOptions').load('listOfFiles.php');
80             $('#dataSourceAdditional').html("");
81         }
82     });

```

SAMPLE CODE 32: dashboard5.php — JavaScript to update the *dataSourceOptions* div tag.

In Sample Code 33, on line 83, if a button with the *ID* set to *addNewCampElement* is clicked, the following code will run. On lines 84–87, the values from the input fields and select boxes are gathered. On line 91, the data source type that is selected is gathered.

As each different data source has a different combination of fields, a string titled *optionString* is added. This is used to hold the configuration for accessing the specified data source. For the data source of type CSV, on lines 96–99, the defined columns are gathered. These are set into the *optionsString* variable separated by commas. For the data type MySQL, on line 104 the *querybox* value is gathered and placed directly into the *optionsString*.

After the values have been gathered, on line 109, a *POST* is performed to the *ajax.php* page. The *type* of request being sent is set to *addElementToCamp*. Each of the attributes gathered is sent. When this function completes, an *alert()* statement with the content output of the *ajax.php* page is shown on line 116.

```

83     $("#addNewCampElement").click(function() {
84         var addElementCid = $("#addElementCid").val();
85         var addElementCol = $("#addElementCol").val();
86         var dataType = $("#addElementDataType").val();
87         var chartType = $("#chartType").val(); 88         var optionsString = ";
89
90         // get the data source type
91         var dataSourceType = $("#addElementSource :selected").val();
92
93         if(dataSourceType == 'csv'){
94             // Select CSV specific options

```

```

95         // filename
96         var selectedFile = $("#file:checked").val();
97         // column of interest
98         var selColX = $("#selectedColX").val();
99         var selColY = $("#selectedColY").val()
100         optionsString = selectedFile + ',' + selColX + ',' +
101             selColY;
102     }
103     else if(dataSourceType == 'mysql'){
104         var queryBoxContent = ($("#querybox").val()); 105         optionsString = queryBoxContent;
106     }
107
108
109         $.post( "ajax.php", { type:
110             "addElementToCamp",
111             cid:addElementCid,
112             customName:
113             addElementCol,
114             dataType:dataType,
115             dataSourceType:dataSourceType,
116             options:optionsString,
117             chart:chartType} )
118             .done(function( data ) {
119                 alert(data);
120             });
121     });

```

SAMPLE CODE 33: dashboard5.php — JavaScript click triggering the saving of a new campaign.

Example code: listOfFiles.php

To list available CSV files to the user, an AJAX call was made to a page titled *listOfFiles.php*. Inside this, file a reference to the *opendir()* function was used to open the *samplefiles* folder that was placed beside the PHP script. In reality, this folder can be placed anywhere as long as the path to the folder is correct. To find out what the current working directory is, a simple called to *echo getcwd()*; will print out the exact directory path aiding the debugging process.

While in the folder, a while loop is used to loop over each individual file. A simple *if statement* is added to ensure that the *entry* or a single file does not contain a full stop or two full stops that would indicate a parent folder. For each individual file, an *input* field is added with the

type set to *radio*. The *ID* of the field is set to *file*. The most essential element is the value that stores the internal reference to the file name. In this case, the value is set to be the value from the *entry* variable. The *entry* variable is printed again for the user to see, followed by a simple *br* tag to break onto the next line. After listing all the files, it is essential to close the connection to the directory using the *closedir()* function. In Sample Code 34, the complete code for *listofFiles.php* can be seen.

```

1          <?php
2          if ($handle = opendir('samplefiles')) {
3              while (false !== ($entry = readdir($handle))) {
4                  if ($entry != "." && $entry != "..") {
5                      echo '<input type="radio" id="file" name="file"
                        ,_ value="'. $entry .'">' . $entry . '<br>';
6                  }
7              }
8              closedir($handle);
9          }
10         ?>

```

SAMPLE CODE 34: *listOfFiles.php* — Retrieving a list of files from the *samplefiles* folder.

Example code: **ajax.php**

Sample Code 35 outlines the *ajax.php* page. On line 2, the *type* variable is first accessed and placed in a local variable. On line 3, if the variable is equal to *addElementToCamp*, on line 4 a call is made to the *addElementToCamp()* function. From lines 7–12 each variable in the *POST* is extracted and placed in local variables. On line 20 a new database connection is made. On line 23, the *INSERT* statement is created, with question marks for the values. On lines 28–33 each value is added to the statement using the *bindParam()* function. On line 34, the statement is finally executed.

```

1          <?php
2          $type = $_POST['type'];
3          if($type == 'addElementToCamp'){
4              addElementToCamp();
5          }
6          function addElementToCamp(){

```

```

7         $cid = $_POST['cid'];
8         $customName = $_POST['customName'];
9         $dataType = $_POST['dataType'];
10        $dataSourceType = $_POST['dataSourceType'];
11        $options = $_POST['options'];
12        $chart = $_POST['chart'];
13
14        try {
15            $host = 'localhost:3306';
16            $dbname = 'bookexamples';
17            $user = 'sampleuser';
18            $pass = '3C9038A509BE6145AAD6827B4568AD918BC3DAC0';
19
20            $DBH = new PDO("mysql:host=$host;dbname=$dbname",
21                $user, $pass);
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39 ?>

```

```

        $sql = "INSERT INTO
        `dashboard_camp_elements` (`campID`,
        `desc`, `datatype`, `datasource`,
        `options`, `chart`)
        VALUES (?,?,?,?,?,?);";
        $sth = $DBH->prepare($sql);
        $sth->bindParam(1, $cid,
        PDO::PARAM_STR);
        $sth->bindParam(2, $customName,
        PDO::PARAM_STR);
        $sth->bindParam(3, $dataType,
        PDO::PARAM_STR);
        $sth->bindParam(4, $dataSourceType,
        PDO::PARAM_STR);
        $sth->bindParam(5, $options,
        PDO::PARAM_STR);
        $sth->bindParam(6, $chart,
        PDO::PARAM_STR);
        $sth->execute();
        echo
        'New Element Added';
        } catch(PDOException $e) {echo 'failed' .
        $e;}
    }
}

```

SAMPLE CODE 35: ajax.php — Add elements to existing campaign.

Output

Figure 4.17 outlines the data source selection process. On the righthand side of this, the data source options become visible when either MySQL or CSV is selected. In this example, CSV has been selected, and a list of files is returned on the right of the modal. The campaign *ID* will then be entered, followed by a name for the unique element. The X column is entered to indicate the row of interest and Y is entered to indicate the column of interest from the selected CSV file. The data type for the column of interest can be set, along with the type of chart we wish to create.

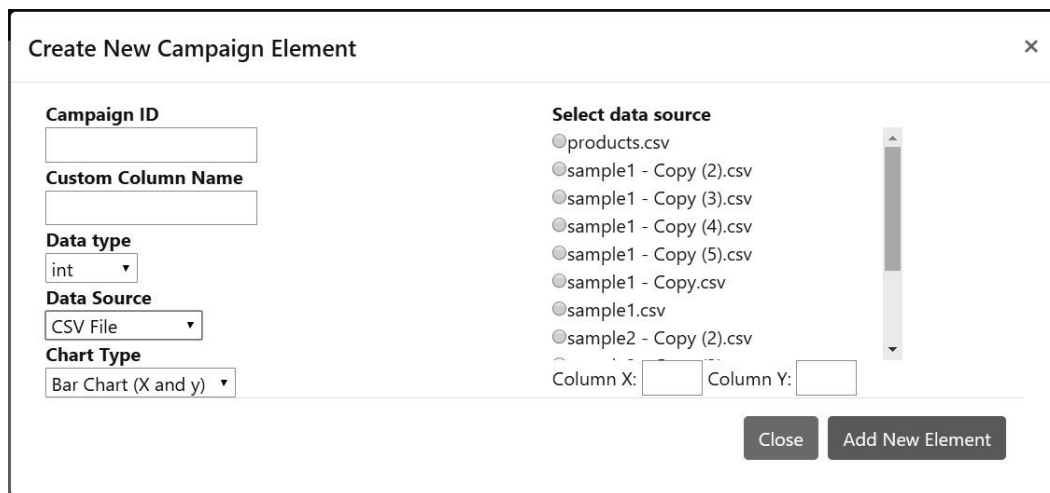
The image shows a modal window titled "Create New Campaign Element" with a close button in the top right corner. On the left side, there are several input fields: "Campaign ID" (a text box), "Custom Column Name" (a text box), "Data type" (a dropdown menu showing "int"), "Data Source" (a dropdown menu showing "CSV File"), and "Chart Type" (a dropdown menu showing "Bar Chart (X and y)"). On the right side, under the heading "Select data source", there is a list of radio buttons next to file names: "products.csv", "sample1 - Copy (2).csv", "sample1 - Copy (3).csv", "sample1 - Copy (4).csv", "sample1 - Copy (5).csv", "sample1 - Copy.csv", "sample1.csv", and "sample2 - Copy (2).csv". Below this list are two text boxes labeled "Column X:" and "Column Y:". At the bottom right of the modal are two buttons: "Close" and "Add New Element".

FIGURE 4.17: The process of adding a new campaign element and selecting the data source.

4.7 SUMMARY

This chapter described the process of creating a login page, register page, and dashboard. This dashboard served as the starting point for adding data visualizations in the form of charts. As data comes in various formats, the process of creating dynamic charts was explored. Although visualizations are useful for exploring data, additional processing is often required to gain a deeper insight into the data.

In Chapter 5, the process of working with text is explored, outlining methods of removing irrelevant terms and processing those of interest.

4 Working with An Text

INTRODUCTION

When gathering data from users, there are times when a predefined collection of answers does not suit the user's need to express themselves. When this happens, providing the user with a text field where they can enter their text feedback can be useful. However, text-based feedback comes with constraints as typical metrics for analysis, such as calculating the sum of a database column, can not be performed.

To gather insights from text, additional processing is required to identify the relevance and importance of the text or terms. This chapter will review the processing methods that can be performed on the text and the metrics for identifying value and relevance.

The area of natural-language processing (NLP) goes back to the early 1950s. As text is one of our primary forms of communication, software that utilizes text became a natural area of interest for researchers. Text in raw form often needs to be further processed to condense and extract meaningful insights.

In a business setting, analysts would like a user to be as specific as possible when describing what issues they are having. However, users often use the wrong keywords or terminology when describing problems. In addition to this, unlike numeric data, text-based data can come in

different forms. Examples of this include:

- a single word or a single sentence;
- large paragraphs of text out of context;

- unfavourable feedback that has no true reflection on the service or product.

Unlike other data formats, the punctuation and order in a sentence can impact the meaning. An example of this is seen in the following sentence: *I really hated how well the people at the front desk treated me. I am shocked!* When a simple analysis is performed on this sentence, for negative terms such as *hate*, the words surrounding this sentence *I really hated* may appear to be negative content directed toward the staff at the front desk. However, when the full context of the sentence is retained, we can see that it was a genuine compliment.

In each of the following sections, we will be working with text that may contain a single word, a single sentence, or a number of different sentences together. As this text has been entered by different users, no assumptions can be made about the format, content, or quality of the data. Some users may choose to use correct punctuation, while others may just create long sentences. These uncertainties are what add to the difficulty in processing text.

STOP WORDS

A single piece of text entered by a user contains two main varieties of terms: (1) terms that are useful to understand what the user is saying, or (2) terms that are used to join a sentence together.

Stop words are terms that are considered the glue in a sentence, example of stop words include: *as, the, it, on, which, and that*. Each of these terms does not describe anything particular to us. If these terms were removed from a piece of text, a person reading the text would still understand what the user is trying to say. Another example is the sentence *to be or not to be*, each term could be considered a stop word.

During analysis, these terms are often removed from the text before any further processing is performed. However, there is no official list of stop words. The length of the word does not necessarily mean that it is a stop word. However, stop words are frequently one and two characters long.

Issues often arise with stop words when the terms are essential. An

example of this is the band name *The Who*. When stop words are removed, e.g., the term *The*, we would be left with the band titled *Who*. This can equally be seen with the band name *Take That*, both terms could be considered stop words. This further reinforces the reason as to why no official lists of stop words exist.

The InnoDB^[9] storage engine for MySQL has a default collection of 36 different terms that it considers to be stop words. The Python NLTK library English stop word list contains 127 terms.^[10]

The great variety in terms and the requirement of application-specific stop words often causes the need to have our own collection of stop words. This collection of terms can be appended to, or have terms removed, as tasks change.

When looking at a collection of terms, we often need to consider what case our text is in. An example of this is when our stop word list contains the term *the*, but the word in our sentence contains the word *The* with a capital *T*, we often find the need to take our text and convert it into a lower case during text processing to avoid missing the term.

Another consideration that needs to be made during the text processing stage is where the processing should be performed. We have a choice to perform these actions on the server-side using a language such as PHP that places the burden on the server, or we can choose to use a client-side language such as JavaScript and place the burden on the user's web browser.

Another interesting element of feedback is the use of swear words inside of text. Often swear words do not add any great taste to data, but we often have a choice of removing these swear words from the text (for the sake of the analyst). As we all know, there are many common swear words, but often the list of swear words expands as people become more creative with methods or slurs to add to their text. Like stop words, there is no official list of swear words, but there are a number of different locations where lists of swear words can be obtained. Such as:

As swear words can often be portrayed in a positive or negative sense, it may or may not be relevant to the task to remove them from the text. If they do appear, they may be an attempt of a person to convey a severe

dissatisfaction with a service or experience.

Example 1: JavaScript stop word function

As stop words can be removed on both the client-side and the server-side, in this section we describe a JavaScript-based approach.

Example code: stopwords.html

In Sample Code 1, on line 3, a reference to the jQuery content delivery network (CDN) is first included. This will be used for selecting *div* tags on the page later. On line 7, a variable is created titled *sampleSentence* that contains a sentence with stop words included. On line 8, a new function is created titled *stripStopWords()*. This function takes a single variable titled *input*. This variable is in an unprocessed state.

Inside of this function on line 9, a placeholder variable is added that will be used as the location where the cleaned data will be placed as stop words are removed from the original *input*. On line 10, a JavaScript array is created that contains each of the MySQL stop words as individual elements of the array. After this on line 15, the *input* variable that was passed into the function is split based on the spaces between the words and placed into an array titled *termArray*.

Once we have each individual term inside of the *termArray*, we then begin the process of detecting if the stop words are present inside of the input. This is done by using a *for loop* to iterate over each term inside of the term array. For each iteration of the *for loop* on line 17, a term is pulled from the *termArray* at position *i*. This variable *i* references the current position in the array. The term is taken and placed inside of a temporary variable titled *currentTerm*.

On line 18, a simple *if statement* is used to check if the term we are currently processing can be found inside of the stop word list. This is done by using the *indexOf()* function that is part of the JavaScript array. If a positive number is returned, this identifies the word was present, if a negative is returned, that means the word was not present.

In the case that the term was not present, on line 20, the *currentTerm* is appended on to the *cleaned* variable. It is important to remember to

add a space before the current term is appended. Otherwise, all terms would appear together. At the very end of this function, on line 24, a simple *return* statement is added to return the cleaned string to the function caller.

On lines 27–32, at the bottom of the page, to provide an output in the web browser, two *div* tags are created, one that contains the *ID output1* and another with the *ID output2*. To test the function on line 33, first we will use the jQuery selector to select the *div* tag with the *ID output1* and append variable *sampleSentence* in its original form. After we have done this on line 35, we will then create a variable titled *cleaned*, that will be used to store the results after making a call to the *stripStopWords()* function that is passed the *sampleSentence* variable. Once the cleaned variable has a result, the output is appended to the second *div* tag with the *ID* set to *output2* on line 36.

```

1           <head>
2           <script
3           src="http://code.jquery.com/jquery-3.5.1.min.js"
4           crossorigin="anonymous"></script>
5           </head>
6           <script>
7           var sampleSentence = "what a great night I loved the food";
8           function stripStopWords(input){
9           var cleaned = "";
10          var stopwords = ["a","about","an","are","as","at","be","by",
11                          "com","de","en",
12                          "for","from","how","i","in","is","it","la","of","on","or",
13                          "that","the","this","to","was","what","when","where","who",
14                          "will","with","und","the","www"];
15          var termArray = input.split(" ");
16          for (i = 0; i < termArray.length; i++) {
17          var currentTerm = termArray[i];
18          if(stopwords.indexOf(currentTerm) >= 0){
19              } else {
20              cleaned += " " + currentTerm;
21              }
22
23          }
24          return cleaned;
25          }
26          </script>
27          <div id = "output1">
28          <b>Before Filter:</b>
29          </div>
30          <div id = "output2">

```

```
31         <b>After Filter:</b>
32     </div>
33     <script>
34         $('#output1').append(sampleSentence);
35     var cleaned = stripStopWords(sampleSentence);
```

```
36      $('#output2').append(cleaned);  
37      </script>
```

SAMPLE CODE 1: stopwords.html — JavaScript stop word function.

Output

Once this has been performed, inside of the web browser we can see the original string and the string after it has been processed shown in [Figure 5.1](#).

Before Filter: what a great night i loved the food

After Filter: great night loved food

FIGURE 5.1: Before and after stop-word removal with JavaScript.

In the *After Filter* result, we still retain the original descriptive terms that are core to the feedback, but the terms that do not add any value to the description have been removed. At any stage, if there were terms that we do not want to see as part of the output, we can simply add these custom stop words into the original stop word array.

Example 2: PHP stop-word function

In place of this client-side based filter, we can perform the same action on the server-side using PHP. As records are often retrieved from the database using PHP, it provides us with a good opportunity to perform this sanitization.

Example code: stopwords.php

In Sample Code 2, a new PHP page is created. On line 2, a new function is created titled *stripStopWords()*. This function takes a single parameter titled *input*, that contains the text to be processed.

On line 3, a variable titled *cleaned* is added as a place holder for the final cleaned string after processing. On line 4, a PHP array is created that contains each of the individual stop words we wish to remove if present.

On line 8, the string input is broken into array format using the *explode()* function that takes in two parameters. The first is the delimiter

used to break the string. As we are working with a string, this will be a single space. The second parameter is the *input* variable that has been sent to the function. This input is converted into lower case before the stop word detection process is started.

Two loops are used to begin processing. On line 9, the first loop that is placed on the outside and is used to iterate over each individual item inside the *inputArray*. Each term is placed into the *inputword* variable temporarily. A flag titled *isStopWord* is first set to *False* on line 10. As at the time of processing the term has not been identified as a stop word.

On line 11, a second internal loop is started that takes each individual term inside of the *stopwords* array and compares it to the *inputword* being processed. At any stage if the term is equal to a stop word then the *isStopWord* flag is set to *True* on line 13.

After the stop word loop has finished a simple if statement is used on line 16 to check if the *isStopWord* variable is set to *False*. If it still remains *False* then the input term is appended to the *cleaned* variable on line 17. After both loops have finished processing the *cleaned* string is returned back to the function caller on line 20.

To run this example a variable title *input* is created that contains a string on line 23. The input is printed to the web browser using an *echo* statement. On line 25, a call is made to the *stripStopWords()* function and the input variable is passed. The *result* variable is used to hold the cleaned string that is returned. On line 26, this string is printed to the console with the stop words removed.

```
1      <?php
2      function stripStopWords($input){
3          $cleaned = "";
4          $stopwords = array("a","about","an","are","as","at","be",
5              "by","com","de","en", "for","from","how","I","in","is","it",
6              "la","of","on","or", "that","the","this","to","was","what",
7              "when","where","who", "will","with","und","the","www");
8          $inputArray = explode(' ', strtolower($input));
9          foreach ($inputArray as $inputword) {
10             $isStopWord = False;
11             foreach ($stopwords as $word) { 12             if($inputword == $word){ 13                 $isStopWord = True;
14             }
15             }
16             if($isStopWord == False){
```

```

17             $cleaned .= ' ' . $inputword;
18         }
19     }
20     return $cleaned;
21 }
22
23 $input = 'I am really happy with the service';
24 echo 'Input: ' . $input . '<br>';
25 $result = stripStopWords($input);
26 echo 'Result: ' . $result;

```

SAMPLE CODE 2: stopwords.php — PHP stop word function.

Output

shows the input and output after stop word removal using PHP.

```

Input: I am really happy with the service
Result: am really happy service

```

stopwords.php — Before and after stop-word removal with PHP.

RED-FLAG MONITORING

Similar to stop words, red flags can be considered terms of interest in a piece of text. A red flag may be a term that is rarely mentioned, but when it is, it should be identified. This can be useful when working with larger collections of text as the size can make reading individual pieces of text a difficult process.

Example 3: Red-flag dashboard option

This example consists of four files and two libraries:

1. redflag_comments.sql — This is the create code for the database table that contains sample comments.
2. redflagterms.sql — This is the database table that contains the list of terms that are identified as red flags.
3. dashboard.php — The original dashboard with additional red flag modal.
4. redFlags.php — Additional file that is dynamically called using jQuery to render the selected records with highlighted red flag terms.
5. jQuery — This is used to select elements on the page and perform AJAX *POST* requests

6. jQueryUI — This is used to provide an interactive date picker.

A red flag may be a term that can be a cause of concern for a business. Before the process begins, we will create a sample collection of words that we consider red flags. The terms used in this example include *staff* and *breakfast*.

Example code: redflag comments.sql

To begin, we will create a simple table titled *redflags_comments* that will contain our test data as shown in Sample Code 3. This table contains a *ID* column that is unique, and a column titled *mytime* that has a record of when the comment was left. A *comment* column is added to store the text gathered from users during the feedback process. An additional column titled *cid* is added. This will contain the *ID* number for this individual campaign.

```
1          CREATE TABLE `redflag_comments` (  
2          `id` INT NOT NULL AUTO_INCREMENT,  
3          `mytime` DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,  
4          `comments` TEXT, 5          `cid` INT, 6          PRIMARY KEY (`id`)  
7);
```

SAMPLE CODE 3: *redflag comments* table SQL.

As each different campaign may have a different collection of terms that are considered red flags, a table titled *redflagterms* is created as shown in Sample Code 6. Each individual term is stored as a row in this table along with the the campaign *ID* in the *cid* column.

```
1          CREATE TABLE `redflagterms` (  
2          `id` INT NOT NULL AUTO_INCREMENT,  
3          `term` VARCHAR(50),  
4          `cid` INT, 5          PRIMARY KEY (`id`)  
6);
```

SAMPLE CODE 4: redflagterms table.

To provide a working example, this database table will be filled with sample data as shown

	timestamp	comments	cid
1	2018-08-05 10:24:29	The staff are horrible here, I am really not hap...	1
2	2019-08-05 10:24:51	The staff have been great, I am really not hap...	1
3	2019-08-05 10:25:09	I was five minutes late and they said i can not ...	1
4	2019-08-05 10:25:37	The breakfast was great.	1
5	2019-08-05 10:25:43	More coffee is needed at breakfast time. How ...	1
6	2019-08-05 21:09:23	breakfast was good but the staff are great	1

FIGURE 5.3: Sample comment records.

Example code: dashboard.php

Building on the dashboard that was described in [Chapter 4](#), additional HTML and JavaScript is added to include a new option to show red flags in text. In Sample Code 5, to present the options to the user a Bootstrap modal is created. On line 2, the *ID* of the modal is set to *selectRanges*. On line 15, a *div* tag is added with the *class* set to *modal-body*. This is the main location where content will be placed. In this tag, we will provide the user with two different jQueryUI date pickers that will allow the user to choose a date range for the analysis process. On line 16, 21 and 26, *div* tags are added to create individual rows. On line 17, the text *Start Date:* is added inside a *div* tag with the *class* set to *col*. On line 18 and 23, input fields are added that will be converted into date pickers using jQueryUI. A third field is added on line 28, to allow the user to enter in the *CID* of interest. A *Show Red Flags* button is added on line 35. The *onclick* attribute is set to call the *showRedFlags()* JavaScript function described later.

```

1          <!-- Red Flag Modal -->
2      <div class="modal fade" id="selectRanges" tabindex="-1"
3          role="dialog" aria-labelledby="selectRanges"
4          aria-hidden="true">
5      <div class="modal-dialog" role="document">
6          <div class="modal-content">
7              <div class="modal-header">
8      <h5 class="modal-title" id="addAtt">Select Date Ranges
9          </h5>
10         <button type="button" class="close" data-dismiss="modal"
11             aria-label="Close">
12         <span aria-hidden="true">&times;</span>
13         </button>
14         </div>
15         <div class="modal-body">
16         <div class="row">

```

```

17         <div class="col-3">Start Date:</div>
18         <div class="col"><input type="text" id=
19         "startDate"></div>
20     </div>
21     <div class="row">
22         <div class="col-3">End Date:</div>
23         <div class="col"><input type="text" id=
24         "endDate"></div>
25     </div>
26     <div class="row">
27         <div class="col-3">CID:</div>
28         <div class="col"><input type="text" id= 29     "selectedCID"></div>
29     </div>
30 </div>
31 </div>
32 <div class="modal-footer">
33     <button type="button" class="btn btn-secondary"
34     data-dismiss="modal">Close</button>
35     <button onclick="showRedFlags()" type="button"
36     class="btn btn-primary">Show Red Flags</button>
37 </div>
38 </div>
39 </div>
40 </div>
41 <!-- end red flag modal -->

```

SAMPLE CODE 5: dashboard.php — Select date ranges and campaign *ID* of interest.

In Sample Code 6, a short hand for the *document ready* function is added on line 43. On line 44, the jQuery selector is used to select the *startDate* input field. A call is made to the *datePicker()* jQueryUI function to convert the input field into a date picker. On line 45, the date format that is being used is set to *yy-mm-dd*. On line 47 and 48, the same process is repeated for the *endDate* input field.

On line 51, the *showRedFlags()* function is added. This is the main function that is called when the *Show Red Flags* button is clicked on the modal. In this function on line 52, the start date of interest is gathered followed by the end date on line 53. As there may be a number of campaigns running, on line 54 the unique *cid* of interest is extracted. On line 55, if the *redFlags div* tag is already on the dashboard it will be removed by using the jQuery selector to select the *redFlags div* tag followed by a call to the *remove()* function.

On line 56, the *output div* tag is selected using the jQuery selector.

This is the main *div* tag in the center of the dashboard. A call is made to the *load()* function that dynamically retrieves the contents of a file. In this example, the *redFlags.php* page is called that is responsible for rendering the output. The *cid*, *start*, and *end* variables are passed via the URL to this page. The output of this page will be gathered and placed into the *output div* tag.

```
42     <script>
43     $( function() {
44     $( "#startDate" ).datepicker({ 45     dateFormat: 'yy-mm-dd',
46         });
47     $( "#endDate" ).datepicker({ 48     dateFormat: 'yy-mm-dd',
49         });
50     } );
51     function showRedFlags(){
52     var start = $('#startDate').val();
53     var end = $('#endDate').val();
54     var cid = $('#selectedCID').val();
55     $('#redflags').remove();
56     $('#output').append($('

SAMPLE CODE 6: dashboard.php — JavaScript to convert input field into date pickers and showRedFlags() function.


```

Output

In the dashboard, when the item called *Text Analysis* is clicked a menu titled *View Red Flags* becomes available, as shown in [Figure 5.4](#). In [Figure 5.5](#) the layout of the modal can be seen. When the user clicks on either of the input boxes, the date picker will then become visible, as shown in [Figure 5.6](#). The values they select will be populated into the *input* fields.

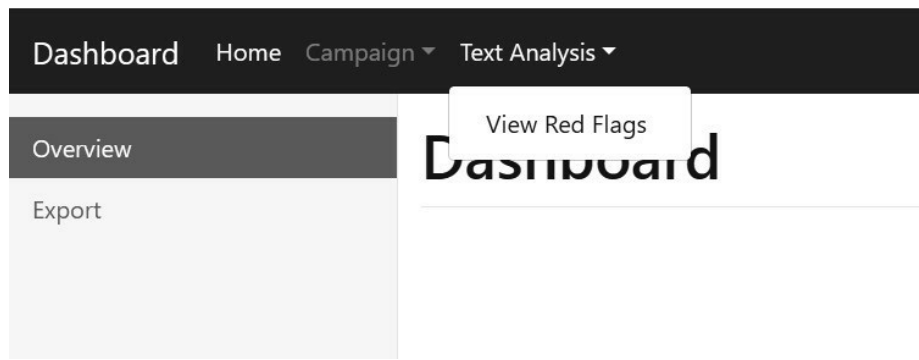


FIGURE 5.4: View red-flags menu option.

A screenshot of a modal window titled 'Select Date Ranges' with a close button (X) in the top right corner. The form contains three input fields: 'Start Date:', 'End Date:', and 'CID:'. At the bottom right of the modal are two buttons: 'Close' and 'Show Red Flags'.

FIGURE 5.5: Select date and CID modal.

Example code: redFlags.php

The core of this process is held in the *redFlags.php* page. In Sample Code 7, at the top of this page on lines 2–4, the *start* date, *end* date, and *cid* are gathered from the *GET* request. A new *div* tag is output to the page on line 5. The *start* date, *end* date, and *CID* are echoed to the page on line 6. A call is made to the *getData()* function on line 9, that is responsible for gathering the data.

The *getData()* function is defined on line 12. Inside this function, on line 19, a connection is made to the database. On line 21, the SQL

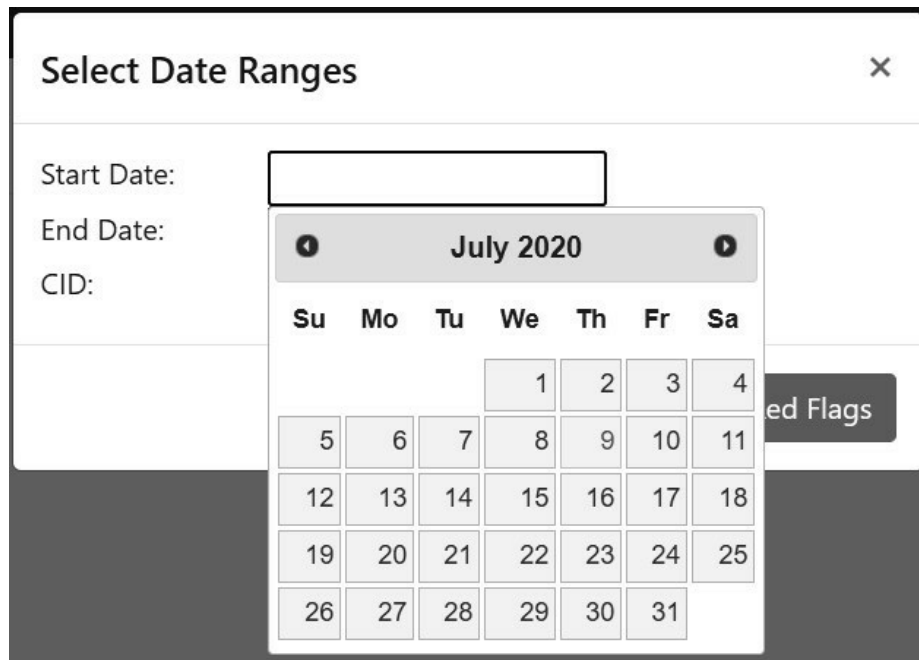


FIGURE 5.6: jQuery datepicker.

SELECT statement to the *redflag comments* table is added. The comments are narrowed down based on the date stamp being between the two dates specified by the user. The *cid* is used for the final aggregation of records for a single specified campaign.

A call is made to the *getRedFlags()* function on line 31 to get the list of specific red flags for the individual campaign. For each individual comment returned from the database, the comment is split using the *explode()* function on line 34 based on the space between each term in the comment. Each individual term is put into lower case.

On line 39, a PHP variable titled *output* is added that will be appended to as each comment is iterated through. On line 37, a single *inputWord* from the comment is compared against the *redFlags* array. This is done by using the PHP *in_array()* function that checks to see if a given term exists in the provided array.

On line 39, If the term currently under analysis exists inside of the *redFlags* array, then the color of the term is set to *red* using a small amount of CSS styling. If the term is not, it is simply appended to the string. The final result is printed to the browser using a PHP *echo*.

```
1 <?php
2 $start = $_GET['start'];
```

```

3  $end = $_GET['end']; 4 $cid = $_GET['cid'];
5  echo '<div id="redflags">';
6  echo '<b>Start:</b> ' . $start . ' - <b>End date:</b> ' . $end.'
7      - <b>CID:</b> ' . $cid.'<br>';
8      echo '<hr>';
9      getData();
10 echo '</div>';
11
12      function getData(){
13      global $start,$end, $cid;
14      try {
15          $host = 'localhost:3306';
16          $dbname = 'test';
17          $user = 'root';
18          $pass = '3C9038A509BE6145AAD6827B4568AD918BC3DAC0';
19          $DBH = new PDO("mysql:host=$host;dbname=$dbname", $user,
20          $pass);
21          $sql = "SELECT * FROM redflag_comments WHERE mytime
22          BETWEEN ?
23          AND ? AND cid = ?";
24          $sth = $DBH->prepare($sql);
25          $sth->bindParam(1, $start, PDO::PARAM_STR);
26          $sth->bindParam(2, $end, PDO::PARAM_STR);
27          $sth->bindParam(3, $cid, PDO::PARAM_INT);
28          $sth->execute();
29          $res = $sth->fetchAll();
30          // Get the list of red flags from the database
31          $redFlags = getRedFlags($cid);
32          foreach ($res as $row){ 33          $comment = $row[2];
34              $inputArray = explode(' ', strtolower($comment));
35              $output = "";
36              foreach ($inputArray as $inputWord) {
37                  $isFlag = in_array($inputWord, $redFlags);
38                  if($isFlag){
39                      $output .= '<span style="color:red"> ' . 40          $inputWord . '</span> ';
41                      } else {
42                          $output .= ' ' . $inputWord;
43                      }
44                      }
45          echo $output . '<br>';
46          }
47          } catch(PDOException $e) {echo $e;}
48          }

```

SAMPLE CODE 7: redFlags.php — Identifying red flag terms — part 1.

In Sample Code 8, to get a list of unique red flags for each campaign, a custom *getRedFlags()* function is created that takes a single parameter title *cid*. At the top of this function, an array titled *flags* is added on line 45. A connection to the database is performed on line 51.

On line 53, a *SELECT* statement is created to select data from the *redflagterms* table. The results are narrowed by returning those only with the *cid* equal to the one provided to the function. For each term, on line 59, they are appended to the *flags* array that is returned on line 63 and used as the source of red flags during the analysis of each comment.

```
44         function getRedFlags($cid){
45             $flags = array();
46             try {
47                 $host = 'localhost:3306';
48                 $dbname = 'test';
49                 $user = 'root';
50                 $pass = '3C9038A509BE6145AAD6827B4568AD918BC3DAC0';
51                 $DBH = new PDO("mysql:host=$host;dbname=$dbname", $user,
52                 $pass);
53                 $sql = "SELECT term FROM redflagterms WHERE cid = ?";
54                 $sth = $DBH->prepare($sql);
55                 $sth->bindParam(1, $cid, PDO::PARAM_STR);
56                 $sth->execute();
57                 $res = $sth->fetchAll();
58                 foreach ($res as $row){
59                     array_push($flags, $row[0]);
60                 }
61             } catch(PDOException $e) {echo $e;}
62
63         return $flags;
64     }
```

SAMPLE CODE 8: redFlags.php — Retrieving a lit of red-flag terms — part 2.

Dashboard

Start: 2018-01-08 - **End date:** 2020-01-01 - **CID:** 1

the staff are horrible here, i am really not happy.
the staff have been great, i am really not happy about the pool.
i was five minutes late and they said i can not have breakfast
the breakfast was great.
more coffee is needed at breakfast time. how can i work without it?
breakfast was good but the staff are great

FIGURE 5.7: Each comment is output with the red flag terms highlighted.

Output

a list of comments that were stored in the database can be seen. Each red flag term is highlighted in red. The *Start date* and *End date* selected by the user are also shown along with the campaign *ID* shown as *CID*.

N-GRAMS

When working with text-based data, a common practice that is performed is the process of creating *n-grams*. Before this process is introduced, we will first begin with an overview of the structure of a document. A document D consists of p individual paragraphs. Each paragraph can contain s individual sentences, each of which contains w words. Each individual sentence, contains words from $w_0...w_y$.

Although documents can be viewed as a collection of independent terms, the context around these terms can provide valuable insight into how the term was used. When considering the sentence *I was shocked, the service was great!*, if term frequency was performed on this sentence,

TABLE 5.1: Sample n-grams with $N=2$

the staff staff are
are horrible
horrible here here i i
am
am really
really not
not happy

TABLE 5.2: Sample n-grams with $N=3$

the staff are
staff are horrible are
horrible here

horrible here i here i am
i am really
am really not
really not happy

the weight for each term would be one as each term was present only once. If these terms were taken without context, the term *shocked* would be seen as a negative term as no additional context is provided.

As a solution to this, n-grams can be generated. N-grams take a sentence, s , as input that contains terms $w_0...w_y$. The n-gram algorithm takes a parameter, N , identifying how many terms should be included in an individual gram sequence. During the n-gram process, different values for N can produce vastly different results. Bi-grams are when the N is set to 2. For the sentence *the staff are horrible here I am really not happy* shows the output when $N = 2$.

Tri-grams is when $N = 3$. Unlike the previous examples, tri-grams include a large body of context. shows the output when $N = 3$.

Example 4: Generating n-grams

The advantage of using n-grams during the data analysis process is the additional context around a term is retained. This is especially useful when the number of records exceeds that of a manual review process.

Example code [ngram.php](#)

Sample Code 9 describes a simple n-gram approach. The *sentence* variable on line 2 holds the sentence to be processed. To break the sentence into individual tokens, the *explode()* function is used on line 3. This function takes two quotation marks with a single space between them. This is the *delimiter* and is used as the token in which the sentence split on. The second variable that is passed is the *sentence* variable containing the text.

As we will be processing any number of grams, a variable titled *gram* is added on line 4. In the case of $N = 2$ for the sentence *This is a sample sentence* the *gram* variable will first contain the term *The*

followed by a space and then the second term *staff* will be added. As the number to grams added will be equal to 2, the gram will be printed, and the variable will be cleared, allowing it to be reused for the next iteration.

This piece of code contains two loops, the first on line 6 loop will iterate over each individual term, while on a single term, y number of grams will be added using the loop on line 8. A small internal variable titled *counter* on line 9 is used to keep track of the number of grams that have been added. When the counter is equal to the desired number of tokens, the *gram* variable is printed to the browser, followed by a *br* tag to break onto the next line. This variable is cleared, and then the loop is broken using the *break* statement on line 14. This will then allow the outside loop to progress on to the next term. This piece of code can be used for any number of gram sequences by modifying the *n* variable.

```

1          <?php
2          $sentence = 'The staff are horrible here';
3          $tokens = explode(' ', $sentence);
4          $gram = '';
5          $n = 2;
6          for ($x = 0; $x <= sizeof($tokens) - 1; $x++) {
7              $counter = 0;
8              for ($y = $x; $y <= sizeof($tokens) - 1; $y++) {
9                  $counter++;
10                 $gram .= ' '. $tokens[$y];
11             if($counter == $n){
12                 echo $gram . '<br>'; 13         $gram='';
14                 break;
15             }
16         }
17     }

```

SAMPLE CODE 9: ngram.php — PHP N-gram algorithm.

Output

shows the n-gram output where $N=2$ for the sentence *The staff are horrible here*.

```

The staff
staff are
are horrible
horrible here

```

Sample n-gram output with $N=2$.

SENTENCE COMPLETION

When working with large quantities of user comments, often an overlap can be seen in specific words that are used together. An example of this can be seen with the word *staff* in the following comments:

1. The staff are great.
2. I had fun. The staff made the experience even better.
3. Good food, but the staff were nasty.

All three comments are outlining how the customer felt about the staff, however, the last comment was negative. This can be beneficial during analysis as comments about one topic, e.g., the staff, may prompt the start of an investigation to see why that one particular piece of negative feedback was left.

The issue that is presented however, is how to view this data. The small piece of information about the staff may be hidden in a larger paragraph or body of the text. The use of sentence completion can greatly enhance how text is analyzed, trimming the irrelevant text before the terms of interest occur, leaving only the relevant text. This is done by first starting with a single term, or set of terms that are considered the *stem* of the sentence.

Example 5: Sentence completion

This example contains two files:

1. create.sql — This file is used to create the *feedback_wild* database table.
2. completion.php — This file provides an example of sentence completion being used.

Example code: **create.sql**

To begin, Sample Code 10 starts with a simple SQL table titled *feedback_wild*. This table contains a column titled *feedback* that is used to hold sample records for the sentence completion process.

```
1 CREATE TABLE IF NOT EXISTS `feedback_wild` (  
2   `id` INT NOT NULL AUTO_INCREMENT,
```

```

3      `mytime` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
4      `feedback` text,
5      PRIMARY KEY (`id`)
6  );

```

SAMPLE CODE 10: feedback wild database table.

Example code: completion.php

In Sample Code 11, a connection to the database is first made on line 7. A variable titled *rootWords* is added on line 9. This variable contains the term of interest. In this example, we are searching for the word *staff*. On line 10, a fragment of SQL is created. A percentage sign is added to the start and end of the *rootWords* variable and placed into a new variable titled *likeString*. The percentage sign indicates that the term can be found anywhere inside of a feedback record, e.g., start with, end with or in the middle of a sentence.

On line 11, a SQL query is created with a question mark for the value of the LIKE. An unnamed placeholder is used, shown as a question mark in the query. During the call to the *execute()* function on line 13, the prepared *rootWords* variable is passed in the form of the *likeString*.

Once the query has been executed, on line 14, a *while* loop is used to iterate over the records that have been identified. On line 15, the text data stored in the *feedback* column is placed into a local variable titled *record*. After this on line 16, the *strpos()* function is used to identify the index of where the first occurrence of the term *staff* was found.

On line 17, a call is made to the *substr()* function. This function takes three parameters. The first is the variable that contains the text we wish to process. The second is the starting index we wish to begin processing from. Finally, the third parameter contains the index number where we wish to stop at. In this example, this index is set to be the length of the *record* variable that is found using the *strlen()* function. The result of the *substr()* is printed to the browser using a simple *echo* statement.

```

1      <?php
2      try {
3          $host = '127.0.0.1:3306';
4          $dbname = 'test';
5          $user = 'root';
6          $pass = '3C9038A509BE6145AAD6827B4568AD918BC3DAC0';

```

```

7          $DBH = new PDO("mysql:host=$host;dbname=$dbname", $user,
8          $pass);
9          $rootWords = 'staff';
10         $likeString = '%' . $rootWords . '%';
11         $stmt = $DBH->prepare("SELECT * FROM feedback_wild WHERE
12         feedback LIKE ?");
13         $stmt->execute([$likeString]);
14         while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
15             $record = $row['feedback'];
16             $start = strpos($record, "staff");
17         echo substr($record, $start, strlen($record)).'<br>';
18         }
19         } catch(PDOException $e) {echo $e;}

```

SAMPLE CODE 11: completion.php — Sentence completion for the term *staff*.

Output

shows an example of the term *staff* being found at the start and the middle of sentences. All text that was before the term of interest is disregarded, leaving only the term of interest followed by the rest of the sentence.

```

staff are horrible here, I am really not happy.
staff have been great, I am really not happy about the pool.
staff are great

```

FIGURE 5.9: Result of searching for the word *staff* inside a feedback records.

WORD STEMMING

When processing terms, often they can have a different suffix depending on how the word is being used. For the term *house*, if this is used in a sentence, it is common to see variations such as *Houses*, *House*, and *Housing*. The issue with this, however, is if any calculations are being performed, e.g., counting the number of times the word was found, often the other variations of the term will be missed.

Example 6: Word stemming

To prevent this from happening, when working with collections of text *stemming* can be performed. This is the process of reducing the term

back to the word stem. Many different approaches can be taken when stemming. One example of a stemming tool is the Porter Stemmer. In this example, we will be working with a PHP implementation of this algorithm.^[11]

This example contains two files:

1. stemming.php — A sample instance of the Porter stemming being used.
2. PorterStemmer — An instance of the stemming algorithm.

Example code: stemming.php

In Sample Code 12, on line 2, the library is first added into the current page. The word to be stemmed is placed inside the *word* variable on line 3. This variable is passed to the *Stem* function on line 4. This function returns the word stem that is printed to the page. The stemming process is completed for the terms *Houses*, *House*, and *Housing*. HTML *br* tags are used to break the output onto individual lines.

```
1  <?php
2  require('PorterStemmer.php');
3  $word = 'Houses';
4  $stem = PorterStemmer::Stem($word);
5  echo $stem . '<br>';
6  $word = 'House';
7  $stem = PorterStemmer::Stem($word);
8  echo $stem . '<br>';
9  $word = 'Housing';
10 $stem = PorterStemmer::Stem($word);
11 echo $stem . '<br>';
```

SAMPLE CODE 12: stemming.php — Using the Porter Stemmer on three terms.

Output

shows the output of the stemming process. Although three different words were passed to the stemming algorithm, the same stem is returned after each call. This approach helps us identify that the three terms are all related.

Hous
Hous
Hous

Result after calling the stemming algorithm.

SYNONYMS

When reviewing feedback records, it can become apparent quickly that two reviews never contain a similar collection of phrases or terms. This can cause difficulty for any feedback processing that requires a match to be made between terms. An example of this can be seen when the term *chef* is used in one record and the term *cook* in another record. By developing a function to identify these synonyms, another dimension is added into the analysis process.

Example 7: BigHugeLabs.com API

In this example, we will outline the process of using the BigHugeLabs.com API for synonyms of a given term. The first step in this process is to get an API key to access the service. They currently offer 500 requests per day for free. Additional calls to the service can be made for a modest fee. To get this key, an account is first created. describes the parameters needed to sign up for a key. A brief description of the application is added, followed by the URL where the requests will be coming from. As this is a test application, *localhost* was added for the URL.

Get an API Key

Describe your application.

Please be as detailed as possible. If you are a user requesting a key for an app, please mention the app.

Website/app store URL

BigHugeLabs.com API key creation process.

After the key creation has been completed, sample REST URLs are shown. Each of these passes the unique API key to the service along with a single word of interest. The difference in each API URL is the format in which the response is returned shows the options available including plain text, XML or JSON.

Output

In Sample Code 13, the API URL is constructed. The first parameter that is passed is the unique API access key. The term *dad* is passed to the API as the term of interest. To aid the processing of the result */json* is appended to the end of the URL to retrieve the result in JSON format.

Sample REST URLs	
Plain text	<code>https://words.bighugelabs.com/api/2/0c7cc83b387027012538b959c8576cfb/word/</code>
XML	<code>https://words.bighugelabs.com/api/2/0c7cc83b387027012538b959c8576cfb/word/xml</code>
JSON	<code>https://words.bighugelabs.com/api/2/0c7cc83b387027012538b959c8576cfb/word/json</code>
Serialized PHP	<code>https://words.bighugelabs.com/api/2/0c7cc83b387027012538b959c8576cfb/word/php</code>

BigHugeLabs.com sample REST URLs.

SAMPLE CODE 13: Complete API call being passed the term *dad*.

Sample Code 14 shows the result set containing the synonyms for the term *dad*.

```
1      {"noun":{
2      "syn":[
3      "dada",
4      "daddy",
5      "pa",
6      "papa",
7      "pappa",
8      "pop",
9      "begetter",
10     "father",
11     "male parent"
12     ]
13     } 14 }
```

SAMPLE CODE 14: Synonyms returned for the term *dad*.

When calculating the number of feedback records that contain the term *dad*, additional records can be included if any of the synonyms are found. Although a useful tool during analysis, some issues do exist. An example of this is the synonym for dad titled *pop*. As context is often removed during the processing of text, it can be difficult to identify if this is a reference to a person's *dad* or to the synonym for *soda* which is *pop*.

Example 8: PHP API request to Big Huge Lab's API

To provide access to the REST API from PHP we can use the *file_get_contents()* function. This allows us to pass the URL of the API that contains a single variable titled *term* that contains the term of interest.

Example code: [syn.php](#)

In Sample Code 15 on line 2, the term of interest is first defined. In this example, we are searching for synonyms of the term *dad*. On line 3, a call is made to the PHP *file_get_contents()* function. This function will make a *GET* request to the provided URL. The value of the variable *term* is substituted into the string. The contents of the page that is output is placed inside a local variable titled *result*.

The service will return JSON content. On line 5, to process the JSON, the result is passed to the *json_decode()* function. The *true* parameter is passed to ensure the result is returned as an associative array. On line 6, after the *decoded* variable is filled, we can pass a key to select the results from the array. The *noun* is the parent result. Inside this, the *syn* collection can be accessed.

Once we have accessed the *syn* results, a *foreach* statement is used to loop over each individual synonym on line 7. A simple *br* tag is added to break each result on to a new line when viewing the results in a web browser.

```
1          <?php
2          $term = 'dad';
3          $result = file_get_contents("https://words.bighugelabs.com/api/2/"
4          . "0c7cc83b387027012538b959c8576cfb/$term/json");
5          $decoded = json_decode($result, true);
6          $syn = $decoded['noun']['syn'];
7          foreach($syn as $item){
8          echo $item . '<br>';
9          }
```

SAMPLE CODE 15: syn.php — Calling the BigHugeLabs API using PHP.

Output

Figure 5.13 shows the output after passing the term *dad* to the Big Huge Lab’s synonym API.

```
dada
daddy
pa
papa
pappa
pop
begetter
father
male parent
```

Example synonyms returned from the Big Huge Labs API called via a *GET* request.

TERM WEIGHTING

In previous examples, text was gathered from the database and different tasks were performed directly on the text without any additional processing. One issue that can be seen with this approach is the scalability of the solutions. When working with a small number of records, the process of searching through the text can easily be performed and insights can be gathered. However, when hundreds or thousands of records are under analysis, any application of the data quickly becomes difficult as irrelevant text may become prominent in the results. For this reason, additional metrics to identify the importance of text can be applied.

Example 9: Term frequency

One simple approach that can be used to aid the analysis process is focused on applying weights to the individual terms. The most simplistic approach to this is to calculate the number of times each individual word has been found across all records. In this example, we will consider five individual records as shown in

TABLE 5.3: Sample Records for Term Frequency

ID Feedback

1	~~I was very happy with the product that I got in the mail.~~

However, the package was handled badly.

2 I am very happy with the product that I got. I will come back again in the future.

3 There was a number of issues with the customer service of this company. I will take my business somewhere else. It was a bad experience.

4 I will never return. I was treated badly by the man on the front desk.

5 This is a great company, I will never go elsewhere.

Example code: **tf.php**

To calculate the term frequency for each word, a total count for each term across all records is performed.

In Sample Code 16, on lines 2–6 individual records are added as strings. On line 8, an array is created that will be used as the main storage place for the strings. On line 9–13, each individual string is added to the array. On line 14, a new array is created that will be used to hold all the individual terms. On line 15, a simple *foreach* statement is used to iterate over all the records. For each individual record, on line 17, the full stops and commas are removed from the string and replaced with spaces. On line 18, the *explode()* function is called to break the individual string into single terms every time a space is found.

On line 20, each individual term is added to the *allWords* array. Finally on line 24, a call is made to the *array count values()* function that will count the total occurrences of each term in the array. This output is printed to the browser using the *print r()* function.

```
1 <?php
2 $doc1 = 'I was very happy with the product that I got in the , _ mail. However, the package was handled badly.';
3 $doc2 = 'I am very happy with the product that I got. I will
  , _ come back again in the future.';
4 $doc3 = 'There was a number of issues with the customer service , _ of this company. I will take my business
  somewhere else. It , _ was a bad experience.';
5 $doc4 = 'I will never return. I was treated badly by the man on , _ the front desk.';
6 $doc5 = 'This is a great company, I will never go elsewhere.';
7
8 $documents = array();
9 array_push($documents, $doc1); 10 array_push($documents, $doc2); 11
array_push($documents, $doc3); 12 array_push($documents, $doc4);
13 array_push($documents, $doc5);
```

```

14         $allWords = array();
15         foreach($documents as $doc){
16             echo '-----document-----<br>';
17             $cleaned = strtr($doc, array(' ' => " ", ' ' => " "));
18             $terms = explode(' ', $cleaned);
19             foreach($terms as $term){
20                 array_push($allWords, $term);
21
22             }
23         }
24         print_r(array_count_values($allWords));

```

SAMPLE CODE 16: tf.php — Sample term frequency calculation with PHP

Output

In this example, the term *badly* was seen once in record 1 and once in record 4. The term frequency for the term *badly*, therefore, would be set to two.

When processing these records, if the word *I* was counted, a value of 9 would be assigned. The issue with this can easily be seen, as very little insight can be gathered from this word. The raw term frequency can be considered naive for the purpose of calculating importance. In a list of all the term frequency scores for words greater than one can be seen. Excluding the stop words that can be seen at the top of this table, interesting terms such as *badly*, *never*, and *happy* can be seen.

Example 10: Term frequency — Inverse Document Frequency

A modification of the raw term frequency is the Term Frequency — Inverse Document Frequency (TF-IDF). This term weighting scheme is focused on the assumption that terms that frequently appear across all documents should be given a lower score, and terms that are unique and

: Sample Term Frequency Scores

8	the
5	was
4	will
3	with

Count Term

9 i

3	a	not seen as often should be given a higher score.
2	very	This weighting scheme could be considered a direct
2	this	solution to the issue of stop words that was seen
2	that	with traditional term frequency calculations in the
2	product	previous section. describes two sample records for
2	of	analysis.
2	never	: Sample Records for Term Frequency—Inverse
2	in	Document Frequency
2	happy	
2	got	
2	company	
2	badly	

ID Feedback

- 1 ~~I was very happy with the product that I got in the mail.~~
However, the package was handled badly.
- 2 I am very happy with the product that I got. I will come back again in the future.

The TF-IDF score for the term *mail* is calculated by first calculating the term frequency (tf) followed by the inverse document frequency (idf). *Mail* occurred once out of the 19 terms in document 1. The first piece of the equation is shown in Equation 5.1.

$$tf(mail, d1) = \frac{1}{19} = 0.05 \quad (5.1)$$

In document 2, the term frequency for *mail* is 0 out of the 18 words in the document as shown in Equation 5.2.

$$tf(mail, d2) = \frac{0}{18} = 0.0 \quad (5.2)$$

The inverse document frequency for the term *mail* is then calculated that takes into consideration the number of times the word appeared across all documents, *D*. In this example, out of the two documents, one document contained the term *mail*. The result is converted to logarithm base 10 as shown in Equation 5.3.

$$idf(mail, D) = \log\left(\frac{2}{1}\right) = 0.301 \quad (5.3)$$

Once the term frequency has been calculated, and the inverse document frequency is calculated, both components of the equation can be brought together to form the TF-IDF score. For document 1, the TFIDF is calculated by multiplying the *tf* score by the *idf* score for the term *mail* as shown in Equation 5.4.

$tfidf(mail, d1, d) = 0.05 \times 0.301 = 0.0158$ (5.4) The TF-IDF for document 2 can then be calculated as shown in Equation 5.5.

$$tfidf(mail, d2, d) = 0.0 \times 0.301 = 0 \quad (5.5)$$

The resulting TF-IDF score is higher for less common words, providing a deeper insight into the terms that have been used. This approach naturally removes common stop words from the results as they often appear in the documents.

Example code: [tfidf.php](#)

Example 17 outlines a sample PHP TF-IDF algorithm. At the top of this example on lines 2 and 3, two simple feedback records are added with their original punctuation. On line 5, an array is created titled *documents* that is used to hold all documents that will be under consideration during the TF-IDF calculation process. On lines 6 and 7, the *array push()* function is used to add the two documents onto the array.

To begin the process, a simple *foreach* statement is added on line 9 that is used to iterate over each document. On line 10, a simple line is added with a *br* tag appended to identify the document we are currently processing in the web browser. On line 11, the *strtr()* function is used to replace any occurrences of a full stop or comma with a space and place the result into the *cleaned* variable. On line 12, the *cleaned* is passed into the *explode()* function that takes the delimiter to break the string on. In this case, a space is passed, followed by a reference to the *cleaned* variable. This will leave a collection of individual terms inside the *terms* array.

On line 13, a simple call to the *count()* function is performed to find out how long the array is, e.g., how many terms we are working with. A

second internal *foreach* statement is started to iterate over each individual term in the document. The *substr_count()* function is used to identify how many times the term appears in the document. The term under consideration, the *length* and the *count* are printed to the browser to aid debugging. The *tf* is calculated by taking the *count* and dividing it by the *length*.

On line 21, the *idf* is calculated by taking the log of the number of documents divided by the number of documents that contain the term under analysis. 10 is passed to the log function to return the answer in logarithm base 10. Finally, on line 23 the final TF-IDF score is generated by taking the *tf* value and multiplying it by the *idf* value. On line 24, a simple *echo* is used to print the score to the browser.

To aid the algorithm, on line 27, the body of the *countInDocs()* function can be seen. This function takes a single term and counts the number of documents that contain the term. At the end of this function, a *return* statement is added on line 35 to return the count.

```

1  <?php
2  $doc1 = 'I was very happy with the product that I got in the ,_ mail. However, the package was handled badly.';
3  $doc2 = 'I am very happy with the product that I got. I will
   ,_ come back again in the future.';
4
5  $documents = array();
6  array_push($documents, $doc1);
7  array_push($documents, $doc2);
8
9      foreach($documents as $doc){
10         echo '-----document-----<br>';
11         $cleaned = strtr($doc, array(' ' => " ", ' ' => " "));
12         $terms = explode(' ', $cleaned);
13         $len = count($terms);
14         foreach($terms as $term){
15             $count = substr_count($doc, $term);
16             echo ' term: ' . $term;
17         echo ' count: ' . $count;
18
19         echo ' length: ' . $len; 19         $tf = $count / $len;
20         echo ' - TF: ' . $tf;
21         $idf = log((count($documents) / countInDocs($term)),
   ,_ 10);
22         echo ' IDF: ' . $idf;
23         $score = $tf * $idf;
24         echo ' - TF-IDF: ' . $score . '<br>';
25     }
26 }
```

```

27         function countInDocs($term){
28         global $documents; 29         $count = 0;
30                 foreach($documents as $doc){
31                 if(substr_count($doc,$term) > 0){
32                         $count = $count + 1;
33                 }
34                 }
35                 return $count;
36                 }

```

SAMPLE CODE 17: tfidf.php — Sample PHP TF-IDF algorithm.

Output

shows the output for the first document during analysis by the TF-IDF algorithm. For debugging, echo statements were added to all calculations. In this output, we can see that the common terms that were in both documents provided a lower TF-IDF score. Unique terms have higher TF-IDF scores.

SENTIMENT ANALYSIS

When working with text, an additional opportunity is provided to gather the overall *sentiment* of the text. The sentiment of a piece of text is the categorization of a record into a defined category of *positive*, *negative*, or *neutral* sentiment. Traditional statistic metrics such as ranges between 1–10 easily allow us to see if a user was satisfied or not with a service. Working with text, however, is a little bit more difficult as the terms in a

```

-----document-----
term: I count: 2 length: 19 - TF: 0.10526315789474 IDF 0 - TF-IDF: 0
term: was count: 2 length: 19 - TF: 0.10526315789474 IDF 0.30102999566398 - TF-IDF: 0.03168736796463
term: very count: 1 length: 19 - TF: 0.052631578947368 IDF 0 - TF-IDF: 0
term: happy count: 1 length: 19 - TF: 0.052631578947368 IDF 0 - TF-IDF: 0
term: with count: 1 length: 19 - TF: 0.052631578947368 IDF 0 - TF-IDF: 0
term: the count: 3 length: 19 - TF: 0.15789473684211 IDF 0 - TF-IDF: 0
term: product count: 1 length: 19 - TF: 0.052631578947368 IDF 0 - TF-IDF: 0
term: that count: 1 length: 19 - TF: 0.052631578947368 IDF 0 - TF-IDF: 0
term: I count: 2 length: 19 - TF: 0.10526315789474 IDF 0 - TF-IDF: 0
term: got count: 1 length: 19 - TF: 0.052631578947368 IDF 0 - TF-IDF: 0
term: in count: 1 length: 19 - TF: 0.052631578947368 IDF 0 - TF-IDF: 0
term: the count: 3 length: 19 - TF: 0.15789473684211 IDF 0 - TF-IDF: 0
term: mail count: 1 length: 19 - TF: 0.052631578947368 IDF 0.30102999566398 - TF-IDF: 0.015843683982315
term: However count: 1 length: 19 - TF: 0.052631578947368 IDF 0.30102999566398 - TF-IDF: 0.015843683982315
term: the count: 3 length: 19 - TF: 0.15789473684211 IDF 0 - TF-IDF: 0
term: package count: 1 length: 19 - TF: 0.052631578947368 IDF 0.30102999566398 - TF-IDF: 0.015843683982315
term: was count: 2 length: 19 - TF: 0.10526315789474 IDF 0.30102999566398 - TF-IDF: 0.03168736796463
term: handled count: 1 length: 19 - TF: 0.052631578947368 IDF 0.30102999566398 - TF-IDF: 0.015843683982315
term: badly count: 1 length: 19 - TF: 0.052631578947368 IDF 0.30102999566398 - TF-IDF: 0.015843683982315

```

Example TF-IDF output for each term in a single document.

sentence can easily be considered positive if specific keywords are used. These words may, however, be associated to a negative response.

Example 11: Sentiment analysis

Synonyms are words that represent the same meaning as other words. An example of this is the word *happy*, synonyms include *cheerful*, *content*, *delighted*, *elated*, and *joyful*. If a feedback record contained the text *delighted* we would want to make this record as having a positive sentiment. Instead of using manual synonym detection, a sentiment analysis tool can be used.

This example contains one file and one library:

1. `sentiment.php` — This page shows an example of the `sentimentanalysis` tool being used.
2. `PHPinsight` — This library provides us with the ability to run the `sentiment analysis` process.

Example code: `sentiment.php`

To perform sentiment analysis in PHP we will use the *PHPinsight* package. This package is available from <https://packagist.org/packages/jwhennessey/phpinsight>. Instead of manually downloading files and then attempting to integrate them into the project, we will use *composer*.

From the command prompt in the root directory of our project, we will issue the following command:

```
composer require jwhennessey/phpinsight
```

SAMPLE CODE 18: Example `composer require` for *PHPinsight*.

In Sample Code 19, on line 2, a *pre* tag is used to format the generated output. On line 3, an array is added that contains the sentences that the sentiment analysis process will be run on. Additional sentences can be added into this array for analysis. On line 8, a reference to the *vendor* folder is added that contains *PHPinsight*. The *autoload.php* is used to pull in the required functionality from the library for the sentiment analysis process. A simple *foreach* statement is

used on line 10 to loop over each individual string that has been added to the array. The `score()` function is used on line 12 to retrieve the sentiment scores. On line 13, the `categorise()` function is used to output the dominant result, e.g., *positive*, *negative* or *neutral* for the individual sentences.

```

1      <?php
2      echo "<pre>";
3      $strings = array(
4          1 => 'I am very happy with the staff',
5          2 => 'I did not like the staff they are unhelpful',
6          3 => 'The food was bad',
7      );
8      require_once('vendor/autoload.php');
9      $sentiment = new \PHPInsight\Sentiment();
10     foreach ($strings as $string) { 11// calculations:
12     $scores = $sentiment->score($string); 13     $class = $sentiment->categorise($string);
14         // output:
15         echo "String: $string\n";
16         echo "Dominant: $class, scores: ";
17         print_r($scores);
18         echo "\n";
19     }

```

SAMPLE CODE 19: sentiment.php — Gathering the sentiment of text using the PHPInsight library.

```

String: I am very happy with the staff
Dominant: pos, scores: Array
(
    [pos] => 0.5
    [neg] => 0.25
    [neu] => 0.25
)

String: I did not like the staff they are unhelpful
Dominant: neg, scores: Array
(
    [neg] => 0.5
    [pos] => 0.25
    [neu] => 0.25
)

String: The food was bad
Dominant: neg, scores: Array
(
    [neg] => 0.5
    [pos] => 0.25
    [neu] => 0.25
)

```

FIGURE 5.15: Result of the sentiment-analysis process.

Output

shows an example of the output of the sentiment analysis process. Three different scores ranges between 0.0 and 1.0 are shown. These show the positive, negative, and neutral score for each sentence. The dominant score can also be seen, indicating the overall highest out of all three categories.

5 THE DOCUMENT OBJECT

Defining the Document Object

The document object is an object that is created by the browser for each new HTML page (document) that is viewed. By doing this, JavaScript gives you access to a number of properties and methods that can affect the document in various ways.

You have been using the `write()` method of the document object for quite some time in this book. This method allows you to write a string of text into an HTML document.

To begin your journey through the document object, you will take a look at the Document Object Model (DOM) and the various properties you can access with this object. Many of these properties will turn out to be quite useful when writing scripts.

NOTE

The DOM is unique to the web browser environment and is not something used when writing Node.js code, for instance. Also, some nonstandard DOM methods may work only in particular browsers, so you will typically want to stick to those methods that work in all browsers.

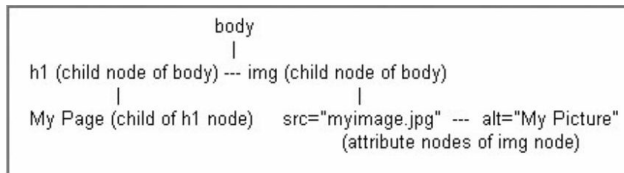


Figure 9-1 An example of part of a document’s structure

Using the Document Object Model

The Document Object Model (DOM) allows JavaScript (and other scripting languages) to access the structure of the document in the browser. Each document is made up of structured nodes (for example, the body tag would be a node, and any elements within the body element would be child nodes of the body element). With this structure in place, a scripting language can access the elements within the document in a number of ways, allowing for the modification of the elements within the document.

If you had the following HTML code, you could use JavaScript to access its structure:

```
<body>
  <h1>My Page</h1>
  
</body>
```

Figure 9-1 shows how the body element is a node, and how it can have child nodes and attribute nodes.

The h1 and img elements are both child nodes of the body element. Each element also has its own nodes. The h1 element contains a text node as its child node (the text “My Page”), while the img element contains two attribute nodes (src=“myimage.jpg” and alt=“My Picture”). This type of structure is available throughout the document, so while this is a simple example, much more complex document structure trees could be drawn for most HTML pages.

You can write scripts to add, remove, or change nodes in the DOM. You can use the `document.getElementById()` method to access elements by their id attribute values, and even get groups of elements using methods like `document.getElementsByTagName()` or `document.`

getElementsByClassName().

First, you will look at the properties and methods of the document object.

Using the Properties of the Document Object

lists commonly used properties of the document object with a short description of each. Following the table, some specific properties are discussed in more detail.

Property	Description
activeElement	Returns the active (focused) element
all	Gets all the nodes in a document
baseURIObject	Returns an object representing the document's URI. Only available in privileged mode.
body	Returns the body or frameset element of the document
characterSet	Returns the character set used in the document
compatMode	Returns the string "BackCompat" if the document is rendered in Quirks mode or the string "CSS1Compat" if the document is rendered in Strict mode
contentType	Returns the Content-Type value from the document's MIME header
cookie	Returns or sets a JavaScript cookie in a document
currentScript	Returns the script element that is currently being read
defaultView	Returns the window object for the document
designMode	Allows a document to be edited
dir	Returns or sets the reading direction of the document
doctype	Returns the doctype declaration associated with the document
documentElement	Returns a string representing the root node of the document
documentMode	Returns the compatibility mode of the document
documentURI	Returns the location of the document
documentURIObject	Returns an object representing the URI of the document (only available to privileged JavaScript code)
domain	Returns the domain name of the server for the document
embeds	An array of all the embed elements in the document
forms	Returns an array of all the form elements in the document
fullscreenEnabled	Returns whether or not full screen mode is available in the browser
head	Returns the head element of the document
images	An array of all the image (img) elements in the document
implementation	Returns the implementation object of the document
inputEncoding	Returns a string representing the document's encoding
lastModified	Returns the date of the last modification of the document
lastStyleSheetSet	Returns the name of the most recent style sheet set using the selectedStyleSheetSet property.
links	Returns an array of all the hyperlink (<a>) elements in the document

The Properties of the Document Object (*continued*)

Property	Description
location	Returns the URI of the document
namespaceURI	Returns the XML namespace of the document
parentWindow	Returns a reference to the parent window (the parent window's document object)
plugins	Returns an array of all the plugins used in the document
preferredStyleSheet	Returns the preferred style sheet based on the order of style sheet declarations in the document
protocol	Returns the protocol portion of the Web address (URL) of the document
readyState	Returns the loading state of the document
referrer	Returns the URL of the document that referred the viewer to the current document
scripts	Returns an array of all the script elements used in the document
scrollingElement	Returns the element that scrolls the document, typically the root element
selectedStyleSheetSet	Returns or sets the current style sheet set
styleSheets	Returns an array of all the style sheets used in the document
styleSheetSets	Returns an array of all the style sheet sets in the document
textContent	Returns or sets the text content
title	Returns the text used inside the title tags of the document
uniqueID	Returns a unique ID that is generated for the document
URL	Returns the URL of the current document
URLUnencoded	Returns the URL of the document without any encoding
xmlStandalone	Returns or sets the standalone value found in the declaration of an XML document
XSLDocument	Returns the root element of an XSL document

The Properties of the Document Object

NOTE

Not all these properties work across all browsers. You can see more information and additional properties by visiting <http://developer.mozilla.org/en/DOM/document#Properties> and [https://msdn.microsoft.com/en-us/ie/ms535862\(v=vs.94\)](https://msdn.microsoft.com/en-us/ie/ms535862(v=vs.94)).

Collections

A number of the properties (anchors, embeds, forms, images, links, plugins, scripts, styleSheets, and styleSheetSets) return an array that holds a collection of elements or values found in the document. These can be accessed like any array using indexes. The element or value at index zero will be the first, index one the second, and so on.

For example, the links property can be used to access all of the links in the

document. If you need to find the first link in the document, you could use `document.links[0]` to access that element as an object. You can then use DOM node properties or methods (discussed later in this chapter) to get or set information, or to perform certain tasks on the element.

The cookie Property

The cookie property is used to set a JavaScript cookie to store information for the viewer. A cookie is a small text file saved to the viewer's computer for a particular amount of time (a set date or a browser session). Cookies can be helpful in allowing a site to remember and retrieve information for a viewer (such as the contents of a shopping cart, special settings, or session information).

To set a cookie, you set the value of the `document.cookie` property to a string that contains the information you want to store for the viewer. The following is the syntax:

```
document.cookie = string;
```

You would replace *string* with a text string that contains the information you want to use.

Usually, this is in a format like the one shown in the following example of setting a cookie:

```
document.cookie = "site=homepage";
```

You can see that there is one thing set by the cookie: the site is homepage. In between the two the equal sign is used to help separate the site and homepage when the cookie is read. Note that setting the cookie does not replace the whole cookie; it just adds the new string to it.

You will see how to use advanced string-handling techniques and how to set and read cookies in more detail in later chapters.

The dir Property

The `dir` property returns a string value that represents the reading direction of the document. This value can be either `ltr` (left to right) or `rtl` (right to left). This property is useful for displaying Web pages in languages that are read from right to left rather than left to right, by setting the value to `rtl`. For fun, you can change the way your page looks on-the-fly with this property, as in the following example code:


```
document.dir = "rtl";
```

shows an example of how a page would look with the dir property set to “rtl”.

The lastModified Property

The lastModified property holds the value of the date and time the current document was last modified. This is used mostly for informational purposes, such as displaying the date the document was last modified so the viewer knows when you last updated your page. The value of this property depends on your browser, as different browsers have different results if you write the last modified date on the page.



A Web page with the direction switched to rtl

Consider the following code, which writes the value of the document.lastModified property into a Web page to display the last modified date and time:

```
<body>
<h1>My Always Updated Web Page!</h1>
<script type="text/javascript">
document.write("Last Updated: " + document.lastModified);
</script>
</body>
```

shows the result of this when viewed in Mozilla Firefox.

When writing the date of the last modification on the page, the differences only matter in terms of space on the page. Some layouts may need to have extra space arranged for the longer version of the property.

The referrer Property

The referrer property is used for informational purposes and holds the value of the URL of the page that the viewer was on before arriving at your page (the referring page). While this can be useful, the viewer doesn't always come in with a referring URL (such as when using a

bookmark or typing in the address), so the value could be nothing. Also, the value of this property isn't always correct, because different browsers may consider different types of things



The last modified date when viewed in Mozilla Firefox

as referring the viewer to the new page, rather than just links, and it is possible for the user to hide or change the referrer.

Placing the code in the following example into the document would send an alert to the viewers of a page telling them where they were before they got to your page:

```
window.alert("You came from " + document.referrer + "!!");
```

So, if the referring page were <http://www.scripttheweb.com/js/>, an alert saying “You came from <http://www.scripttheweb.com/js/!>” would be sent to the viewer.

The title Property

The title property holds the string value of the title of the HTML document. The title is set inside the <title> and </title> tags of a page.

One way you can use the title property is to display the title of the page to the viewer someplace other than in the top bar of the window. The following code would allow you to do this:

```
<head>
<title>Lions, Tigers and Bears!</title> ← The title of the document is set here </head>

<body>
<script type="text/javascript">
document.write("<h1>" + document.title + "</h1>"); ← </script> Lions and tigers and bears
were what I saw when I went to ... </body>
```

The title is shown as a heading to the viewer

This displays your title as a heading on the page. shows the result of this when viewed in a browser.



The title of the document is shown as a heading on the page.

The URL Property

The URL property holds the value of the full URL of the current document. This information can be useful if you print it at the bottom of your HTML page, because it will show the page URL for anyone who prints out your page.

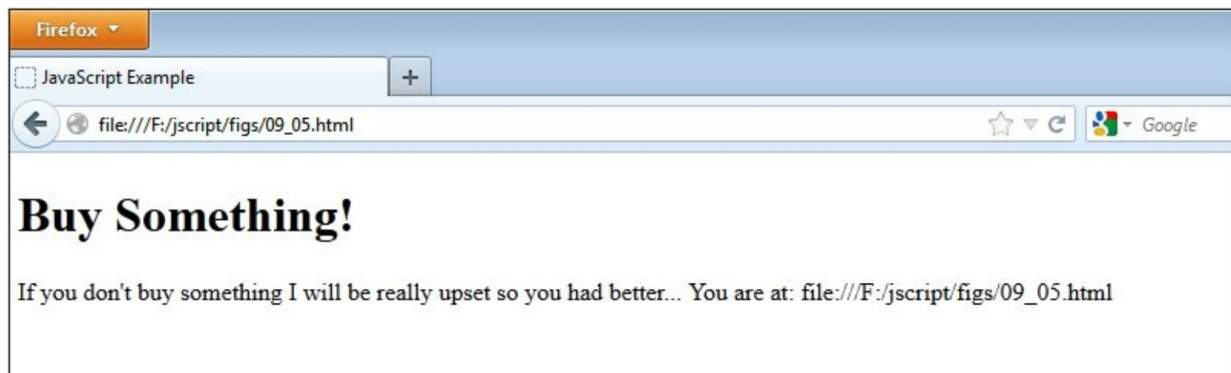
While you could just type the URL at the bottom on your own, this could become tedious when it needs to be done on numerous pages. This is where this property can be handy, because you can cut and paste a little script to each page rather than type the various URL addresses each time. An example of writing the URL address on the page is shown in the following code:

```
<body>
<h1>Buy Something!</h1> If you don't buy something I will be really upset so you had better...
<br><br>
<script type="text/javascript">
document.write("You are at: " + document.URL);
</script>
</body>
```

Figure 9-5 shows the result of the preceding code in a browser. The last line of the page tells the viewer the current location. The figure shows a local file address, but it would show a regular URL if the page were online.

The URLUnencoded Property

The `URLUnencoded` property returns the URL of the document without any encoding. For instance, if there is a filename with a space in it, the property will return the space rather than a `%20` in its place. For the URL <http://www.scripttheweb.com/my script.html>, the document `.URL` property would return <http://www.scripttheweb.com/my%20script.html>. The document `.URLUnencoded` property returns <http://www.scripttheweb.com/my script.html>, the URL without the encoding for the space. Note that at the time of this writing this property was only available in Microsoft Internet Explorer.



The URL of the document is shown at the end of the page contents.

Q: A: Q: A: Q: access elements in the document? A: section.	

Using the Methods of the Document Object

The methods of the document object allow you to do some new things that you haven't been able to do yet. Table 9-2 lists common methods with a short description of each. Because a number of these methods are browser-specific (as with the properties), only some specific methods are described in more detail following the table.

NOTE

Not all these properties work across all browsers. You can see more information and properties by visiting <http://developer.mozilla.org/en/DOM/document#Methods>.

The get Methods for Elements

There are a number of methods that allow you to get one or more elements in the document for use in your scripts. Each of these provides you a unique way to gain access to any elements you may need.

Method	Description
<code>adoptNode()</code>	Adopts a node from another document
<code>createAttribute()</code>	Creates an attribute with a name that is sent to it as an argument
<code>createCDATASection()</code>	Creates a new CDATA section
<code>createComment()</code>	Creates a comment with the value that is sent to it as an argument
<code>createDocumentFragment()</code>	Creates a new document fragment
<code>createElement()</code>	Creates an element of the type sent to it as an argument
<code>createElementNS()</code>	Creates an element in a particular URI and a particular type sent to it as arguments
<code>createEvent()</code>	Creates an event
<code>createNodeIterator()</code>	Creates a node iterator object
<code>createNSResolver()</code>	Creates a namespace resolver
<code>createProcessingInstruction()</code>	Creates a processing instruction
<code>createRange()</code>	Creates a range object
<code>createStyleSheet()</code>	Creates a style sheet for the document to use (Internet Explorer only)
<code>createTextNode()</code>	Creates a text string from the value sent to it as an argument
<code>createTreeWalker()</code>	Creates a treewalker object
<code>elementFromPoint()</code>	Returns the element object that appears at the location that is sent to it in two argument values (pixels from left and pixels from top)
<code>evaluate()</code>	Returns a result based on the arguments sent to it
<code>execCommand()</code>	Executes a command on the document when the document is in design mode
<code>getElementById()</code>	Returns a reference to the object with the ID attribute that is sent to it as an argument
<code>getElementsByName()</code>	Returns references to the elements with the class name that is sent to it as an argument
<code>getElementsByTagName()</code>	Returns references to the objects with the name attribute that is sent to

	it as an argument
getElementsByTagName()	Returns references to the elements with the tag name that is sent to it as an argument
getElementsByTagNameNS()	Returns references to the elements with the tag name and namespace sent to it as arguments
getSelection()	Returns the value of a string of selected text in the document
hasFocus()	Returns a Boolean value based on whether or not the document has focus

Methods of the Document Object (*continued*)

Method	Description
importNode()	Returns a copy of a node from another document
isDefaultNamespace()	Returns whether or not the namespace is the default namespace for the current document
isEqualNode()	Returns whether two nodes are equal or not
isSameNode()	Returns whether or not two node references refer to the same node
load()	Loads an XML document
lookupNamespaceURI()	Returns a namespace URI
lookupPrefix()	Returns the namespace of a specified prefix
normalize()	Normalizes the nodes in a document so that there are no empty text nodes and so that there are not multiple text nodes next to one another (makes them all a single text node)
close()	Closes a new document that has been opened with the open() method
open()	Opens a new document that allows you to write its contents using write() or writeln() statements
queryCommandEnabled()	Returns a Boolean value based on whether or not a command sent to it as an argument can be executed
queryCommandIndeterm()	Returns a Boolean value based on whether or not a command sent to it as an argument is in the indeterminate state
queryCommandState()	Returns a Boolean value based on whether or not a command sent to it as an argument has executed
queryCommandSupported()	Returns a Boolean value based on whether or not a command sent to it as an argument is supported
queryCommandValue()	Returns the current value of the document for the command that is sent to it as an argument
querySelector()	Returns the first element node that matches the selectors
querySelectorAll()	Returns all of the element nodes that match the selectors
write()	Allows you to write a string of text into an HTML document
writeln()	Allows you to write a string of text into an HTML document, but ends the line with a JavaScript newline character

The Methods of the Document Object The

getElementById() Method

The `getElementById()` method allows you access to an element by the value of its `id` attribute. For example, if you have the following HTML code, you can access the `div` element with the `getElementById()` method:

```
<div id="some_text">This is some text.</div>
```

Since the `id` attribute of the `div` element has the value of `some_text`, the `document.getElementById()` method can access the `div` element using that value as an argument:

```
let text_element = document.getElementById("some_text");
```

Valid HTML only allows an `ID` value to be used on a single element. If the same `ID` value is used on multiple elements, `getElementById` will only get one of the elements using that `ID`. As you go through this chapter, you will see that you can use this and the other selection methods to retrieve information from elements, alter elements, add elements, or delete elements.

The `getElementsByClassName()` Method

This method allows you to get an array filled with all the elements in the document that have the specified class name (from a CSS class). For example, to obtain all of the elements with a class name of `number_one`, you could use the following code:

```
let my_class = document.getElementsByClassName("number_one");
```

The `getElementsByName()` Method

This method allows you to get an array filled with all the elements in the document that have the specified value for the `name` attribute. In many cases, this value is unique to one element. However, radio buttons that are part of the same group all need to have the same `name` value (so that only one selection can be made). This is a case where being able to retrieve all of those elements could be handy:

```
<fieldset>
<legend>Choose a dinosaur:</legend>
<input type="radio" name="dino" id="dino_enigmo" value="Enigmosaurus">
<label for="dino_enigmo">The mysterious type</label>
<input type="radio" name="dino" id="dino_spino" value="Spinosaurus">
<label for="dino_spino">One with lots of backbone</label>
<input type="radio" name="dino" id="dino_xeno"
value="Xenotarsosaurus">
<label for="dino_xeno">The unique name type</label> </fieldset>
```

This HTML code has a set of radio buttons, all named `dino`. If you want to get all of these elements, you could use the following code:

```
let dino_radios = document.getElementsByName("dino");
```

This will retrieve all of the radio button elements with the name dino in an array, which you could then loop through to perform any necessary tasks:

```
for (let i = 0; i < dino_radios.length; i++) {  
  // Do some work... }
```

As you can see, you could now work with all of the elements in a loop, rather than gathering each one by its id to perform the same task.

The `getElementsByTagName()` Method

This method allows you to get an array filled with all the elements in the document that have the specified tag name. For example, to obtain all of the image elements in the document, you could use the following code:

```
let all_images = document.getElementsByTagName("img");
```

The `querySelector()` Method

This method allows you to use a CSS selector to get the first element that matches the selector. For example, you can get an element by ID:

```
let mydiv = document.querySelector("#mydiv");
```

Notice that the CSS ID symbol is used here, whereas it is not with `document.getElementById()`. The same holds for CSS classes and other types of selections. For example, to select by class name, you could use the following code:

```
let header = document.querySelector(".header");
```

Remember that `querySelector()` only selects the first match, so even if you have multiple elements with the header class, only the first element with the header class will be selected.

The `querySelectorAll()` Method

Whereas `querySelector()` selects only the first matching element, the `querySelectorAll()` method will select all matching elements for the provided CSS selector. For example, consider the following code:

```
let headers = document.querySelectorAll(".header");
```

This time, all elements that have the header class will be selected rather than just the first one, and you can access each individual element through a loop, much like you did with `getElementsByClassName()`:

```
for (let i = 0; i < headers.length; i++) {  
  // Do some work... }
```


Getting More Specific

The get methods can be used with more specific objects once you have them. For instance, suppose you had the following HTML code:

```
<nav id="sitelinks">
<a href="page1.html">Page 1</a>
<a href="page2.html">Page 2</a>
  <a href="page3.html">Page 3</a>
</nav>
<div>
  Here is some content and check out <a href="someplace.com">my other site</a>! </div>
```

If you wanted to get only the links within the nav element, using document

.getElementsByName("a") wouldn't work, because you would also get the link that is within the div tag (still part of the document). To do this, you can first get the nav element and then use the getElementsByTagName() method on the object created for the nav element, as in the following code:

```
let nav_element = document.getElementById("sitelinks"); let nav_links =
nav_element.getElementsByTagName("a");
alert("There are " + nav_links.length + " links in the nav element.");
```

The number of links in the nav element is displayed using the length of the nav_links array that was created by getting each of the a elements within the nav element

Here, you get the nav element by its id, creating the nav_element object. This object is used to get all of the link elements within it and place them into an array named nav_links. This array can now be used to cycle through each link within the nav element or to find out how many links are contained within the nav element. In this case, the number of links is alerted using the length of the nav_links array (3).

The open() and close() Methods

The open() method allows you to open a new document and create its contents entirely with document.write() or document.writeln() statements. When the open() method is called, the browser looks for these statements so that it can write the new page. Once the write() and/or writeln() statements are completed, you need to use document.close() to finish the new page.

NOTE

The `open()` and `close()` methods are not necessary (or desirable) when you are writing to the current loading page.

To get an example of the use of the `open()` method, suppose you want to write a new page based on the name of the viewer. To do this, you not only need to use the `open()` and `close()` methods, but also need to create a `formName` property to use so that you can grab the name entered by the viewer in a text box.

Start with the code for the body section of the initial page. You need a form with a text box and a way to invoke a function that will create the new document. The following code shows a way that you can do this (save the file as `document_open.html`):

```
<body>
<strong>Enter your name in the box below, then click the button to see a personalized page!
</strong>
    <br>
    <form id="newp" onsubmit="newpage();" >
        Name: <input type="text" id="yourname" size="25">
        <br><br>
        <input type="submit" value="Submit">
    </form>
<script type="text/javascript" src="document_open.js"></script>
</body>
```

The form is given an id and a function (newpage) to run when submitted

The text box is given an id

This sets up your script, giving you a form with an id of `newp` and a text box with an id of `yourname`. It also has a submit button to submit the form. When the user clicks the submit button, the `onsubmit` event handler will be triggered, which executes the `newpage()` function (you will learn more about event handlers later in this chapter and in Chapter 10). You now need to create the `newpage()` function in your external JavaScript file so that this form will work.

The `newpage()` function needs to grab the contents of the text box and assign it to a variable. It then needs to open your new customized page in the browser window. The following code shows how this can be done (save the file as `document_open.js`):

```
function newpage() { The value of the text box contents is assigned to a variable let thename =
document.getElementById("yourname").value; document.open(); A setup for a new document is opened
document.write("<h1>Welcome!</h1>");
```

```
document.write("Hello, " + thename + ", and welcome to my page!"); document.close(); The new document uses these
statements
```

```
}
The setup for the new document is to know what to display
closed, allowing it to be displayed
```

The first thing the function does is to grab the contents of the text box. To get the contents of the text box, you need to use the value property that is available for form elements, which in this case is accessed using `document.getElementById("yourname").value`. This value is then assigned to the `thename` variable for easy use within your `document.write()` commands.

Once you have that value, you are ready to open the new page. To do this, you use the `document.open()` command, which allows you to use a series of `document.write()` statements until the `document.close()` command is used. You use the `document.write()` statements to write a greeting to the viewer on the page.

You can now try this out by opening the HTML page in your browser. Figure 9-6 shows the initial page with the form (the page before the form button is clicked). This is where the viewer can enter a name and click the button.



This is the page that allows the viewer to enter information.

shows the result of entering the name “John” in the text box and clicking the button. The new page appears with a greeting!

The write() and writeln() Methods

You started using the write() method early in the book, so you know how it works already.

The document.write() method is used to display a string value on the page where it is called. The writeln() method works the same way, but adds a newline character (\n) at the end of the statement. Recall that Chapter 3 discussed how the JavaScript newline character works—it only places a new line in the page source code and not in the final appearance of the HTML page.

TIP

While the newline character only affects the appearance of the source code when using document.write(), it can be used to create new lines in display elements created by JavaScript such as alert, prompt, and confirm boxes.



Once the button is clicked, the viewer will get a new page similar to this one.

The appearance of the page itself is not affected by the JavaScript newline character. Recall the example from Chapter 3 that split the code into two different lines with the newline character:

```
<body>
<script type="text/javascript">
document.write("<strong>JavaScript Rules!</strong>\n This is fun.");
</script>
</body>
```

Since the document.writeln() method adds a newline character at the end

of the statement, you could rewrite the preceding code using the following `document.writeln()` statements:

```
<body>
<script type="text/javascript">
document.writeln("<strong>JavaScript Rules!</strong>");
document.writeln(" This is fun.");
</script>
</body>
```

This would put the lines of code on two different lines in the page source, but would not affect the appearance of the page in the browser.

Using DOM Nodes

There are methods of the document object (such as `createElement()`, `createAttribute()`, and `createTextNode()`) that allow you to create various elements or nodes on the page using JavaScript. To make use of them, though, the new content must be appended as a child of an existing node in the DOM. This is where DOM node properties and methods are needed.

DOM Node Properties

Each DOM node is an object with properties and methods that can be accessed. The DOM node, HTML Element, and Element properties are listed in Table 9-3.

When Table 9-3 mentions the specified node, a node works in much the same way as you worked with elements in previous chapters. For instance, you might have the following HTML code:

```
<body>
<div id="div1" title="All about me!"> This page is about me, me,
and... me!
</div> </body>
```

Property	Description
attributes	Returns an array of all of the attributes in the specified node
baseURI	Returns the base URI of the node
childElementCount	Returns the number of child nodes that are element nodes
childNodes	Returns an array of all the child nodes of the specified node
children	Returns an array of all the child elements of the specified node
classList	Returns a list of the classes within the class attribute of the element
className	Returns the value of the class attribute of the element
clientHeight	Returns the height, in pixels, of the element
clientWidth	Returns the width, in pixels, of the element

clientLeft	Returns the width, in pixels, of the left border of the element
clientTop	Returns the width, in pixels, of the top border of the element
contentEditable	Returns whether or not the element is editable
dataset	Gives access to custom data attributes of the element
dir	Returns the value of the direction of the text in the specified node
firstChild	Returns the first child node of the specified node
firstElementChild	Returns the first direct child element of the specified element
id	Returns the id of the specified node
innerHTML	Returns or sets the HTML code within the specified node
innerText	Returns or sets the inner text of the specified node
isConnected	Returns whether or not a node is attached to the DOM
isContentEditable	Returns whether or not the content of the element can be edited
lang	Returns the language value of the specified node
lastChild	Returns the last child node of the specified node
lastElementChild	Returns the last direct child element of the specified element
localName	Returns the local portion of the node name
name	Returns the name attribute of the specified element
namespaceURI	Returns the namespace URI of the specified node
nextSibling	Returns the node following the specified node
nextElementSibling	Returns the next element on the same level as the specified element

The DOM Node Properties (*continued*)

Property	Description
nodeName	Returns the name of the specified node (such as div for a div element)
nodeType	Returns the type of the specified node
nodeValue	Returns the value of the specified node (such as the text within a div element or the value of an attribute)
offsetHeight	Returns the offset height of the specified node
offsetParent	Returns the nearest containing element position-wise
offsetTop	Returns the offset of the node from the closest relative positioned element in pixels
offsetWidth	Returns the offset width of the specified node
outerHTML	Returns or sets the HTML code of the specified node (includes the specified element itself)
ownerDocument	Returns the document object that contains the specified node
parentElement	Returns the parent element of the specified node
parentNode	Returns the parent node of the specified node
prefix	Returns the namespace prefix of the specified node
previousElementSibling	Returns the element before the specified node
previousSibling	Returns the node before the specified node
scrollLeft	Returns the difference between the left edge and the left edge in view of the specified node
scrollTop	Returns the difference between the top edge and the top edge in view of the

	specified node
scrollHeight	Returns the entire height (including anything hidden and viewable via a scroll bar) of the specified node
scrollWidth	Returns the entire width (including anything hidden and viewable via a scroll bar) of the specified node
style	Returns the style object of the specified node
tabIndex	Returns the tab index of the specified node
tagName	Returns the tag name (in uppercase) of the specified node
textContent	Returns or sets the text content of a node
title	Returns the value of the title attribute of the specified node

The DOM Node Properties If you wanted to obtain the value of the title attribute of the div element, you could use `document.getElementById()` to grab the div element by its id of `div1`. This would be the specified node for the DOM node title property. Then, you could access the title property of the element node, as in the following code:

```
let me_div = document.getElementById("div1");
window.alert("The title of the div element is " + me_div.title);
```

Gets the element by its id and assigns it to a variable

Alerts the value of the element's title attribute to the viewer

This works just like object properties, as you learned in the previous chapter. The `me_div.title` property returns the string value “All about me!”, which is the value of the div element node’s title attribute.

Knowing this, you can use the DOM node methods in the same way.

DOM Node Methods

Table 9-4 lists some commonly used DOM node methods. Note that some methods of the document object mentioned previously can be used on nodes, so they are not repeated here.

As mentioned earlier, to make the creation methods of the document object useful by adding the created node to the document, a DOM node method, such as `appendChild()` or `insertBefore()`, is needed to add the new node to the document.

For instance, you might have the HTML code used earlier, as follows:

```
<body>
<div id="div1" title="All about me!"> This page is about me, me, and...
me!
</div> </body>
```

This code has a div element node with a child text node (and attribute nodes).

If you want to create another div element as the last child node of the div1 element node, you could use a combination of document.createElement(), document.createTextNode(), and the DOM node method appendChild().

Each node is an object that can use all of the node properties and methods. Since you have a div node available, you can use getElementById() to access it as an object, which will allow you to add an element within it using the appendChild() method.

First, go into the JavaScript code and grab the div1 element by its id and assign it to a variable:

```
let me_div = document.getElementById("div1");
```

Next, create the new element node using document.createElement():

```
let inner_div = document.createElement("div");
```

Method	Description
addEventListener()	Adds an event listener to the specified node to run a function on the event sent to it as an argument
appendChild()	Appends a node as the last child of the specified node
attachEvent()	Attaches an event to the specified node to run a function on the event sent to it as an argument
blur()	Removes focus from the specified node
click()	Executes the click event on the specified node
cloneNode()	Creates a clone of the specified node
compareDocumentPosition()	Returns a number representing the position of the node in comparison to the node sent as an argument
detachEvent()	Detaches an event from the specified node
dispatchEvent()	Executes an event on the specified node
focus()	Gives focus to the specified node
getAttribute()	Returns the value of the attribute name sent to it as an argument on the specified node
getAttributeNS()	Returns the value of the attribute name and namespace sent to it as an argument on the specified node
getAttributeNode()	Returns the attribute node of the attribute name sent to it as an argument for the specified node
getAttributeNodeNS()	Returns the attribute node of the attribute name and namespace sent to it as arguments for the specified node
getElementsByName()	Returns an array of all the child element nodes with the tag name sent to it as an argument in the specified node
getElementsByTagNameNS()	Returns an array of all the child element nodes with the tag name and namespace sent to it as arguments in the specified node
hasAttribute()	Returns true if the attribute name sent to it as an argument exists on the specified node, or false if not
hasAttributeNS()	Returns true if the attribute name and namespace sent to it as arguments exist on the specified node, or false if not

<code>hasAttributes()</code>	Returns true if the specified node has any attribute nodes defined, or false if not
<code>hasChildNodes()</code>	Returns true if the specified node has any child nodes, or false if not
<code>insertBefore()</code>	Inserts a node sent to it as an argument before the node sent to it as a second argument inside the specified node

The DOM Node Methods (*continued*)

Method	Description
<code>normalize()</code>	Normalizes the specified node
<code>removeAttribute()</code>	Removes the attribute node for the attribute name sent to it as an argument from the specified node
<code>removeAttributeNode()</code>	Removes the attribute node for the attribute node object reference sent to it as an argument from the specified node
<code>removeAttributeNS()</code>	Removes the attribute node for the attribute name sent to it as an argument with the namespace sent to it as an argument from the specified node
<code>removeChild()</code>	Removes the child node sent to it as an argument from the specified node
<code>removeEventListener()</code>	Removes an event listener from the specified node
<code>replaceChild()</code>	Replaces the child node sent to it as the second argument with the child node sent to it as the first argument in the specified node
<code>scrollIntoView()</code>	Scrolls the specified node into view in the browser window
<code>setAttribute()</code>	Sets an attribute node's name (first argument) and value (second argument) for the specified node
<code>setAttributeNode()</code>	Sets an attribute node as the attribute node object sent to it as an argument for the specified node
<code>setAttributeNodeNS()</code>	Sets an attribute node as the attribute node object sent to it as an argument with the namespace sent to it as an argument for the specified node
<code>setAttributeNS()</code>	Sets an attribute node's namespace (first argument), name (second argument), and value (third argument) for the specified node

The DOM Node Methods

After that, create the text node for the `inner_div` node by using the `document.createTextNode()` method:

```
let inner_div_text = document.createTextNode("More about me...")
```

Next, use the DOM node method `appendChild()` to add the text node as a child of the new `inner_div` node:

```
inner_div.appendChild(inner_div_text);
```

To give the `inner_div` node a title attribute, you can assign a value to its title property:

```
inner_div.title = "More";
```

Finally, use the DOM node method `appendChild()` to add the `inner_div`

node to the document structure as the last (and in this case, only) child element of the me_div node:

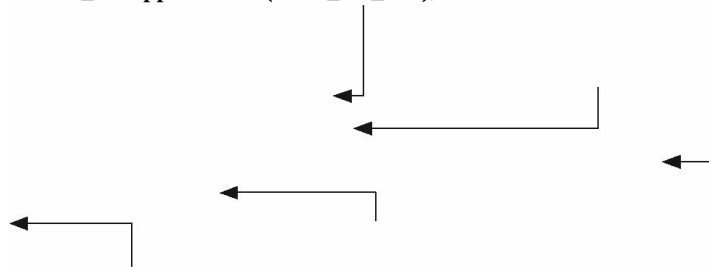
`me_div.appendChild(inner_div);` The div1 element is assigned
to the me_div variable

The complete JavaScript code looks like this:
created and

A new div element is

assigned to the inner_div variable

```
let me_div = document.getElementById("div1"); let inner_div =  
document.createElement("div");  
let inner_div_text = document.createTextNode("More about me...");  
inner_div.appendChild(inner_div_text);
```



`inner_div.title = "More";`

A text node is created

`me_div.appendChild(inner_div);`

The text node

(`inner_div_text`) is added to

and assigned to the `inner_div_text` variable

The `inner_div` element is officially
appended to the DOM structure
inside the `me_div` element using

The `inner_div` element
is assigned a value
for its title attribute `appendChild()`
the `inner_div` element
(the new div element)

This adds your new div node at the end of the original div element (but before the original element is closed, since it will be a child node). Thus, the document structure for the HTML code would now be like this (though it won't show up when you use the browser's "View Source" command):

```
<body>  
<div id="div1" title="All about me!"> This page is about me, me,  
and... me!  
<div title="Me"> More about me...  
</div>  
</div> </body>
```

As you can see, the div is added with the text "More about me..." and a title attribute with a value of "Me".

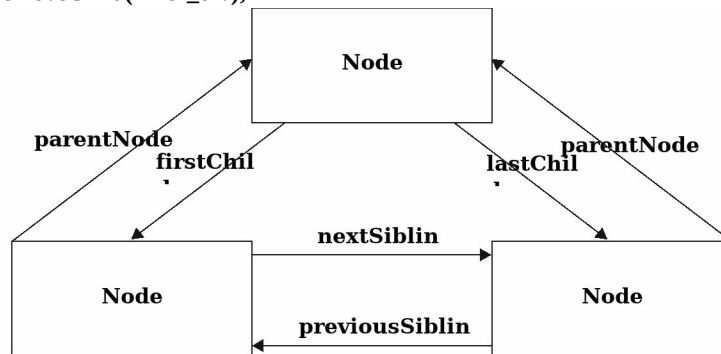
NOTE

Another way to add or alter content is to use the `innerHTML` property, which is discussed in the next section, "Creating Dynamic Scripts."

If you decide you want to delete a node from the DOM structure, you can use the `removeChild()` DOM node method. For example, if you want

to remove the inner_div node added in the previous example, you can use the following code:

```
me_div.removeChild(inner_div);
```



DOM

node relationships

This will remove the inner_div child node from the me_div parent node. Each node has one or more relationships with other nodes in the DOM structure.

Figure 9-8 shows how the relationships are tied together.

As you can see, there are properties and methods that can be used with each of these relationships (for example, previousSibling, parentNode, firstChild, and so on). You can use these to access or alter any element within the DOM structure as needed. For example, you might have an HTML document like this:

← Parent node of the two “employee” divs within it

```
<body>
<div id="company">
  ← <div id="employee1">Marty</div> First child node of the outer “company” div
  ← <div id="employee2">John</div> Last child node of the outer “company” div
</div> </body>
```

Each div element could be accessed in different ways, depending on where you begin. For example, you could access the outer “company” div using any of the following paths:

```
document.getElementById("company"); // direct route
document.getElementById("employee1").parentNode; // from first child
document.getElementById("employee2").parentNode; // from last child
```

Likewise, the “employee1” div can be accessed a number of ways:

```
document.getElementById("employee1"); // direct route
document.getElementById("company").firstChild; // from parent
document.getElementById("employee2").previousSibling; // from its sibling
```

The same, again, goes for the “employee2” div:

```
document.getElementById("employee2"); // direct route
document.getElementById("company").lastChild; // from parent
document.getElementById("employee1").nextSibling; // from its sibling
```

While it may seem easiest to simply use the direct route all the time, there may be situations where you do not know the id of the node you are seeking beforehand. Suppose one of the nodes had been added by another script? If you always wanted to find the last “employee” div, for example, it might be better to use lastChild from its parent node, so that if other employees are added (for example, “employee3”, “employee4”), you will always be accessing the last employee:

```
let last_hired = document.getElementById("company").lastChild;
```

This will ensure that last_hired will always point to the last “employee” div that is a child of the “company” div.

Try This 9-1 *Add a DOM Node to the Document*

This project allows you to practice using the new document and DOM node properties and methods you have learned in this chapter.

Step by Step

1. Create an HTML page with the following code for the body section and save it as pr9_1.html:

```
<body>
<div id="div1" title="Gosh!"> Whatever I feel like I want to
write...
</div> <script type="text/javascript" src="prjs9_1.js"></script> </body>
```

2. Create an external JavaScript file and save it as prjs9_1.js. Use it for

steps 3–5.

3. Get the value of the title attribute of the element with the id of “div1” and send that value as an alert to the viewer.
4. Create a new div element with the text “See you!” and a title attribute with the value “Lucky!”.
Add it to the document structure as a child of the div element with the id of “div1”.
5. Save the JavaScript file and open the HTML file in your browser to view the results.

Try This Summary

In this project, you used your knowledge of the properties and methods of the document object and the DOM nodes to alert a DOM node property and to create a new div element in the document’s structure.

Creating Dynamic Scripts

As you have seen, JavaScript gives you access to all of the elements in the document with the various methods such as `document.getElementById()` and `document.getElementsByTagName()`.

This allows you to get information, add nodes, alter nodes, and remove nodes, as you have seen. In addition, you can access the style attributes of elements (typically, initially set by a style sheet) to make changes to such things as their positions, colors, borders, sizes, or just about any other part of the styles, using the style DOM node property.

Styles in JavaScript

When setting styles using Cascading Style Sheets (CSS), you may set up something like the following in your CSS code (save it as `dyn_01.css`):

```
#div1 { color:#000000; background-color:#FFFFFF; }  
#div2 { border-style:solid; border-width:1px; border-color:#000000; }
```

This gives you style attributes for two ids, `div1` and `div2`. Thus, if you had the following HTML code (save as `dyn_01.html`), the div elements would use the preceding styles in their presentation on the Web page:

The CSS file is linked to the
document here

```

<head> <link rel="stylesheet" type="text/css" href="dyn_01.css">
</head>
<body>

```

I am in div1. It seems like a nice place.

```

<div id="div1">
</div>
<div id="div2">
I am in div2. It's a little fancier here.
</div> <script type="text/javascript" src="dyn_01.js"></script>
</body>

```

The JavaScript file is called here

The HTML code is linked to the CSS code via the link tag. Thus, the div1 element is going to display as simple black text on a white background and the div2 element is going to have a plain, solid-black border around it.

As you can see, this page is using ids for each div element—so not only can you access those elements' ids with the CSS code, but you can also access the elements via their ids in the JavaScript code. So, you can start out your JavaScript file (called in the preceding HTML code—saved as dyn_01.js) with some code to grab both div elements by their ids by using the document.getElementById() method:

```

let d1 = document.getElementById("div1"); let d2 =
document.getElementById("div2");

```

Now you have variables for both elements.

If you want to alter the styles that were set up in the CSS code via JavaScript, you'll need to make use of the style property that is a part of each element node. Then, JavaScript uses the same name as the CSS selector to access that particular property. For instance, if you wanted to change the color of the text in the div1 element to a light gray color, you would use the following code:

```

let d1 = document.getElementById("div1"); let d2 =
document.getElementById("div2"); d1.style.color = "#DDDDDD";

```

In the CSS code, the selector color is used to alter the element's foreground color. In JavaScript, it is also the name of the property used to alter it after accessing the element's style property.

What if the CSS selector is not all one single word? For example, the background-color selector would not work in JavaScript if you used it the same way as in the CSS code. The following code attempts to change the

background color to green:

```
let d1 = document.getElementById("div1"); let d2 =  
document.getElementById("div2"); d1.style.background-color = "#DDDDDD";
```

This won't work, because JavaScript doesn't allow the hyphen (-) character as part of a property name. Instead, JavaScript puts both words together and capitalizes the first letter of any additional words after the first word. Thus, the CSS selector background-color becomes backgroundColor in JavaScript. Other selectors would follow the same pattern (for example, border-right-color would become borderRightColor). This style is called camel case, which is a very common style found in JavaScript.

The JavaScript code could be rewritten as follows to make the change effective:

```
let d1 = document.getElementById("div1"); let d2 =  
document.getElementById("div2"); d1.style.backgroundColor = "#DDDDDD";
```

This will change the background color of the div1 element to green. The only issue now is that this will happen as soon as the script runs, which in this case is probably before the viewer ever notices there was another background color on the div1 element in the first place. To make this more useful, you need a way to run portions of your script at a time other than the loading or reloading of the document.

CAUTION

Reading a style isn't as easy as setting it, as there are browser differences and other issues that occur. For further information and techniques on how to get styles, see www.quirksmode.org/dom/getstyles.html.

Simple Event Handling

So far, your scripts have simply run during the loading (or reloading) of the Web page. This is good for a number of tasks, but many, such as node and style changes, would be more useful if they were executed in response to an action taken by the user. For instance, changing the background color of an element while the page is loading may not even be noticed by the viewer, but changing the color in response to a mouse click would allow the viewer to see the change and provide the choice of leaving the original color or changing to the new color via a click.

The click Event

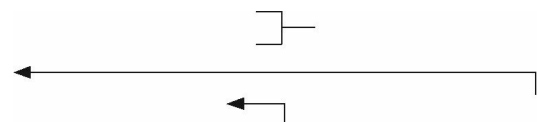
The click event is available for most elements within the DOM. This event is triggered when the user clicks the primary mouse button while the mouse

pointer is over a specific element. One way to gain control of the click event is through onclick, as shown in the following code (other methods will be covered in the next chapter):

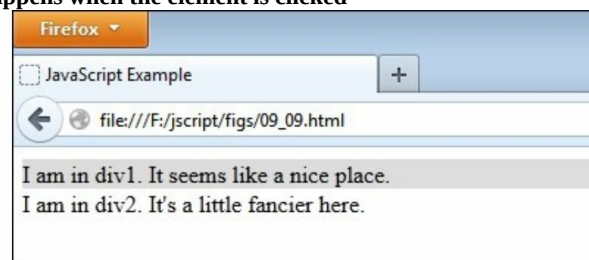
```
let d1 = document.getElementById("div1"); let d2 =  
document.getElementById("div2");  
d1.onclick = function() {  
  // Code to execute when the element is clicked };
```

Here, the d1 element's onclick property is assigned a function expression (refer to Chapter 4 if you need to review function expressions), which can then execute code. Thus, when the d1 element is clicked by the user, any code within the function will be executed.

So, you can now take the code you were using to change styles and perform the task when the user clicks the element rather than while the page is loading. The following code shows an example of this:



```
let d1 = document.getElementById("div1"); The d1 and d2 elements are let d2 = document.getElementById("div2");  
accessed using getElementById() d1.onclick = function() {    d1.style.backgroundColor = "#DDDDDD"; A function  
expression is assigned  
    }; The style of the d1 element is changed within the function, to the d1 element using onclick so that  
it only happens when the element is clicked
```



The background color of the first div element is changed and is now light gray.

shows what the page looks like after the user clicks on the first div element. The background color has changed from white to a light gray.

NOTE

Event handling will be covered more thoroughly in Chapter 10. For the time being, only the click event will be covered using simple DOM0 event handling.

Coding a Dynamic Script

Now that you know how to alter an element's style properties, you can

alter them in reaction to user events to create dynamic scripts. In addition, you can alter the entirety of the HTML code within a given element using the `innerHTML` property.

The `innerHTML` Property

The `innerHTML` DOM node property allows you to change the HTML code that is inside a specified element. For instance, you could start out with the following HTML code:

```
<body>
<div id="div1"> What is 2+2?
</div>
<div id="div2">
<a href="answer.html" id="answer_link">Get the answer</a>
</div> </body>
```

This code sets up a `div` element node with an `id` of `div1`, another `div` element node with an `id` of `div2`, and the link with an `id` of `answer_link`. Now, if you want to change the HTML code in the `div1` element so that it displays the answer for the viewer when the link is clicked, you can use the `click` event on the link to start the script in motion and then change the `innerHTML` property on the `div1` element node to change the contents of that element from the question to the answer:

```
let d1 = document.getElementById("div1");
let answer_link = document.getElementById("answer_link");
answer_link.onclick = function() {
  d1.innerHTML = "That is easy, the answer is <strong>4</strong>!";
  return false; };

```

Note that in order to keep this link from being followed when clicked, you add the `return false` statement after performing the desired actions. This will be discussed in more detail in the next chapter.

When run, this script changes the content inside the `div1` element on the page, so that the HTML code would now be the following (though it will not be seen using the browser's "View Source" command):

```
<body>
<div id="div1"> That is easy, the answer is <strong>4</strong>!
</div>
<div id="div2">
<a href="answer.html" id="answer_link">Get the answer</a>
</div> </body>
```

TIP

To make the script accessible to those without JavaScript, a default link destination (`answer.html`) is used. Including the answer text on the linked page allows users without JavaScript to still click the link and obtain the answer to the question.

As you can see, this can be a handy way to make dynamic changes to the content of a Web page. The `innerHTML` property and `appendChild()` method allow you to change information displayed on the page dynamically and without reloading the page. This provides a definite improvement over `document.write()`, which you have been using up to this point, since `document.write()` can only change content when the page is loaded or reloaded.

Now that you know the newer methods, you will make use of these methods rather than `document.write()` as you continue through this book.

Using what you have learned in this chapter to access the DOM and use the properties and methods of the document object and the DOM nodes, you can build scripts to make any number of alterations to the document's appearance or content.

Try This 9-2 Try Out Property Changes

`pr9_2.html` This project allows you to practice using the new style and `innerHTML` properties `prjs9_2.js` you have learned in this chapter.

Step by Step

1. Create an HTML page with the following code for the body section and save it as `pr9_2.html`:

```
<body>
<div id="div1"> When will I update this page?
</div>
<div id="div2">
<a href="answer.html" id="c_link">Find out!</a>
</div> <script type="text/javascript" src="prjs9_2.js"></script> </body>
```

2. Create an external JavaScript file and save it as `prjs9_1.js`. Use it for steps 3–5.
3. Write some code so that when the link is clicked, the background color of the div element with the id of `div1` changes to `#CCCCCC` and the content of the same element changes to the following:

```
<strong>Right now!</strong> Was that quick or what?
```

4. When complete, the JavaScript file should look like this:

```
let d1 = document.getElementById("div1"); let c_link =
document.getElementById("c_link");
c_link.onclick = function() {
  d1.innerHTML = "<strong>Right now!</strong> Was that quick or
what?"; return false;
```

};

5. Save the JavaScript file and open the HTML file in your browser to view the results.

Try This Summary

In this project, you used your knowledge of the style and innerHTML properties of DOM nodes to make style and content changes to the Web page.



6 THE EVENT HANDLERS

When creating scripts, you will often find that there are user “events” (such as a user moving a mouse over a certain element or clicking a particular element) to which you want your script to react. The way you do this is through the use of event handlers.

To learn how the event handlers work, you need to learn what they are and why they are useful to you. You will then learn where event handlers are placed in a document and how to use them. Finally, you will see the various events in JavaScript and the event handlers that take care of each event. To get started, this chapter presents a general overview of event handlers.

What Is an Event Handler?

An event handler is a predefined JavaScript property of an object (in most cases an element in the document) that is used to handle an event on a Web page.

You may ask the question “What is an event?” An event is something that happens when the viewer of the page performs some sort of action, such as clicking a mouse button, clicking a button on the page, changing the contents of a form element, or moving the mouse over a link on the page. Events can also occur simply by the page loading or other similar actions.

When events occur, you are able to use JavaScript event handlers to identify them and then perform a specific task or set of tasks. JavaScript enables you to react to an action by the viewer and to make scripts that are interactive and more useful to you and to the viewer.

Why Event Handlers Are Useful

Event handlers are useful because they enable you to gain access to the events that may occur on the page. For instance, if you wanted to send an alert to the viewer when he or she moves the mouse over a link, you could use the event handler to invoke the JavaScript alert you have coded to

react to the event. You are now making things happen based on the actions of the viewer, which enables you to make Web pages that are more interactive.

In creating this interactivity, many people find that JavaScript starts to become a little more fun to code and to use. With event handlers, you can create scripts that will add more functionality to the page and also perform practical tasks like validating form input. JavaScript can make a number of things happen on a Web page that will make the page more interesting than a static HTML document.

Understanding Event Handler Locations and Uses

To see how event handlers work, you need to know where you can place them in a document and how to use them to add JavaScript code for an event.

Event handlers can be used in a number of locations. They can be used directly within HTML elements by adding special attributes to those elements. They can also be used within the `<script>` and `</script>` tags or in an external JavaScript file.

NOTE

Node.js uses events and event handlers as well. This chapter will concentrate on browser-based events, while Node.js event handling will be covered in Chapter 11.

To understand better where event handlers are located, you need to learn how to add event handlers to your script.

Using an Event Handler in an HTML Element

To use an event handler directly in an HTML element, you need to know the keyword for the event handler and where to place the event handler within the HTML code. To give you an example, I will introduce the `onclick` event handler, which is used to make something happen when the viewer clicks a specific area of the document.

One element that can be clicked is a form button. So, suppose you want to alert the viewer to something when the user clicks a form button. You would write something similar to the following code:

```
<input type="button" value="Click Me!" onclick="JavaScript code here">
```

To use an event handler, you add it as an additional attribute to an

HTML tag. The only difference between an event handler “attribute” and an HTML attribute is that you can add JavaScript code inside an event handler attribute rather than just an attribute value. In the previous code, you would replace the *JavaScript code here* text with some actual JavaScript code. So, to make an alert pop up when the user clicks the button, you can add the necessary JavaScript code right inside your onclick attribute, as shown in the following example:

```
➤ <form> <input type="button" value="Click Me!" onclick="window.alert('Hi!');" >/form>
```

Notice how the onclick event handler works much like an HTML attribute

When the viewer clicks this plain button, an alert will pop up with a greeting. Notice that the rules on the quote marks apply here. Using the onclick event handler as an attribute requires you to use double quotes around all of your JavaScript code, so when you need quote marks for the alert, you use single quotes in order to avoid possible errors.

Also notice that the alert command ends with a semicolon. This enables you to add additional JavaScript code after the alert, which enables you to perform multiple actions on the click event rather than just a single JavaScript statement.

You could code in two alerts if you wanted to do so. All you have to do is remember to include the semicolons to separate the alert commands. This will be a little different because all of the code will be on one line rather than separate lines, as you normally see:

```
<form>
<input type="button" value="Click Me!"
  ⬅ onclick="window.alert('Hi!'); window.alert('Bye!');" >
</form>
```

Notice how the semicolons separate the JavaScript statements

This example is able to perform two JavaScript statements on the same event by using semicolons to separate them. When using event handlers, you can execute multiple commands this way. It is important, however, to keep everything between the event handler keyword (in this case, onclick) and the ending set of quotes (in this case, after the last semicolon in the code) on one line in your text editor; otherwise, a line break in the code could cause it not to run properly or to give a JavaScript error.

If the code you want to use becomes really long, you may wish to put the

code in a function instead. The event handler can be used for any JavaScript code, so you can use it to call a function you have defined elsewhere. For example, you could place your two alerts within a function inside an external JavaScript file, and call the function from an event handler in the HTML code. First, code the external JavaScript file (here it will be saved as `js_event_01.js`) as follows:

```
function hi_and_bye() { window.alert('Hi!');  
window.alert('Bye!'); }
```

Next, add the script tags and the event handler to your HTML code:

Notice how the function is called using the event handler

```
    <form> <input type="button" value="Click Me!" onclick="hi_and_bye();">  
    </form>  
<script type="text/javascript" src="js_event_01.js"></script>
```

Notice how the function is called using the event handler just like a normal function call within a script. This enables you not only to shorten the code within the event handler, but also to reuse the function on another button click or event later in the page instead of writing the two alerts out again. The use of a function can help you quite a bit, especially when the code you want to use becomes extremely long.

NOTE

As you continue, you will notice that there is a difference between using `type="button"` and `type="submit"`. For a submit type, you will need to return false or prevent the default action using `preventDefault`. You will see more on this as you proceed through the chapter.

Using an Event Handler in the Script Code

You can also use an event handler within the script code (whether using the script tags in the HTML document or using an external JavaScript file). One way to do this is to give the element an id attribute and then use the JavaScript method `document.getElementById()` to access the element, as you learned in

Add the id Attribute

To use the previous script in this way, you will first add an id attribute to the HTML tag for the input button, as shown here:

```
<form> <input type="button" value="Click Me!" id="say_hi">  
</form>  
<script type="text/javascript" src="js_event_01.js"></script>
```

Notice that the button input element was given an id of `say_hi`. You will use

this to access the button and tie it to an event in your script.

Access the Element

The `document.getElementById()` method allows you to access any element in the HTML document that has an `id` attribute using the value of its `id` attribute. In order to access the button input element you have been using with an `id` of `say_hi`, you could use the following code:

```
let hi_button = document.getElementById("say_hi");
```

Now, you can place the function code into a function expression in the JavaScript file (`js_event_01.js`), as shown here:

<pre>window.alert("Hi!"); window.alert("Bye!");</pre>	<pre>let hi_button = document.getElementById("say_hi"); hi_button.onclick = function() { window.alert("Hi!"); window.alert("Bye!"); }</pre>

The button input element is accessed via its `id` (`say_hi`) and assigned to `hi_button`. The `onclick` event handler is used to assign the function expression to the `onclick` event for the button input element.

The function expression (which displays the two alerts) is assigned to handle the click event on the input button. Thus, when the button is clicked, the viewer will see the two alerts!

This method of handling events allows you to place all of your JavaScript code outside of your HTML elements, which keeps your HTML code cleaner (especially if an external JavaScript file is used). Later in this chapter, you will see that newer methods are also available to handle events.

CAUTION

Make sure an element has been added to the page before accessing it with `getElementById()`. This can be done by placing the `<script></script>` tags at the end of the document (just before the closing `</body>` tag) or by using the `load` event to determine that the document has loaded before accessing DOM elements.

Blur	onblur	Viewer removes focus from an element
Change	onchange	Viewer changes the contents of a form element
Click	onclick	Viewer clicks an element
ContextMenu	oncontextmenu	Viewer opens the context menu
Copy	oncopy	Viewer uses the copy command on part of a page
Cut	oncut	Viewer uses the cut command on part of a page
Dblclick	ondblclick	Viewer double-clicks the mouse on the element
Error	onerror	Viewer's browser gets a JavaScript error or an image that does not exist

Focus	onfocus	Viewer gives focus to an element
Keydown	onkeydown	Viewer presses down a key on the keyboard
KeyPress	onkeypress	Viewer presses down a key on the keyboard and the corresponding character is typed
Keyup	onkeyup	Viewer releases a key on the keyboard
Load	onload	Web page finishes loading
Mousedown	onmousedown	Viewer presses the mouse button
Mouseenter	onmouseenter	Viewer moves the mouse over an element (child elements excluded)
Mouseleave	onmouseleave	Viewer moves the mouse away from an element (child elements excluded)
Mousemove	onmousemove	Viewer moves the mouse (moves the cursor)
Mouseout	onmouseout	Viewer moves the mouse away from an element
Mouseover	onmouseover	Viewer moves the mouse over an element
Mouseup	onmouseup	Viewer releases the mouse button
Mousewheel	onmousewheel	Viewer rolls the mouse wheel in either direction
Paste	onpaste	Viewer uses the paste command on part of the page
Reset	onreset	Viewer resets a form on the page
Resize	onresize	A window is resized
Scroll	onscroll	Viewer scrolls an area that is scrollable
Select	onselect	User makes a selection
Submit	onsubmit	Viewer submits a form on the page
Unload	onunload	Viewer leaves the current page

Common Events and Event Handlers

NOTE

There are numerous additional events available that you may find useful. Since the number of available events can change quickly, the table lists some commonly used events. Events for drag-and-drop functionality are discussed in Chapter 16. A complete listing of events relevant to browser-based JavaScript that will stay up to date can be found at developer.mozilla.org/en-US/docs/Web/Events.

Now that you have a general idea about event handlers, you will take a look at some of the most often used ones in detail to see how they work.

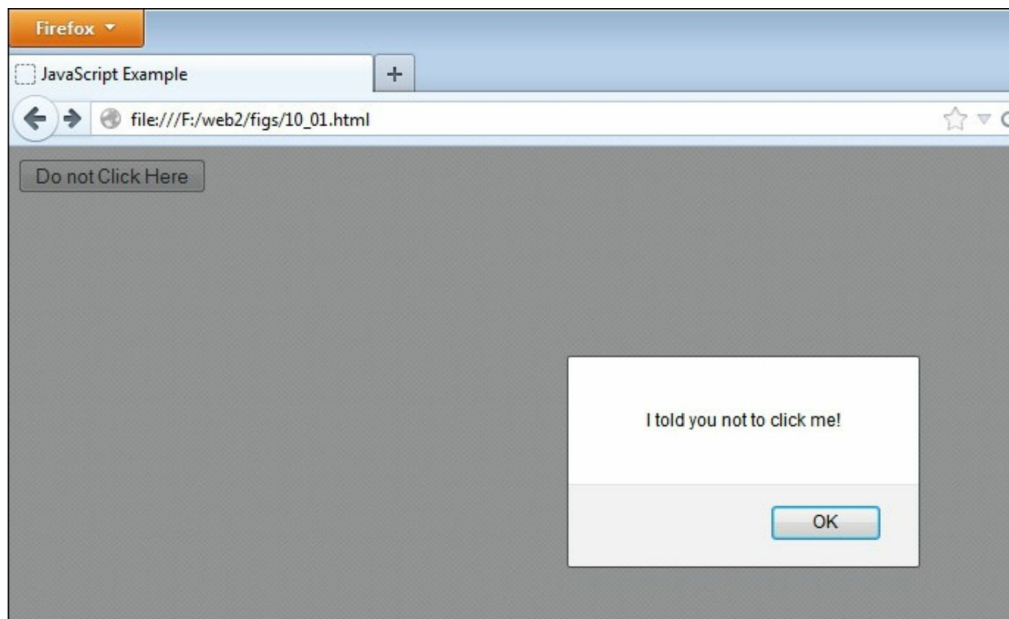
The Click Event

The click event, which you have already been studying, occurs when a viewer clicks on an element in a Web page. For example, you could use a form button and send an alert when it is clicked:

```
<body>   The onclick event handler <form>      is used on a button
          |
          |   <input type="button" value="Do not Click Here"  onclick="window.alert('I told you not to
          |   click me!');">
          |
          |   </form>
          |
          |   </body>
```

This will send the viewer an alert once the button has been clicked. shows

the result of this code when the viewer clicks the button.



This alert pops up when the viewer clicks the button.

To use this event handler to do the same thing with a link, you might be tempted to do something similar to the following:

Oh no! This will give you problems because the browser will try to follow the link

```
<body> <a href="http://none" onclick="window.alert('Hey! You clicked me!');">  
  Don't Click Me</a>  
</body>
```

The problem with this code is that the alert will work, but the browser will try to continue the original action of the link tag and attempt to go to <http://none>. This would probably cause a “Server not found” error in the browser.

One way you can avoid a “Server not found” error is to link to an actual page (which is good for accessibility); however, if the viewer has JavaScript enabled, it may take the viewer away from the current page. To keep the link from being followed when JavaScript is enabled, you need to add an extra statement to the JavaScript in the click event. You need to tell the browser not to continue after you have shown the viewer your alert. To do this, you will add a return false statement, which will keep the browser from following the link after the alert has been shown. The following code shows how to add in the return statement inline:

```
<body>
```

```

<a href="http://none"
  ← onclick="window.alert('Hey! You clicked me!'); return false;">
    Don't Click Me</a>
</body>

```

The return false statement
keeps the browser from trying to
follow the link

With this code in place, the click event will be taken care of by the onclick event handler, and the browser will not need to worry about attempting to follow the link in the href attribute. Later in this chapter, you will learn how to use the modern method of preventing the default action, `event.preventDefault()`.

NOTE

You can also code JavaScript for a link by using the `javascript:` command—for example, `<a href="javascript:window.alert('Hi');">Click</a>`—but this method is not recommended for accessibility reasons (if JavaScript is off, the link doesn't go anywhere).

Focus and Blur Events

The focus event occurs when the viewer gives focus to an element or window. A viewer gives focus to something by clicking somewhere within the item, by using the keyboard to move to the item (often via the tab key), or via a script. For instance, a viewer who clicks a text input box (before entering anything) gives that text box focus. Also, clicking an inactive window and making it the active window gives the window focus. The focus event also has a related method called `focus()`, which is covered in Chapters 11 and 14.

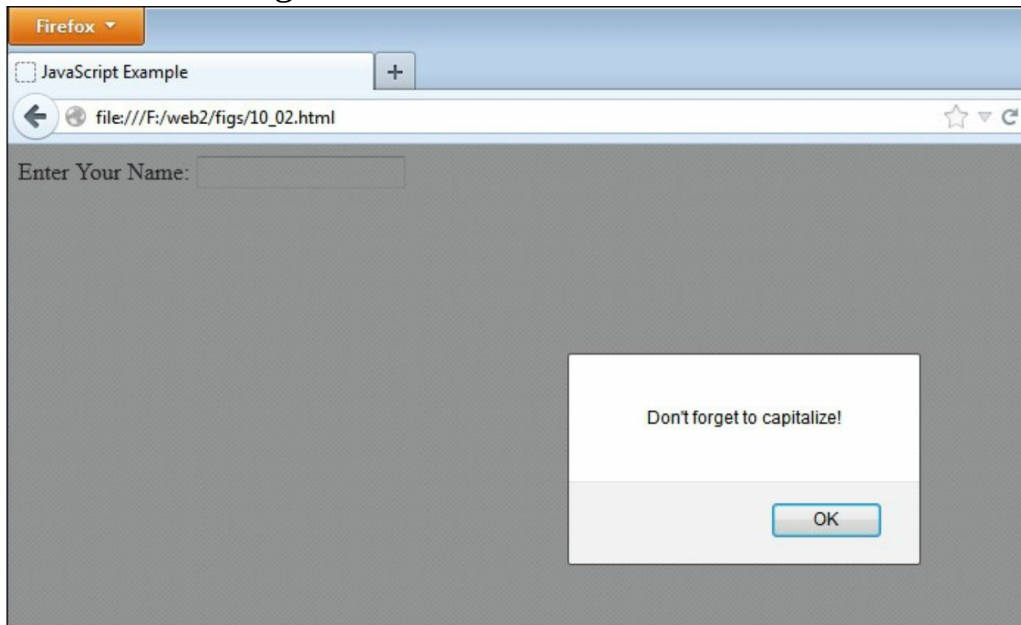
To see the focus event in action, you can create a text input box, which is one of the form elements that will enable you to give the element focus. The following example shows how to do this, as well as how to code a reminder alert to pop up when the viewer gives focus to the text box:

```

<form>                                     The focus event in a text box
  ← Enter Your Name:
    <input type="text" id="user-name">
</form>
<script type="text/javascript">
  let counter = 0;
  let nameElement = document.getElementById("user-name");
  nameElement.onfocus = function () {
    if (counter === 0) {
      window.alert('Don\'t forget to capitalize!');
      counter += 1;
    }
  }
</script>

```

This code will give the viewer an alert before he or she can begin typing. The alert serves as a reminder to capitalize the name. The counter variable is set and incremented in order to keep the alert from coming up every time the user focuses on the text box (limiting it to just the first focus event). Figure 10-2 shows the result of the preceding code in the browser when the viewer gives focus to the text box.



This alert pops up when the text box receives focus from the viewer.

The blur event occurs when the viewer takes the focus away from an element or a window. To take the focus off something, the viewer usually gives focus to something else. For instance, the viewer could move from one form element to another, or from one window to another. The blur event also has a related method called `blur()`, which will be covered in Chapters 11 and 14.

NOTE

The blur event is triggered only when the viewer gives focus to another area, which is the only way the browser will know the viewer released the focus from the first area. For example, when the viewer presses the enter key in an input field, the focus goes from the input field to the document window. Also, the event triggers if you place focus on the text field and then switch tabs.

To see the blur event in action, the following example uses two text boxes: clicking the first text box gives it focus, and clicking the second text box invokes the blur event in the first text box.

`<form>`

The onblur event

Give this box focus:
 in a text box

```
<input type="text" onblur="window.alert('Hey! Come back!');"><br>
```

Then give this box focus to blur the first one:


```
<input type="text">
```

Clicking this text box after giving the focus to the first </form> one will trigger the blur event on the first text box

When viewers click the second text box, they get the alert from the first one telling them to come back.

Figure 10-3 shows the result of the preceding code when run in the browser. Notice that the focus is in the second text box when the alert pops up. By clicking the second text box, the viewer invoked the blur event in the first text box.

The Load and Unload Events

The load event occurs when a Web page finishes loading. If you want an alert to be shown when the page has finished loading, you could use the following code:

```
<body onload="window.alert('I'm done loading now!');"> Text for the body of the page...
</body>
```

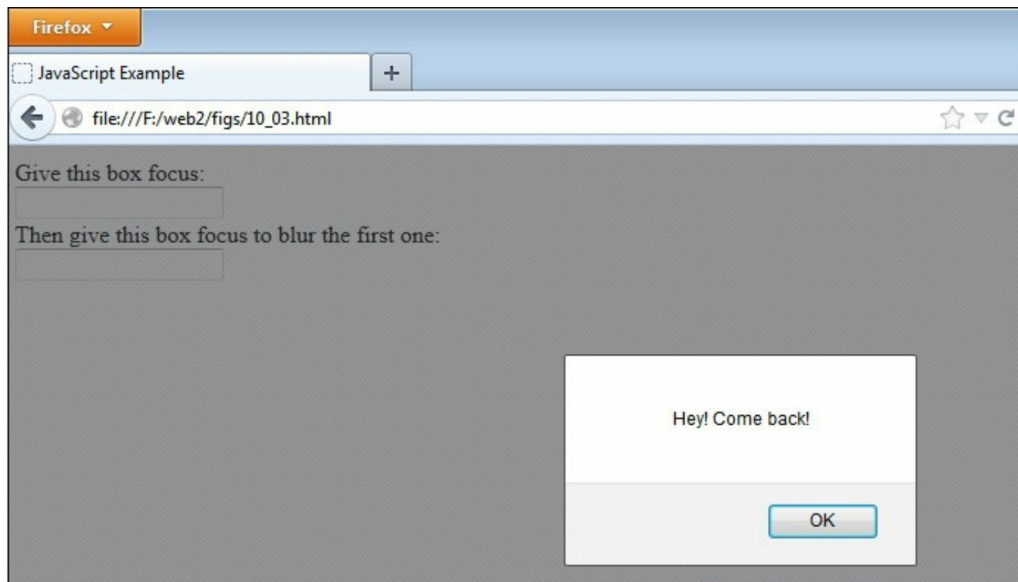
Notice that the onload event handler is added in the body tag

When the page has finished loading, viewers will get an alert that tells them it is finished.

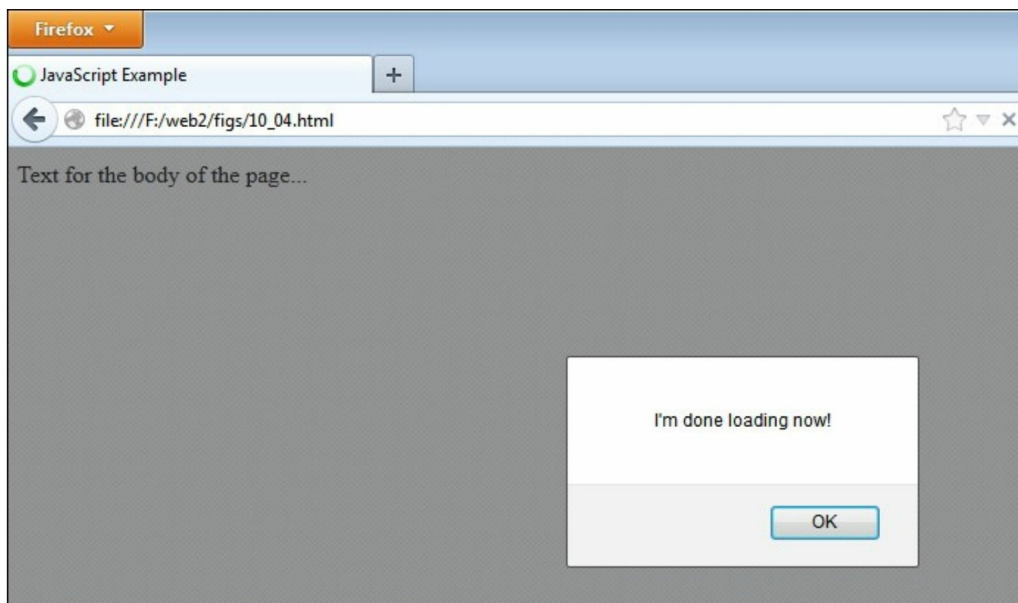
shows how the preceding code example would appear in the browser.

If you want to use the onload event handler in the script code rather than in the body tag, you could write this code into an external JavaScript file (save it as load_alert.js):

```
window.onload = function() {
  window.alert('I'm done loading now!'); };
```



This alert pops up when the viewer takes the focus off a text box.



This is displayed in the browser window after the page has finished loading.

NOTE

Only one load event can be specified in your document. If there is more than one, only one of them will work. This could happen by adding onload to the <body> tag, using addEventListener (mentioned later in this chapter), and/or using the preceding window .onload method at the same time, so be careful when you add a load event to check for any others.

This will capture the load event for the current window (the window object

will be explained in more detail in Chapter 11). You can now access all of the elements in the HTML code, and you can now remove the event handler from the opening `<body>` tag, as shown here:

```
<body> <script type="text/javascript" src="load_alert.js"></script> Text for the body of the page...
</body>
```

NOTE

Once the load event occurs, you can no longer use `document.write()` to write to the document.

The unload event occurs when a viewer leaves the current Web page. The viewer could leave by clicking a link, typing another address in the browser, or closing the window.

This event is known to annoy viewers, because it enables the site owner to do something while visitors are trying to move on to another page or another Web site (forcing them to wait). To have an alert pop up when the user leaves the page, you could write the following code:

```
⌞ <body onunload="window.alert('Be sure to come back, OK?');"> Other HTML code goes here...
    </body>
The onunload event handler is added to
the body tag
```

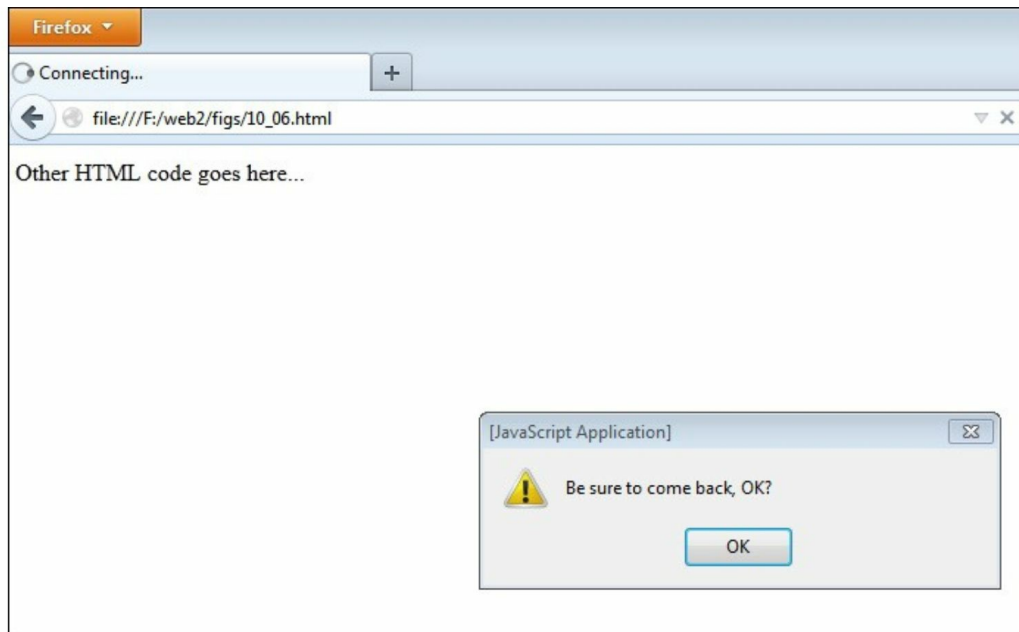
Figure 10-5 shows the result of the preceding script. As viewers try to leave the page that contains this script, an alert pops up telling them to be sure to come back. Of course, this could cause a viewer to become quite inconvenienced if it is used on an index page or on a number of pages within a Web site.

As with the onload event handler, you can use the onunload event handler in the script code rather than as an attribute of the body tag by using `window.onunload`.

Overall, be sure to think twice before using the unload event on a live page, because it will almost surely annoy most Web users.

TIP

If `onunload` does not work properly, try using `onbeforeunload` if you need this functionality.



This alert pops up when the viewer tries to leave the page.

The Reset and Submit Events

Reset and submit events are used when one or more forms are included in a document. A form can have its contents reset or can be submitted for processing. These events allow you to program a response to one of these form actions.

The reset event occurs when a viewer uses a form reset button to reset the form fields in a form. The reset event also has a related method called `reset()`, which is covered in Chapter 14. The submit event occurs when the viewer submits a form. The submit event also has a related method called `submit()`, which will be covered in Chapter 14.

To see the submit event at work, you have to create a form that can be submitted with a submit button. The following code will give a “Thank You” alert to the viewer once the submit button is clicked:

```
← <form onsubmit="window.alert('Thank You');"> onsubmit used in the opening form tag
  What's your name?<br>
  ← <input type="text" id="thename"><br>
    <input type="submit" value="Submit Form"> The submit button triggers the
                                          submit event when it is clicked
</form>
```

The submit event doesn't do you much good now (especially with the

contents of the form not really going anywhere), but this event will become more useful when you get to form validation in Chapter 14.

The Mouse Events

There are a number of events that are available for user actions involving the mouse. These are listed below.

- **mousedown** Occurs when a viewer presses the mouse button down but before the click is complete (the button has not yet been released).
- **mouseup** Occurs when the viewer releases the mouse button after pressing it down.
- **mouseenter** Occurs when a viewer moves the mouse cursor over an element (excluding child elements).
- **mouseleave** Occurs when a viewer moves the mouse cursor away from an element (excluding child elements).
- **mouseover** Occurs when a viewer moves the mouse cursor over an element.
- **mouseout** Occurs when a viewer moves the mouse cursor away from an element.
- **mousemove** Occurs when the viewer moves the mouse cursor.
- **mousewheel** Occurs when the viewer scrolls the mouse wheel up or down.

An example of using a mouse event can be shown by creating a text link. When you add the onmouseover event handler to the link, you have the option to perform JavaScript commands when the viewer passes the cursor over the link. Thus, if you want an alert to pop up when the viewer moves the mouse over a link, you could code something like the following:

```
<a href="http://www.pageresource.com"
  onmouseover="window.alert('I told you not to try to click me!');" > Don't Try Clicking Me!</a>
```

The onmouseover event handler in a link tag

This time the visitor doesn't even get to click the link before being greeted with an alert. Keep in mind that a script like this could annoy your visitors if it is overused. Figure 10-6 shows the result of this script in a browser. The alert pops up as soon as the mouse cursor moves over the link.

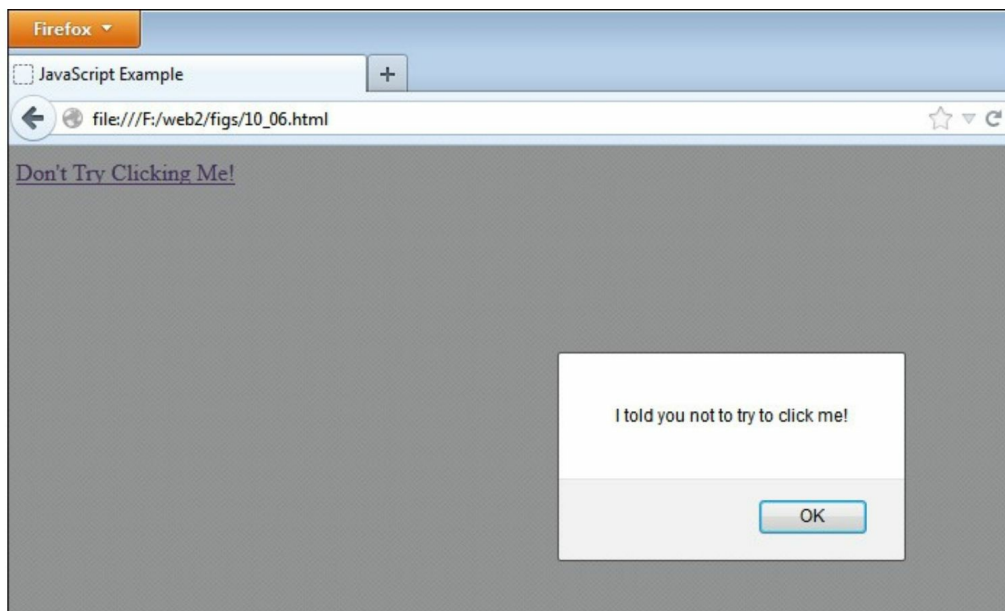
Since there is no need to click the link for something to happen, the

browser won't try to follow the link afterward. In fact, with this script in place, it is impossible to click the link at all because the alert keeps popping up when you move your mouse over to click it!

The Keyboard Events

Keyboard events occur when the user interacts with a Web page using the keyboard. These allow you to react to each phase of a key being pressed and released by the viewer. The keyboard events are listed here:

- **keydown** Occurs when the viewer presses down a key on the keyboard.
- **keypress** Occurs when a viewer presses down a key on the keyboard and the corresponding character is typed. This occurs between the keydown and the keyup events.
- **keyup** Occurs when the viewer lets go of a key on the keyboard, releasing the key.



This alert pops up when the mouse cursor moves over the link.

For example, if you wanted to send an alert each time a character was typed into a text box, you could use the following code:

```
<input type="text" id="typing" onkeyup="alert('You typed a character!');">
```

This would remind the viewer each time a character is typed into the text

box. This can get quite annoying after a couple of characters are typed, but once you learn about the event object later in this chapter, you will learn how to find out what key was pressed, which will give you more flexibility with the keyboard events.

Try This 10-1

Focus and Blur

pr10_1.html
prjs10_1.js

In this project, you will use your knowledge of the focus and blur events and innerHTML (from Chapter 9) to create a script that will remind a user what to type into a text box.

Step by Step

1. Create an HTML page and save it as pr10_1.html. Add the necessary script tags to point to an external JavaScript file named prjs10_1.js. Add code so that there is a text box for a phone number with an id of “pn” and a text area for an address. Beside the phone number text box, there will be an empty span tag with an id of “reminder”. The body section of the HTML file should look like this when complete:

```
<form> Phone Number:  
<input type="text" id="pn"> <span id="reminder"></span><br><br>  
<textarea></textarea>  
</form>
```

2. Create an external JavaScript file and save it as prjs10_1.js. Use this for step 3 and step 5.
3. Get the “pn” and “reminder” elements by their ids and assign them to the variables pn and rm, respectively.
4. When pn receives focus, change the innerHTML of rm to the text “Format: 123-456-7890.” When pn is blurred, change the innerHTML of rm to an empty string (“”).
5. The JavaScript file should look like this when complete:

```
let pn = document.getElementById("pn"),    rm =  
document.getElementById("reminder");  
pn.onfocus = function() {  
    rm.innerHTML = "Format: 123-456-7890";  
};  
pn.onblur = function() { rm.innerHTML = "";  
};
```

6. Save the HTML and JavaScript files, and load the HTML page in your

browser. Click inside the text box and a reminder message should appear next to it. Click the text area afterward to blur the text box and make the message disappear.

Try This Summary

In this project, you used your new skills, the focus and blur events, to create a script that could help a user fill out a form.

Ask the Expert	
Q: Why are there so many events? A: that you end up with a bunch of possible events.	There are so many things that a viewer (or the browser itself) can do while on a Web page <i>(continued)</i>

Q: Do I need to memorize all of these events?

A: You probably only need to memorize them if you are taking a test, or you are doing this for a job and need to know things quickly, or if you just like knowing the events off the top of your head; otherwise, you can just refer to Table 10-1 in this chapter if you are not sure which event needs to be used.

Q: Will I be using every single event in this book while doing the projects in this book?

A: Since this book is a beginner’s guide, you will not get to the point where you use every single event.

Other Ways to Register Events

Up to this point, you have been using what are called DOM Level 0 event

handlers (onclick, onmouseover, and so on). There are two other methods for registering events in addition to the method you have used: `addEventListener()` and `attachEvent()`. These are called DOM Level 2 event handlers, and they offer the ability to attach multiple events to elements, whereas the DOM 0 method only allows one event to be registered on any given element. For example, suppose you had the following JavaScript code:

```
let mydiv = document.getElementById("mydiv");
mydiv.onclick = function() { alert("First Click!");
}
mydiv.onclick = function() { alert("Second
Click!"); }
```

In this case, the second instance of assigning the onclick event handler to the mydiv element will overwrite the first one, so only the “Second Click!” alert will be displayed when a click occurs. DOM Level 2 event handlers allow you to avoid this issue.

NOTE

If you have `<div id="mydiv" onclick="...">` and then have `mydiv.onclick` in the document, it will overwrite the onclick attribute, so make sure to choose a single method of adding the event to avoid any conflicts.

The addEventListener() Method

The `addEventListener()` method is the standard from W3C, and is currently supported in all the most recent browsers (Internet Explorer needs to be version 9+). It allows you to specify

an event, a function to execute for the event, and a value of true or false depending on how you want the event handler function to be executed in the capturing (true) or bubbling (false) phase. The general format looks like this:

```
element.addEventListener("event_type", function_name, true_or_false);
```

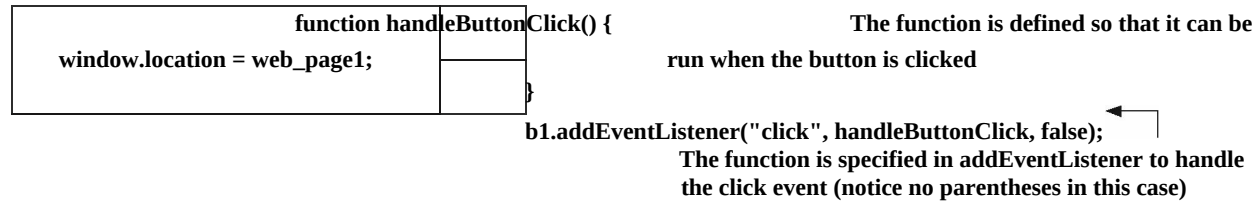
Thus, if you want to create a linked input button as you did earlier in this chapter, you could adjust the JavaScript code to look like this:

```
let web_page1 = "http://www.yahoo.com"; let b1 =
document.getElementById("btn1"); b1.addEventListener("click", function() {
    window.location = web_page1; }, false);
```

Notice that rather than using the event handler, this method uses the name of the event (instead of onclick, you simply use click). Also, this method will accept a function name or a function expression to handle the event (as shown previously). It is important to note that when you specify a separate

function, you need to leave off the parentheses. For example, look at the following code:

```
let web_page1 = "http://www.yahoo.com"; let b1 = document.getElementById("btn1");
```



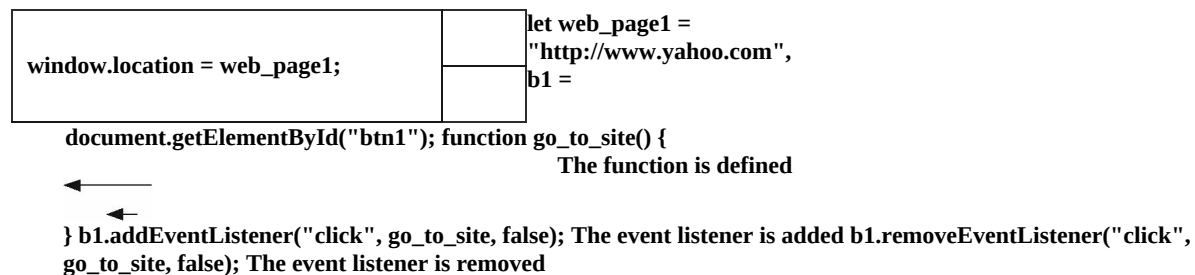
A common bug that will cause this not to work is including the parentheses when specifying the handling function in addEventListener. Therefore, if you find unexpected behavior, check to be sure parentheses were not added.

Removing an Event

To remove an event, you would use the removeEventListener() method:

```
element.removeEventListener("event_type", function_name, true_or_false);
```

Note that if a function expression is used in the addEventListener() method, the event registration cannot be removed (the function expressions will be seen as two different arguments, even if they use exactly the same code). If there is a chance you will need to remove the event later, it is best to use a separate function and use the function name as the second argument in both method calls. An example of this is shown here:



Since this is the standard method for registering events in modern browsers, you will use this method of event registration for the remainder of this book.

Capturing and Bubbling Phases

When you have elements inside other elements that both have the same event type registered to them, which event occurs first will depend on whether the capturing or bubbling phase is used to register the events. If

capturing is used, then the outermost element's event occurs first and the innermost element's event will occur last. If bubbling is used, the opposite is the case. It is most common to use the bubble phase, since it has the greatest browser compatibility.

The `attachEvent()` Method

The `attachEvent()` method works in a similar way to `addEventListener()`. However, it only works with event bubbling and is only useful in versions of IE below 9, so it's referenced here in case you need to support an older IE version.

```
element.attachEvent(event_handler, function_name);
```

The Event Object

When an event occurs, an object named *event* is created, which stores information about the event. This information can be used by your script to help it perform the required actions. For example, a keyboard event will have information about the key that was pressed stored in the event object. This allows you to program different responses when different keys are pressed.

DOM and Internet Explorer: DOM Level 0 Registration

In most browsers, the event object is accessed by using the name *event*. It can be passed as the lone argument to a function that handles the event, as in the following DOM Level 0 code:

```
let b1 = document.getElementById("btn1"); b1.onclick = function(event) {  
    alert("The " + event.type + " event started this!");  
};
```

Notice that the argument *event* is passed to the event listener. The *type* property of the event object (which returns the type of event) is then used to display an alert to the viewer, which will let it be known that the click event is what caused the function to run. The argument can be any identifier you like, so it is common to see shorter identifiers such as *e* or *evt* as opposed to *event*. Use whatever is comfortable for you or what your organization has standardized. As long as you are consistent, any identifier is perfectly valid here.

Internet Explorer, however, uses *window.event* to access the event object, which means the DOM Level 0 event handler would use the following code:

```
let b1 = document.getElementById("btn1");
b1.onclick = function() {
  alert("The " + window.event.type + " event started this!"); };
```

With DOM Level 0 functions, the event object can be implemented cross-browser by assigning the needed value, based on what the browser supports, to a variable, as in the following code:

```
let b1 = document.getElementById("btn1"); b1.onclick = function(event) {
let e = event || window.event;
  alert("The " + e.type + " event started this!"); };
```

Notice that the variable is assigned the event object if it is available, or the window.event object if not. The use of the logical OR (||) allows you to provide a preferred value if it is available, or to set another (default) value if not. Now that the variable e will hold the proper value cross-browser, it can be used to alert the event type.

Using event with Modern Event Registration

When using addEventListener() or attachEvent(), the event object is accessible via *event*. This allows you to use the event object as expected. For addEventListener(), you could use the following code:

```
let b1 = document.getElementById("btn1"); b1.addEventListener("click", function(event) {
  alert("The " + event.type + " event started this!"); }, false);
```

For attachEvent(), you could use the following code:

```
let b1 = document.getElementById("btn1"); b1.attachEvent(onclick, function(event) {
  alert("The " + event.type + " event started this!"); });
```

Support for older browsers, if needed, can be achieved using the DOM Level 0 method. Again, you can refer to the resources mentioned in the “The attachEvent() Method” section earlier in the chapter, as they have been coded to handle the various event registration models.

Properties and Methods

You have already seen the type property, but the event object also has other properties and methods that are useful, which are listed.

As you will notice, some of the listed properties work with Internet Explorer (prior to version 9), while the others will work with other modern browsers. Since most modern browsers (including Internet Explorer 9 and above) support the standard DOM Level 2 event registration model, you will use this method through the remainder of this book. Notes will be added for cases where you may need to know how to

perform a task in Internet Explorer prior to version 9.

Property/Method	Description
bubbles	Whether or not the event bubbles.
cancelable	Whether or not the default action of the event can be canceled.
cancelBubble	Cancels event bubbling when set to false (Internet Explorer 8 and earlier).
currentTarget	The element that is currently handling the event.
defaultPrevented	Whether or not preventDefault() has been called.
detail	Additional information about the event.
eventPhase	The phase in which the event handler was called: 1 = capturing, 2 = at target, 3 = bubbling.
preventDefault()	Prevents the default action of the event from occurring.
returnValue	Prevents the default action of the event from occurring (Internet Explorer 8 and earlier).
srcElement	The element that is the target of the event (proprietary but supported). Typically, target (see later in the table) is used.
stopImmediatePropagation()	Ends all capturing and bubbling on the event, and stops other event handlers from being called.
stopPropagation()	Ends all capturing and bubbling on the event.
target	The element that is the target of the event.
trusted	Whether or not the event was initiated by the browser or the programmer.
type	The event type (for example, click, mouseover, and so on).
view	The window object where the event happened.

The Properties and Methods of the event Object

Preventing the Default Action

You will recall that when using the DOM Level 0 registration, preventing the default action on a link was achieved using a return statement (return false). However, you can do this without the need to return immediately by using the preventDefault() method of the event object.

For example, the following code can now be used to prevent a link click from loading the specified page:

```
let nlink = document.getElementById("notlink"); nlink.addEventListener("click", function(event) {
    event.preventDefault();
    alert("This link doesn't go anywhere!");
    // Further Code...
}, false);
```

As you can see, you can continue handling the event, rather than using a return statement to leave the function.

NOTE

When using Internet Explorer prior to version 9, you can use the `returnValue` property. Setting its value to `false` (for example, `window.event.returnValue = false;`) will do the same thing as `preventDefault()`.

Event Information

Some events, such as keyboard events, have additional properties that are not shown in the table (which are available to all events).

For example, when a keyboard event occurs, the key code of the key that was pressed is stored in `event.keyCode`, which allows you to determine which key was pressed. This can be used by your script to react only when a particular key is pressed. For example, you could use this code:

```
let text = document.getElementById("textbox1"); text.addEventListener("keyup", function(event) {
  if (event.keyCode === 80) {
    alert("Please do not press the p key!");
  }
}, false);
```

This will alert the viewer that you do not want the p key to be pressed, since 80 is the key code for the p key on the keyboard. You can see more on key codes at: [http://protocolsofmatrix](http://protocolsofmatrix.blogspot.com/2007/09/javascript-keycode-reference-table-for.html)

[.blogspot.com/2007/09/javascript-keycode-reference-table-for.html](http://protocolsofmatrix.blogspot.com/2007/09/javascript-keycode-reference-table-for.html)

Try This 10-2

Using `addEventListener()`

pr10_2.html
pr10js10_2.js

In this project, you will use `addEventListener` to register a `mouseover` and `mouseout` event on a piece of text.

Step by Step

1. Create an HTML page and save it as `pr10_2.html`. Add code so that it links to an external JavaScript file named `prjs10_2.js`. The body section of the HTML code should look like this:

```
<p>
This is very <span id="int">interesting</span>! </p>
```

2. Create a JavaScript file and save it as `prjs10_2.js`. Use it for step 3.
3. When the mouse is moved over the “int” span, change the color to `#FF0000`. When the mouse moves away, change the color to `#000000`.

When complete, the JavaScript file should look like this:

```
let int = document.getElementById("int"); int.addEventListener("mouseover", function() {
  int.style.color = "#FF0000";
}, false);
int.addEventListener("mouseout", function() {
  int.style.color = "#000000";
}, false);
```

4. Save the JavaScript file and open the HTML file in your browser. Move the mouse over the text and off the text to change colors.

Try This Summary

In this project, you used your skills with event handlers and `addEventListener()` to change the color of text when the mouse moves over or away from it.

Creating Scripts Using Event Handlers

Now that you have tackled the long list of event handlers and the event object, it's time to have a little fun. In this section, you are going to learn how to do things other than sending alerts.

Show Hidden Content

For this script, create an HTML file with the following code inside the body section and save it as `news.html`.

```
<div>
<strong>Breaking News Story!!!!</strong>
<a href="story.html" id="morelink">More...</a>
</div>
<div id="morediv">
Today, there was a major development in a story that we have been covering here at my site. We
will post later with further details. Hurry back!
</div>
<script type="text/javascript" src="news.js"></script>
```

Here, you have a “breaking news story” headline. A link (with an id of `morelink`) is placed so that when clicked, it will display the content of the story. Below this, you have a div (with an id of `morediv`) that contains the content of the story. At the moment, it is perfectly visible, since no CSS or JavaScript has been applied to change it. Your goal is to hide this content initially, and then display it when the user activates the link.

Next, create a JavaScript file and save it as `news.js`. The first thing you will want to do is to define some variables that will grab the two needed elements: the `morelink` and `morediv` elements.

```
let mlink = document.getElementById("morelink"),    mdiv =
document.getElementById("morediv");
```

These are now assigned as objects to the `mlink` and `mdiv` variables. Next, you will need to add an event listener that will allow you to show or hide the

mdiv element when the mlink element is clicked, as in the following code:

```
mlink.addEventListener("click", function(event) {  
    event.preventDefault(); mdiv.style.display = (mdiv.style.display === "none") ? "block" :  
    "none";  
}, false);
```

Notice that the event object is passed as an argument, allowing `preventDefault` to be used. The `display` property of the `mdiv` element is changed to “block” if it is “none”, or to “none” if it is anything other than “none” (for example, when it has been set to “block” from a previous change).

Now everything is set for the click event, except that you need to hide the `mdiv` element for the initial page display. This could be done in CSS, but here you will use JavaScript to simply set its `display` style to “none”:

```
mdiv.style.display = "none";
```



The initial view of the page, with only the headline and link displayed

The full code for the JavaScript file now looks like this:

```
let mlink = document.getElementById("morelink"), mdiv =  
document.getElementById("morediv"); mlink.addEventListener("click", function(event)  
{  
    event.preventDefault(); mdiv.style.display = (mdiv.style.display === "none") ? "block" :  
    "none";  
}, false); mdiv.style.display = "none";
```

Save the JavaScript file and open the HTML file in a Web browser. You should see the headline and the link, with no additional content. Figure 10-7 shows what the initial page should look like.

Click the link. The story content should now display below the headline and link. Clicking again should hide the story, which you probably will want to do after reading a teaser headline that simply leads to another teaser telling you to check back soon! Figure 10-8 shows the result of clicking the link to display the content.

Change Content

Instead of showing and hiding one piece of content, you may want to give the viewer several options and display the chosen content when a particular link is clicked. To do this, you will

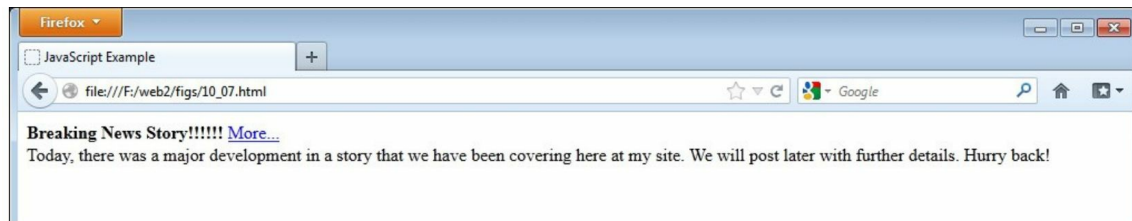


Figure 10-8 Once the link is clicked, the content is displayed.

first need an HTML document. In this case, you will use the following HTML and save the file as `technews.html`.

```
<!doctype HTML>
<html>
<head>
<title>Selected Tech News</title>
<link rel="stylesheet" type="text/css" href="technews.css">
</head>
<div class="wrapper">
  <div class="headline">Nick Delivers Crushing Defeat to John</div>
  <div id="top_story_nav" class="story_nav">
    <a href="#sumdiv" id="sumlink">Summary</a>
    <a href="#statsdiv" id="statslink">Stats</a>
    <a href="#comdiv" id="comlink">Comments</a>
  </div>
  <div id="sumdiv" class="spart">
```

Earlier today, we attended a JavaScript programming competition, where we watched a first round showdown between Nick and John. In the beginning, it looked like this might be a close battle, but Nick got the upper hand quickly and never looked back.

<p>John started writing some strange code and put everything into the global scope, while Nick was busy building objects, using inheritance, and optimizing his code until he had a very slick, clean application. John's app, on the other hand, just seemed clunky and buggy in comparison.</p> <p>Perhaps it was an off day for John, but now he can sit back and watch as the remaining coders go toe to toe. We will write up another report once we see the next showdown!</p>

```
</div>
<div id="statsdiv" class="spart">
App. Loading Speed: Nick 0.002 ms, John 3043.231 ms
<p>Globals: Nick 1, John 137</p>
<p>JS File Size: Nick 0.5K, John 1.2 MB</p>
<p>Final Score: Nick 15298, John: 12</p>
</div>
```

```
<div id="comdiv" class="spart"> H311oW0rID: Is it any wonder John lost with all of those
globals? Really, John? REALLY?
```

```
<p>Anne: I saw this coming. Honestly, Nick is just better.</p>
```

```
<p>Paul: John, I know the challenge had to do with balloons, but to even be in this competition you should know
better than to overwrite the pop()
method on the Array prototype! Wow!</p>
```

```
<p>Sup3rCod3r: This was a schooling. I hope John learns from this...</p>
```

```
<p>Con5pir@cy: I think the competition is rigged... can't you all see it?</p>
```

```
<p>Doug: @Conspir@cy or whatever- See what? It looked to me like one was just a better programmer than the
```

```

other. John actually used this line:  while (1 == true) { alert('True!'); } ! The judges had to reboot!</p>
<p>Christie: Yikes! How did John fall into that trap?!</p>
<p>Z25S1l0O: Has the video been posted yet? I've gotta see this! lol!</p>
</div>
</div> <script type="text/javascript" src="technews.js"></script>
</body>
</html>

```

Notice that there is a wrapper div, which you will use to contain all of the content. Within it, you have the headline div, the top_story_nav div, and the three content divs (sumdiv, statsdiv, and comdiv). The script will allow each link to display one of the content divs while hiding the other two. This way, the user can view one section at a time rather than scrolling to the desired section.

The next thing you will do is add a little style to the page by creating a CSS file. Include the following code and save the file as technews.css.

```

.wrapper { width:590px; height:390px; padding:5px; overflow:scroll;          font-size:1em; font-family:Verdana, sans-
serif; }
.headline { font-weight:bold; font-size:1.8em; font-family:Arial,          sans-serif; background-color:#6699CC; text-
align:center; } .story_nav { margin:0 0 10px 0; padding:2px; background-color:#DDD;
word-spacing:20px; text-align:center; } .spart { margin-bottom:10px; }

```

This defines the size of the wrapper div, which will be 600×400 pixels when the padding is included. It allows content that is longer to be scrolled by the user within the wrapper. The other styles provide some color, margins, padding, and spacing for the other elements. You can of course adjust these to fit your preferences.

Finally, you will need to create the JavaScript file. Include the following code and save the file as technews.js.

```

let slinks = [document.getElementById("sumlink"),
document.getElementById("statslink"),      document.getElementById("comlink")],  sdivs =
[document.getElementById("sumdiv"),        document.getElementById("statsdiv"),
document.getElementById("comdiv")];
function change_div(count) {
  slinks[count].addEventListener("click", function(event) {
    event.preventDefault();  sdivs[count].style.display = "block";  for (let j = 0; j
< sdivs.length; j++) {
    if (j === count) {      continue;
    }   else {
      sdivs[j].style.display = "none";
    }
  }
}, false);
}
for (let i = 0; i < slinks.length; i++) {
  change_div(i);
}
sdivs[1].style.display = "none"; sdivs[2].style.display = "none";

```

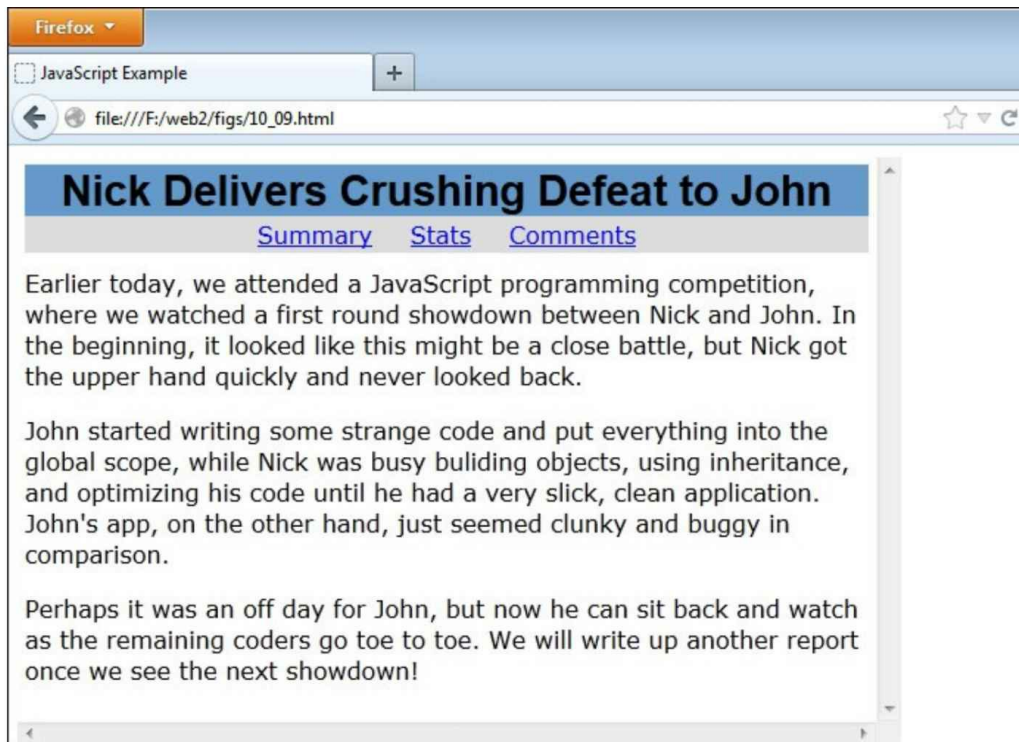
The three links correspond to the three content divs, and so the elements from

both sets are placed into arrays (slinks and sdivs). The `change_div()` function will be explained shortly, but first notice the for loop after the function. This cycles through the slinks array and calls the `change_div()` function for each link, passing the value of `i` along as an argument. This argument will be used to register the click event for each element and to change the appropriate content when each link is clicked.

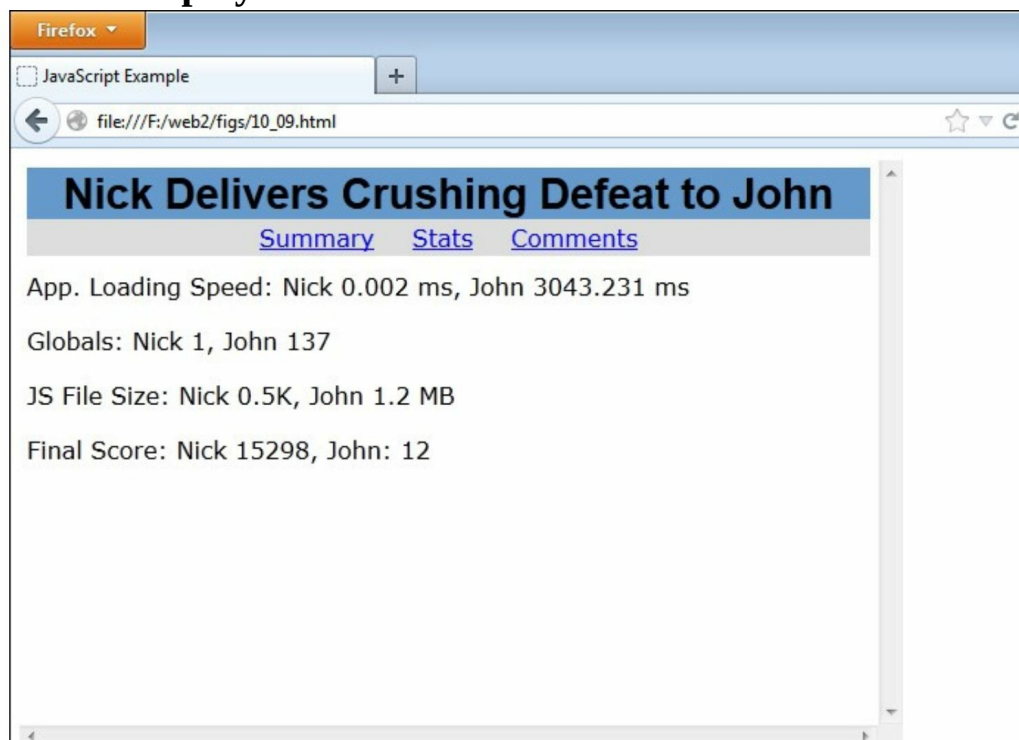
The `change_div()` function takes in the passed count variable from the for loop as an argument. The first time through the loop, `change_div` will be sent a value of 0, the second time a value of 1, and so on. Since the links and content divs are in arrays, this allows you to use the count value to access the desired elements. The function begins by using `addEventListener()`. The listener is added to the link element found in the slinks array at index *count*. In other words, `slinks[0]` will be accessed when count is 0. This accesses the first link element, since the value of `slinks[0]` is `document.getElementById("sumlink")`. Since this loop is run three times, the event listener will be added with the count values of 0, 1, and 2.

The event listener function first prevents the default action. Then, depending on the link that was clicked, the corresponding div element in the sdivs array is displayed by setting its style display to "block". A loop is used to hide the remaining two div elements. Notice that if the value of `j` is equal to the count value (the element being shown), then it is skipped so that the element currently being shown remains visible.

Finally, at the end of the script, you will see that the `statsdiv` and `comdiv` elements are hidden. This initializes the page with the `sumdiv` element still visible by default while the other two content shows the initial display of the page when viewed in a browser.



The initial display. Notice the links available for the user to click.



The result of clicking on the "Stats" link

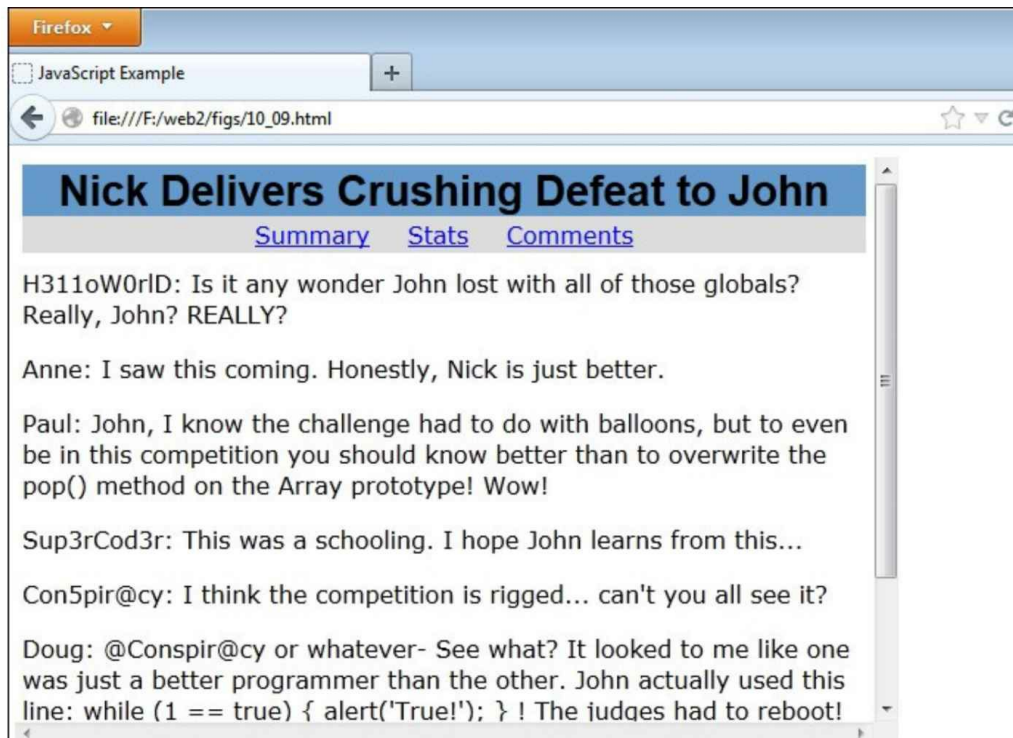
Clicking on the “Stats” link will show the statistics (makes statsdiv visible and the others hidden). Figure 10-10 shows the result of clicking the “Stats” link.

Finally, clicking the “Comments” link will display all of the comments that were directed at me for my embarrassing showing at that programming competition! Figure 10-11 shows the comments being displayed.

This script combines many of the techniques you have learned (events, loops, arrays, and more). The arrays and loops were particularly helpful here as they kept you from typing three versions of `addEventListener()` with each value hard-coded into the event registration and function. When working with events, it is often helpful to have these handy ways of typing less code.

Custom Events

Modern browsers allow you the ability to create custom events that can be fired (dispatched) at a time of your choosing. Custom events can be helpful when creating custom functionality or for triggering an action to occur once data is finished loading from another source, such as an API call (AJAX will be discussed in Chapter 15).



The comments display when the “Comments” link is clicked.

To create a custom event, you can use the CustomEvent constructor, as shown in the following code:

name: "John"	

} custom data that can be retrieved when
});

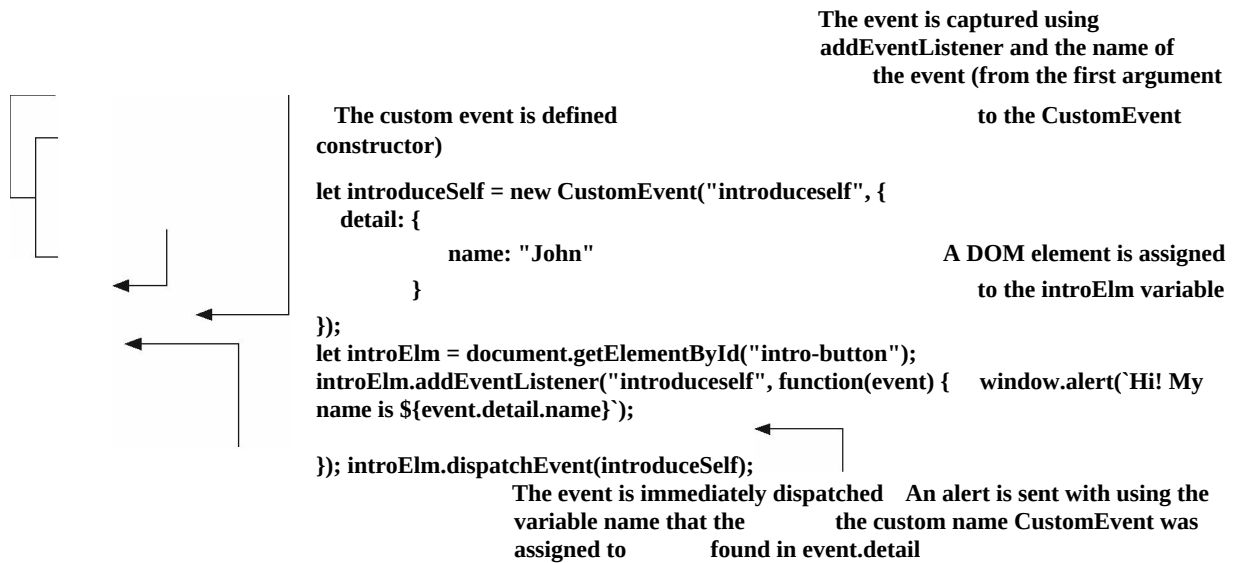
let introduceSelf = new CustomEvent("introduceself", {
detail: {
The detail property of the object for the
second argument allows you to define

the event is dispatched.

The CustomEvent constructor is used to create a new event named introduceself. The second argument is an object with event information.

Notice that the first argument is what the name of the event will be. This name can be used in the addEventListener method to handle the event. In this case, the name of the event is introduceself. The variable name the event is assigned to can be used to dispatch the event—in this case, introduceSelf (notice the capital S here).

With that in mind, the next steps for making use of the new event are to listen for it and to dispatch it so that the event listener will capture it and respond. The following code shows an example of this.



Notice that the custom event is defined and attached to an element from the DOM using `addEventListener`. Here, the event is dispatched immediately, but the last line in the example could be placed anywhere in your code in order to dispatch the event at the time of your choosing. Note that with a custom event, you have to use the identifier `detail` in order to pass along the additional information. Using another identifier won't work, as in the following code:

```
let introduceSelf = new CustomEvent("introduceself", {
  name: { first: "John" }
});
```

Using the identifier name instead of `detail`; detail at the top level will cause the details to not be passed along properly.

7 INTRODUCTION OF NODE.JS

Key Skills & Concepts

- Introducing Node.js
 - Installing Node.js
 - Using Node Modules
 - Installing a Database
 - Creating a Web Server
-

So far in this book, you have been using browser-based JavaScript in order to add functionality to a Web page. In recent years, Node.js has come along and expanded the

number of places where JavaScript can be used. Not only can you use it for browser-based applications, you can now use it for applications on a local machine or to create a Web server and use it for server-side applications.

This chapter will introduce Node.js and give you some basic information on how you can use it to create local applications. In addition, you will be introduced to a simple database that allows you to store data that can then be used in your applications.

NOTE

Although this book provides some beginning information on Node.js and database usage, it will not get into all the details needed for you to be an expert at each, as they are beyond the scope of a JavaScript beginner's guide.

Introducing Node.js

Node.js was originally created with the idea of allowing real-time applications to have push capability—meaning that the server could send information to the client without the client needing to request information from the server first. This two-way communication allows for data to be shared in real time, which makes Node.js a great environment for Web applications that involve chat, stock market information, collaboration, or even games.

While Node.js is excellent for real-time applications using [Socket.io](https://socket.io/), it can also be used to create a standard Web application and to respond to requests for information via an application programming interface (API). It's nonblocking I/O is great both for real-time and standard applications. In this chapter, you will see the basics of using Node.js, which includes the creation of a basic Web server. With this in place, you will begin creating a small Web application that will be built upon throughout the remainder of the book, as needed. In this way, you can see how JavaScript can be used for server-side development in addition to the standard client-side development that has been done using JavaScript for a long time.

Q: You mentioned Socket.io A: Socket.io https://socket.io/. Q: A: Q: A: more information at development.	

Installing Node.js

Since Node.js runs in its own environment (rather than in a Web browser), it needs to be installed so that you can use it to create scripts or applications. First, you need to be sure you do not have Node.js installed already. If you don't, or if you want to update your Node.js version, you can install the latest Node.js for your system.

Check for a Current Installation

To see if Node.js is installed system-wide already, you can open a command prompt on your system and type a command. Opening a command prompt differs a bit for each operating system. Here are some tips for Windows, Mac, and Linux.

Windows Command Prompt

In later versions of Windows, you can simply type **cmd** into the search bar and click on Command Prompt when it displays.

In older versions, these options are available:

- Click Start | All Programs | Accessories | Open Command Prompt.
- Click Start | Run, enter **cmd**, and click OK.

Mac Command Prompt

For macOS, click the Finder icon in the dock and click Applications in the favorites list. Find the Utilities folder and click to open the folder. Find the Terminal icon and double-click to open the Terminal program.

Alternatively, press command-spacebar to launch Spotlight for searching and then type **Terminal**. Double-click the Terminal search result to open the Terminal program.

Linux Command Prompt

The terminal can have different names, depending on the flavor of Linux you use. For example, the following link has some tips for the different type of Ubuntu desktops: <https://help.ubuntu.com/community/UsingTheTerminal?action=show&redirect=BasicCommands>.

Running the Command

Once you have a command prompt open on your system, type the following command and then press enter:

```
node -v
```

If Node.js is already installed, the version of Node.js installed is displayed. Otherwise, an error message will display indicating that node is not a recognized command on your system.

Install Node.js

If you do not have Node.js or want to update your version, you will want to install Node.js on your system.

If you want to install Node.js by running a file (default), visit <https://nodejs.org/en/download/>.

The options will be displayed for various operating systems, as shown in Figure 11-1.

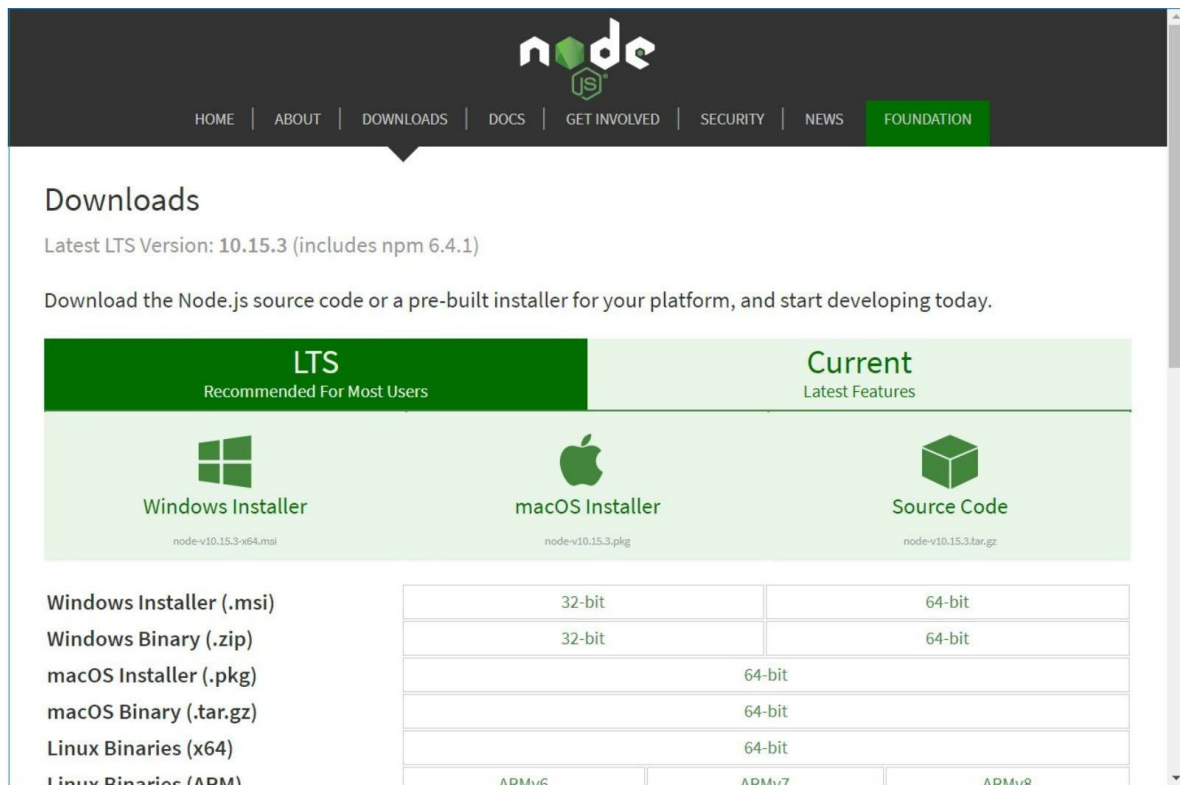
Download the installer for your system and follow the prompts to install Node.js on your system. The installer should do the work of installing Node.js for you, and once it's done, you should be ready to try Node.js!

Alternately, you may wish to install Node.js on your system with a

package manager. You can find instructions on how to do this at <https://nodejs.org/en/download/package-manager/>.

Write a “Hello World” Script

Now that you have Node.js installed, you can write a simple “Hello World” script and then run it from the command line. First, create a folder (or use one you already have) on your machine



The Node.js

for saving your project files. This folder will be used for creating the examples and projects in this book.

Create a new JavaScript file and simply place the following line of code in it:

```
console.log("Hello there!");
```

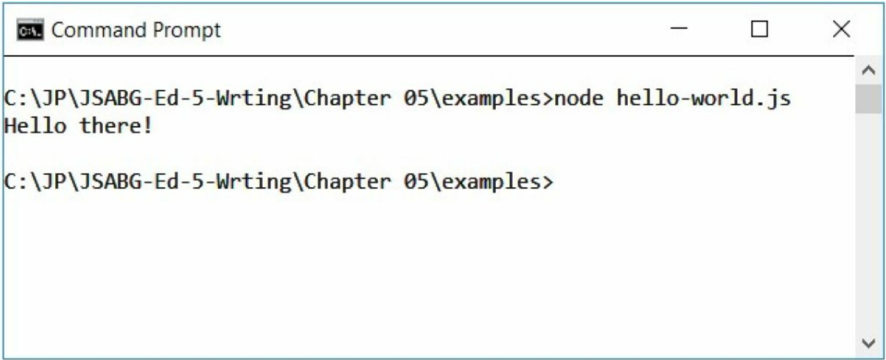
Save this file as hello-world.js in your project folder. To see the script in action, you will need to open a command prompt in your project folder (as described earlier in this chapter).

At the command prompt, type the following command:

```
node hello-world.js
```

Press enter and you will see the result. The text “Hello there!” will be written into the command window (console) and then you will be sent back to the command prompt. Figure 11-2 shows how the command window looks once the script is complete.

As you can see, the console.log command simply logs a message to the console. Much like what you have done in previous chapters with document.write, you simply pass a string value to the console.log command in order to have that string value logged to the console.



The result of running the Hello World script

NOTE

The console.log and some other console commands are also available when you are writing browser-based (front-end) JavaScript. You will see more about using these for the browser in Chapter 13.

While the default console.log command is useful for some quick logging, you can use a Node module to add different levels of logging, such as debug, warning, error, and so on. You will learn more about logging and installing Node modules as you proceed in this chapter.

Q: A:	
..	

Windows

- **Mac**
- **Linux**

Q: I was able to install Node.js on Windows, but Windows still didn't recognize it when I tried to run the Hello World script. What should I do?

A: In some cases, you may need to check that Node.js was set in your Windows path. This is typically done for you with the installer file, but if not, you can add it to your path manually instead. For information on adding a program to the Windows path, see <http://helpdeskgeek.com/windows-10/add-windows-path-environment-variable/>.

Q: The script I wrote was for the command line. Aren't we going to set up a Web server?

A: Yes, later in this chapter you will set up a Web server. In the meantime, there are some things you can do with command-line scripts that may prove helpful over time, and you will look at these more as this chapter progresses.

Using Node Modules

Node modules allow you to extend the default functionality available to Node.js scripts. Some Node modules are available without the need to install them, while others require that you install them before they are available for use.

Using Native Node Modules

You can use a native Node module in a script by simply including it in your JavaScript file. A commonly used native module is the File System module (fs). You can include it by requiring the module and assigning the

result to a variable (usually the variable name it is assigned to matches the module name). For example, here is how you could include the fs module in your Hello World script:

```
const fs = require("fs"); console.log("Hello there!");
```

The fs module is included with a require statement.

This requires the fs module but doesn't make use of it. Much like the commands you have used previously, fs has certain methods that can be called. As an example, the mkdir method of fs allows you to make (create) a directory (folder) on your local machine. This is one of a number of methods you can use to work with files and folders. The following code shows the inclusion of the fs module and the use of fs.mkdir as an example. Save this file as create-folder.js in your project folder.

```
const fs = require("fs");
if (err) {
  throw(err);
} else {
  console.log("Directory was created!");
}
```

The fs module is included with the require statement.

The fs.mkdir method is called.

The first argument is the name of the folder to be created, and the second is a function to run fs.mkdir("test-folder", function (err) {

once the

throw(err); An if-else statement determines whether an error occurred during the creation of the folder. If there

was no error, the phrase "Directory was created!" is logged to the console.

Notice that the code begins with the inclusion of the fs module. Without this, the next line would throw an error indicating that fs is not defined. On the second line, the fs.mkdir method is called.

The fs.mkdir method takes two arguments. The first argument is the name of the folder you want to create (in this case, test-folder). This folder will be created in the same folder where your node script is located by default (in this case, within your project folder). You can alter this path using an absolute path, relative path, or by methods available on the path module (you will see more on the path module later in the chapter).

The second argument is a function that will be executed once the folder creation has completed. Here, an if-else statement is used to throw an error if there was an error creating the folder. Otherwise, the “Directory was created!” message will be logged to the console.

To run this, open a console in your project folder and type the following command:

```
node create-folder.js
```

Once it runs, you should be able to find the new folder (test-folder) inside your project folder! Note that you will need proper permissions on any folders you are dealing with for this to work, so be sure you have write permissions on your project folder on your system.

As you can see, a native module can be used pretty quickly by simply including it and finding which methods are available for use.

Asynchronous Execution

You may have noticed the following description of how the last example works:

“The second argument is a function that will be executed once the folder creation has completed.”

Node.js, by default, does almost everything asynchronously, meaning that your code can continue on to do other things while waiting for an action to complete. By going back to the previous example, you can make an alteration that shows how this works. Consider the following code:

<pre>if (err) { throw(err); } else { console.log("Directory was created!"); } }</pre>	<pre>const fs = require("fs"); fs.mkdir("test- folder", function (err) { The fs.mkdir method is called. The phrase "Directory was created!" is logged to the console, if successful, once the operation has completed. }); console.log("This can display before the success message does!");</pre>

This console.log comes after the fs.mkdir method is called, but will start executing immediately after the method is called and will not wait for the folder creation to be complete!

Save this file as test-node-async.js and run it from your console window by typing **node testnode-async.js**. Unless your OS is incredibly fast, the result should look like this:

This can display before the success message does! Directory was created!

Notice that the message does indeed display before the folder creation is complete.

NOTE

If test-folder already exists inside your project folder (and it will if you ran the last example), the script will error out when you run it because it will not re-create an existing folder. You can run it without error by deleting test-folder before running the example or by changing the name of the folder to be created in the script (the first argument to fs.mkdir).

In Node.js, the default behavior is to allow other code to run while a specific operation that could take time is executing. This is often referred to as *nonblocking*, meaning that the initiation of an action that may take time does not block the rest of the code from continuing to execute while waiting for that action to complete. When that action is complete, there is typically a function (in such a case, it can be referred to as a *callback* function) that executes upon completion.

Alternately, the completion of the action may spawn an *event* that you can respond to by assigning a function to be run when that event is triggered (you'll see examples of these as you progress). In the previous example, the fs.mkdir method takes in a callback function as the second argument, and this function is executed when the folder creation is complete.

It should be noted that Node.js does offer synchronous versions of most functions, though it is not typically the best practice to use them. For example, you can alter your test-node-async.js file to use the synchronous version of fs.mkdir, which is fs.mkdirSync:

```
const fs = require("fs"); fs.mkdirSync("test-folder");  
console.log("This displays after the folder is created!");
```

Run the script again (remember to delete test-folder or use a different folder name in the script), and the code following the fs.mkdirSync call will not run until the creation of the folder is complete. Thus, the message will be logged to the console after the operation is complete.

Using the synchronous version of a Node module method makes the code run in a synchronous fashion, but again this is usually not the best practice since asynchronous code does not block other code from being able to run. This is almost always good for users, as they can continue to interact with a program while the asynchronous function is waiting for a completed action. Synchronous code will force users to wait along with it, and that usually is

not seen in a very positive light by users! Keep in mind, though, that with asynchronous code, you can't assume when an action is completed and must put any dependencies inside the callback method.

Non-Native Modules

To include non-native modules in your Node.js script, you have a couple options: you can include a prewritten module from the npm (Node Package Manager) library, or you can write your own modules and include them. If npm has the package you need, you can install that package in your project and use it. If it doesn't, or if you simply want to write your own module code, you can do so and include that module in your script.

Node Package Manager

If you want to look for and use a module that is already available as a package, you can look in the npm library at www.npmjs.com. In the search bar, you can search for packages by name and view more details about them. For an example here, you will simply install the Express package, which allows you to create a Web server with Node.js quickly.

By default, Node.js comes with a copy of npm, though this copy will likely not be the latest npm release. You may wish to update npm before you begin using it. To do so, go to a console window and type the following:

```
npm install npm@latest -g
```

This should update npm to the latest version for you.

NOTE

Some versions of Windows may not completely update npm with this command, as the console window of choice may not yet find the new version. If your system does not, you can use the npm-windows-upgrade tool at www.npmjs.com/package/npm-windowsupgrade.

Once you are done, you can use npm to install a package into your project directory. For an example, open a command window in your project folder and run the following command:

```
npm install express
```

This will install the Express package within your project folder. If you don't have a node_modules folder in your project folder already, it will be created and the Express package will be within that folder. Any modules in the node_modules folder can be included without specifying a path (as you will see next).

NOTE

If npm install yields a permissions error, you may need to run your command window as an Administrator (Windows) or use sudo (Linux) to get the proper permissions.

Once the module is installed, making use of it is the same process as using a native module. Require the module and then use it in your script. For example, save the following code as `include-express.js` and then run it from your console window by typing **node include-express.js**, as shown here:

```
const express = require("express");
let http = express(); console.log("Express has been included and used!");
```

You should see the message logged to the console. Later in this chapter, you will learn more about how to use Express and create a Web server for an application.

Custom Modules

Custom modules are similar to the external script files you used previously for browser-based JavaScript, though with more options. To create a custom module, you simply write your code into a separate JavaScript file, make sure it exports the needed items, and then require it in your main script as you would other types of module.

For example, save the following code as `calculate-sum.js` in your project folder:

```
let calculateSum = function (value1, value2) {
  return value1 + value2; }
```

Notice that this file simply includes a function that can be called to calculate the sum of two values. This function can be improved to check that both values have been sent, are numeric, and provide error messages, but for this simple example, it will suffice.

Next, save the following code as `get-monthly-income.js`:

```
const sum = require("./calculate-sum"); and sent the paycheck values let paycheck1 = 1500;
defined as arguments. let paycheck2 = 1700; _____ to be paycheck values. let monthlyIncome =
sum.calculateSum(paycheck1, paycheck2); console.log(`My monthly income is ${monthlyIncome}`);
```

The custom module is required. The calculateSum function of the custom module is called

Two values are defined as arguments. _____ to be paycheck values.

The value returned from the function is logged to the console.

Notice that everything works in much the same way as using a native

module once the custom module has been required. However, there is one thing you will need to remember when writing your own modules: you need to export the values or functions you want to be able to use in your main script. The preceding code will fail because it will not be able to find the `sum.calculateSum` function.

To fix this, go back to your `calculate-sum.js` file and update it to use the following code:

```
let calculateSum = function (value1, value2) {  
  return value1 + value2;  
}
```

calculateSum: calculateSum	

```
module.exports = {  
  calculateSum: calculateSum  
};
```

Notice that a

module.exports value
is added and that the name of the
function to be used is exported.

Go to your console window and run the `get-monthly-income.js` script. It should now display the message “My monthly income is 3200” in the console. Notice that the `module.exports` value is added. It is assigned an object with a key and value. The first `calculateSum` is the key, while the second is the value of the key. What this means here is that the main script can use the key `calculateSum` to run the `calculateSum` function inside the module. If you want to use a different name to call the function outside of the module, you could change the value of the key. For example, look at the following code:

```
let calculateSum = function (value1, value2) {  
  return value1 + value2;  
}
```

```
module.exports = {  getSum: calculateSum
```

← Notice that the key has changed to `getSum`. };

This time, the `calculateSum` is only accessible to the main script (or other scripts) using `getSum`. For your example script, using `sum.getSum` will now work but `sum.calculateSum` will not.

Note that you can also export the function more directly if you wish by replacing the `module.exports` portion of the code with this:

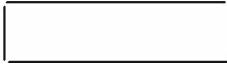
```
exports.calculateSum = function (value1, value2) {  
  return value1 + value2;  
}
```

Use what works for you in your situation or organization. With this, you can now use custom modules in your Node.js scripts!

Try This 11-1 *Use a Custom Module*

pr11_1.js
script.

This project gives you practice using a custom module in a Node



pr11_1_module.js You will create a script, require your custom module, and make use of the module in your main script.

Step by Step

1. Set up a JavaScript file with the following code and save it as pr11_1.js:

```
const leftover = require("./pr11_1_module");  
let paycheck = 2500; let bills = 2499;  
let bankBalance = leftover.getBalance (paycheck, bills); console.log(`My bank account has  
${bankBalance} left!`);
```

2. Create a second JavaScript file with the following code and save it as pr11_1_module.js:

```
let getBalance = function (paycheck, bills) {  
    return paycheck - bills;  
}  
module.exports = { getBalance:  
    getBalance  
};
```

3. In a command window, type **node pr11_1.js**.
4. The result should be a message indication you have only \$1 left in the bank:

```
My bank account has $1 left!
```
5. Try out some different numbers in the main script and rerun the script from the command line. The message should change accordingly.

Try This Summary

In this project, you created a Node script and a custom Node module. From the main script, you were able to require the custom module and to use its `getBalance` function to determine what was left in a bank account after the bills have been paid. The first iteration left you only \$1, but you were then able to change these values as you saw fit and reuse the module to calculate the balance based on the new values you provided!

Installing a Database

A database allows you to store data in a way that is structured so that you can easily access it later. While you can store data in text-based files, these eventually become too large to make getting specific information from them a quick-and-easy task. Also, if you are storing anything even somewhat sensitive, you probably don't want that sitting in a text file where it could easily be read by an outside source if compromised.

Using a database with Node.js typically just requires using a module that will allow you to write JavaScript code to query the database and get results. Before you do this, though, you will need to install a database. For the purposes of this book, you will simply need to install a database on your computer. If you were to use a database for a Web site or Web app, you would need to install the database on a server that can be accessed by your Web site/app (if you are on a hosted Web environment, almost every Web host has some type of database option available). To begin, you will simply want to install a database on your computer to use locally and get familiar with making queries to it from Node.js.

Database Options

There are numerous databases you can choose. From mySQL, MongoDB, PostgreSQL, Oracle, and many others, you have plenty of options. What you ultimately use will depend on your situation and may even vary from project to project.

For the purposes of this book, PostgreSQL will be used. You may use one of the other databases if you choose, though instructions won't be provided for the other databases. That said, if you choose one of the SQL databases, the queries will be very similar, if not the same, as the queries shown for PostgreSQL (though the module used to run the queries from Node.js might use much different syntax).

NOTE

If you use a NoSQL database like MongoDB, things will be quite different, but if you have some previous experience with it, you still should be able to work alongside the exercises using the different query structure and syntax.

Install PostgreSQL

To install PostgreSQL, go to www.enterprisedb.com/downloads/postgres-postgresqldownloads. Here, you will need to follow the download and installation instructions for your operating system. For an OS like Windows, an installation program will be downloaded that you can run and then follow the on-screen prompts. You can simply keep all the default options for the purposes it will be used for in this book. The main thing to remember is the password you will be prompted to input. Be sure to keep this, as you will need it to access the pgAdmin utility, which will allow you to create users, databases, tables, and so on.

If you find you need help with the installation process, here are some tutorials that may help you through the installation process in more detail:

- **Windows** www.guru99.com/download-install-postgresql.html
- **Mac** www.robinwieruch.de/postgres-sql-macos-setup/
- **Linux** www.poftut.com/how-to-install-postgresql-server-into-linux-debian-ubuntu-centos-mint-fedora-redhat/

Once PostgreSQL is installed and you have the main (superuser) password, you are ready to begin making use of the database server to create users, databases, tables, and more.

Create a Database Using pgAdmin

The installation process should have installed the pgAdmin tool. Open this tool, go to the panel on the left, and expand the Servers node. This will display the node “PostgreSQL X.X” (where X.X is replaced with the version of PostgreSQL you installed). Click to expand this node, and you will be prompted for the superuser password you created during installation. Type in the password and press enter. Figure 11-3 shows the password prompt in pgAdmin.

Once you have entered the password successfully, you will have additional nodes available in the left panel, as shown in.

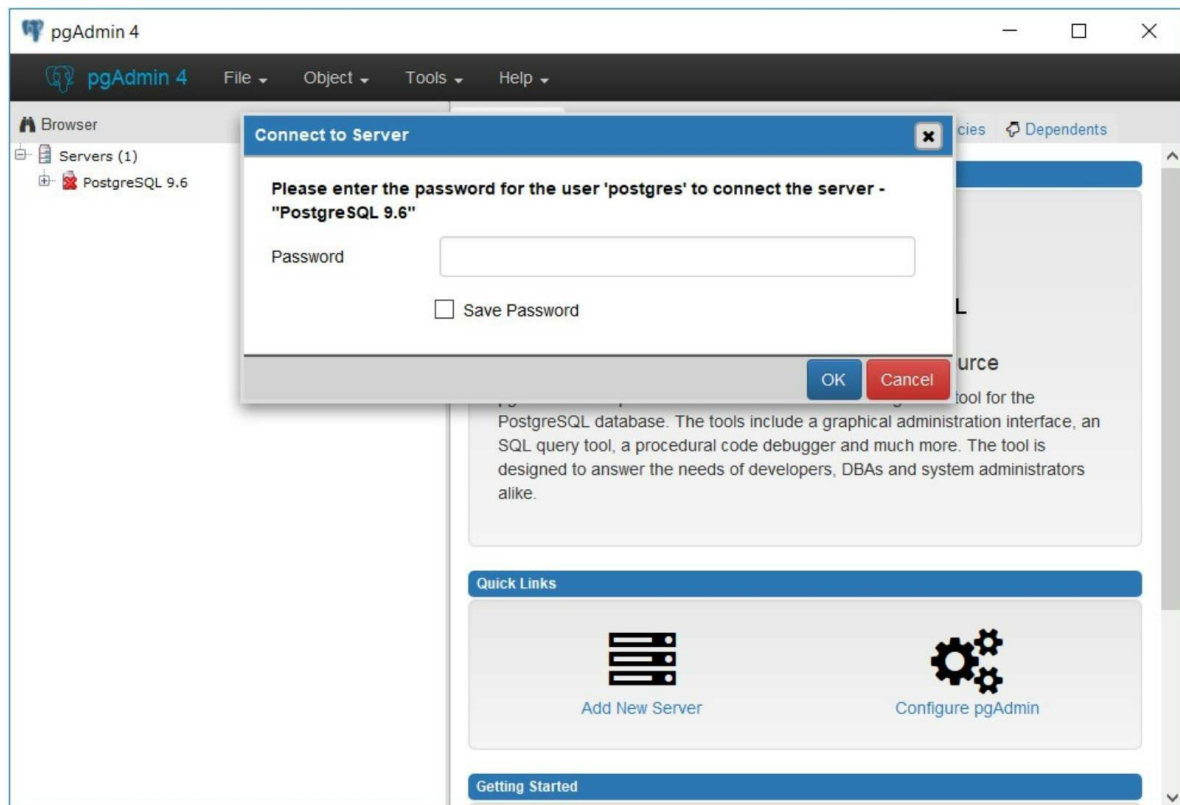
From here, right-click the Databases node and select Create and the Database option. This will bring up the prompt shown

Type in the name for the database (name this one mywebapp), click the Save button at the bottom of the window, and then close the window. You

just created a database that you can use!

Creating a Table

Once you have the database ready, you will want to put some sort of data in it. For the purposes of the simple Web app you will create, this can simply be done using an SQL query inside pgAdmin. First, select your database (mywebapp) in the left panel. Next, go to the top menu, select Tools, and then select Query Tool. A new query tab will display on the right side. Figure shows how to select the Query Tool in pgAdmin 4.

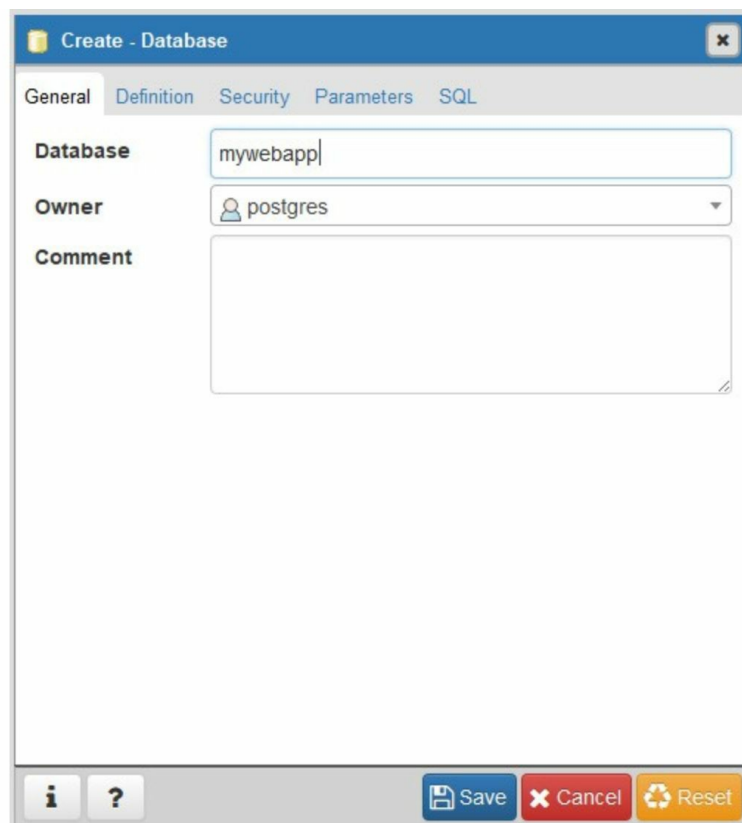


pgAdmin prompting for the password



The additional nodes available after you have logged in

The create table command allows you to create a table and to give it columns with particular data types. These data types are not the same as those used in JavaScript, but instead are PostgreSQL data types. You will notice the first and last names are of type varchar, which stands for character varying. The number in parentheses refers to the maximum length that any data in the column can be. For the first and last names, there will be a maximum of 20 characters. The remaining columns use the smallint and text data types (smallint includes integers from -32768 to 32767, and text is like varchar but does not have a length limit).



The image shows a 'Create - Database' dialog box with the following fields and controls:

- Database:** Text input field containing 'mywebapp'.
- Owner:** Dropdown menu showing 'postgres'.
- Comment:** Large text area for additional notes.
- Buttons:** 'Save' (blue), 'Cancel' (red), and 'Reset' (yellow) buttons at the bottom right.

Prompt for the creation of a new database

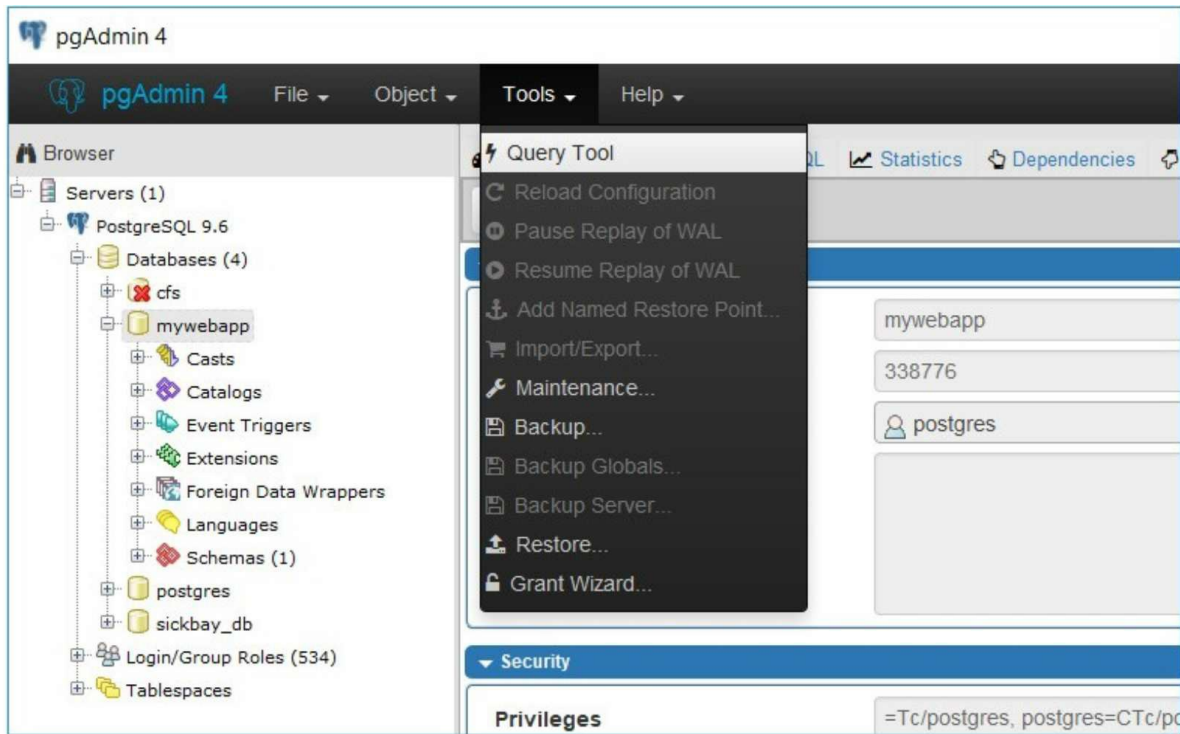


Figure Selection of the

The insert command allows you to insert data into the table. In this case, you inserted multiple values by listing the column names (first set of parentheses) and the values to put in the respective columns (second set of parentheses). Notice that character data is inside single quotation marks, while numeric data does not need the quotation marks.

Next, you will need to type in a query that creates a new table. The query you will use will both create a new table and add one row of data to the table. In the query pane, type the query shown in Figure 11-7 and then press the `f5` key (or click the lightning bolt button near the top).

This will create the table with some initial data for you.

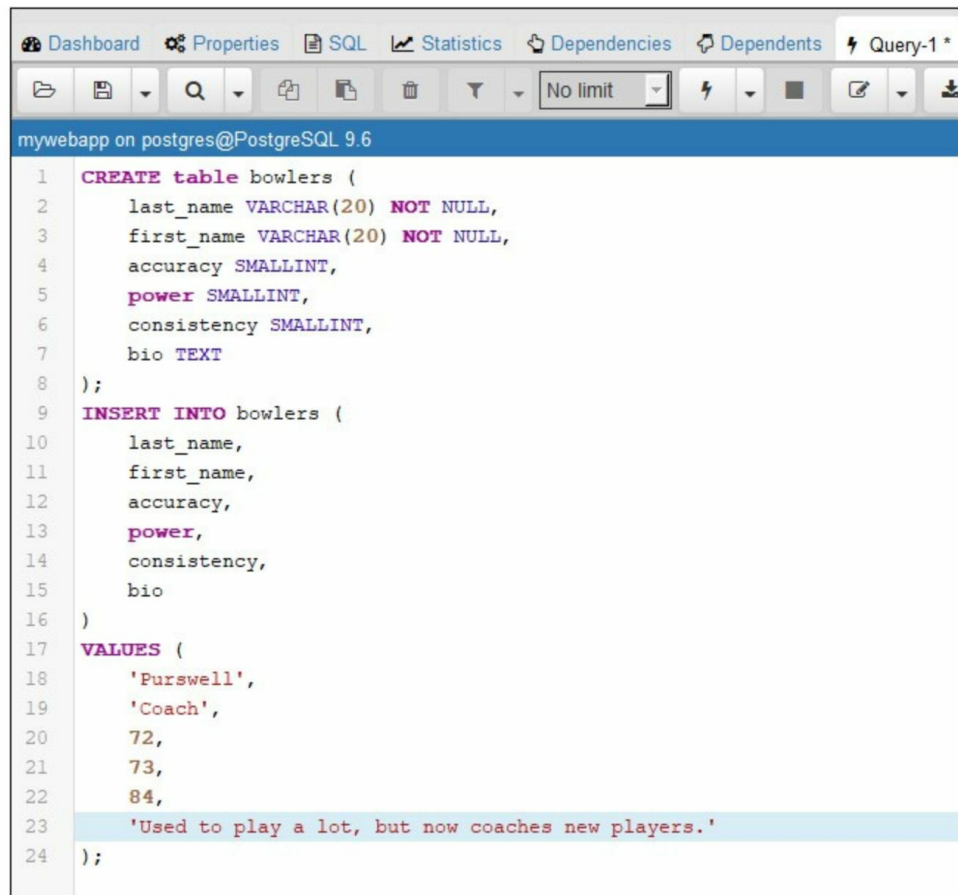
To make sure the table and data exist, you can use the Query Tool again to open another query tab and then enter the following query:

```
SELECT * FROM bowlers;
```

Figure 11-8 shows the result of this query in the bottom pane in pgAdmin.

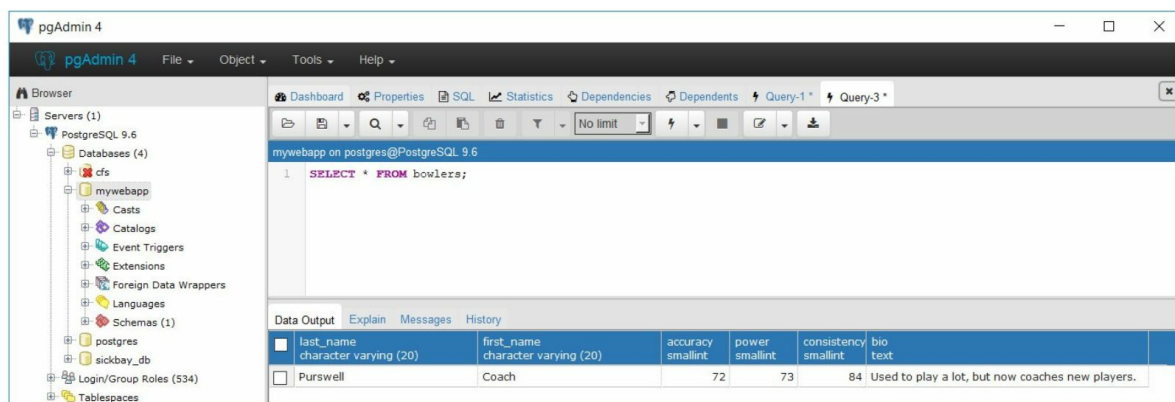
The select statement in SQL indicates that you want to make a selection of data from the database. In this case, you used the asterisk (*) to request that

all columns be selected from the bowlers table. Now you not only have a database, but you also have data that you can use!



```
1 CREATE table bowlers (  
2     last_name VARCHAR(20) NOT NULL,  
3     first_name VARCHAR(20) NOT NULL,  
4     accuracy SMALLINT,  
5     power SMALLINT,  
6     consistency SMALLINT,  
7     bio TEXT  
8 );  
9 INSERT INTO bowlers (  
10     last_name,  
11     first_name,  
12     accuracy,  
13     power,  
14     consistency,  
15     bio  
16 )  
17 VALUES (  
18     'Purswell',  
19     'Coach',  
20     72,  
21     73,  
22     84,  
23     'Used to play a lot, but now coaches new players.'  
24 );
```

The query to create and insert information into a table



The screenshot shows the pgAdmin 4 interface. On the left, the 'Servers' tree is expanded to show the 'postgres' database. The main pane displays the SQL query: `SELECT * FROM bowlers;`. Below the query, the 'Data Output' tab is active, showing the results of the query in a table format.

last_name	first_name	accuracy	power	consistency	bio
Purswell	Coach	72	73	84	Used to play a lot, but now coaches new players.

The query result when selecting all data from the table
Adding a Query User

While querying the database from the superuser account is convenient for administration and for creating databases, tables, users, and so on, it is not a good idea to have the superuser query your database from your Web app. Since the superuser can also delete anything, one wrong bit of code in your app or one malicious user being able to get a query to run on your database could turn into a huge loss of data or even a full security breach. For these reasons, it is best to create a separate user for your Web app and only grant that user the minimum access needed to be able to use the Web app.

For the Web app you will create, the new user only needs to be able to select data from your mywebapp database. You will first need to create the user and then grant the proper permissions using the Grant Wizard on your database.

To create a new user, go to the left panel in pgAdmin and right-click the Login/Group Roles node. Select Create | Login/Group Role. A prompt will display that requires a name for the user. In this case, type the name **mywebappuser**. This is shown in Figure 11-9. Next, go to the Definitions tab and enter a password for this new user. After that, click the Privileges tab, use the values shown in Figure 11-10, and then click the Save button at the bottom of the screen to save the new user.

The image shows a screenshot of the 'Create - Login/Group Role' dialog box in SQL Server Enterprise Manager. The dialog has a blue title bar with the text 'Create - Login/Group Role' and a close button. Below the title bar is a tabbed interface with the following tabs: 'General', 'Definition', 'Privileges', 'Membership', 'Parameters', 'Security', and 'SQL'. The 'General' tab is currently selected. In the 'General' tab, there are two main sections: 'Name' and 'Comments'. The 'Name' section has a text input field containing the text 'mywebappuser'. The 'Comments' section has a large, empty text area. At the bottom of the dialog, there is a row of buttons: an information icon ('i'), a help icon ('?'), a 'Save' button with a floppy disk icon, a 'Cancel' button with a red 'X' icon, and a 'Reset' button with a circular arrow icon.

Giving the new user a name

Create - Login/Group Role

General Definition **Privileges** Membership Parameters Security SQL

Can login? ☒ Yes ☐ No

Superuser ☐ Yes ☒ No

Create roles? ☐ Yes ☒ No

Create databases? ☐ Yes ☒ No

Update catalog? ☐ Yes ☒ No

Inherit rights from the parent roles? ☒ Yes ☐ No

Can initiate streaming replication and backups? ☐ Yes ☒ No

Save **Cancel** **Reset**

Setting the privileges for the new user

Once the user has been saved, you will give the user access to your mywebapp database.

In the left panel, select your mywebapp database, right-click the node, and select Grant Wizard. You will receive a prompt asking you to select a table. Select the bowlers table by checking the checkbox and then click the Next button at the bottom of the screen.

The next screen prompts you to add privileges to the table. Click the + button at the top right of the table and a new row will be added. In the Grantee column, select your new user, mywebappuser, from the drop-down list. Next, click inside the blank Privileges column. This will display a series of checkboxes. Check only the SELECT checkbox, which will only allow the user to perform select queries on the database (if you later decide users can insert data as well, you can come back and check the INSERT checkbox as well). Figure 11-11 shows this screen with SELECT checked. Finally, click the Finish button at the bottom of the screen. Your new user should now be able to select data from the

database!

Adding More Data

Before completing the database setup, you will add a few more rows to the bowlers table. For now, running the upcoming insert query will add four more entries, giving you a total of five rows. Once you create your Web app, this data can be queried in various ways that may be helpful to the user.

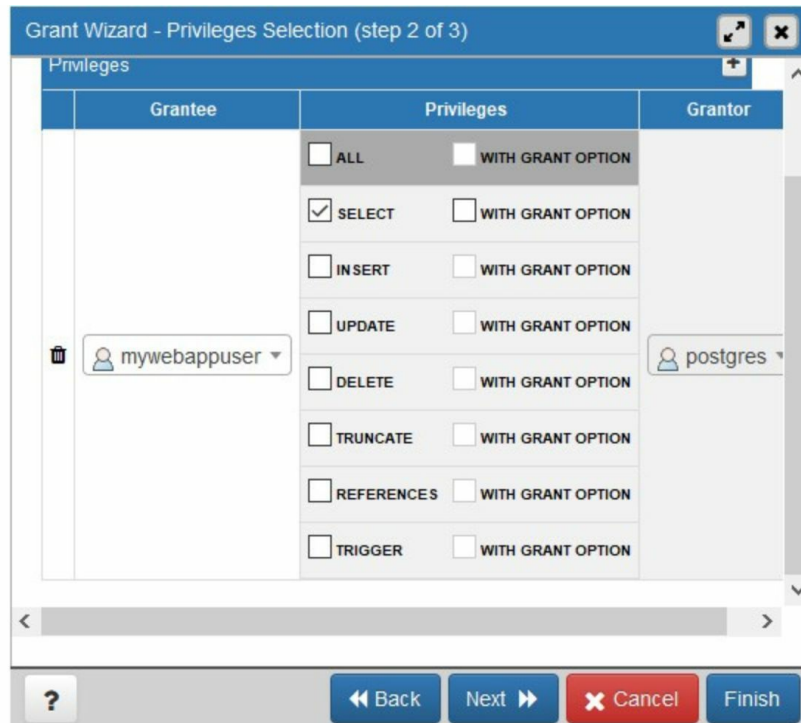


Figure Granting privileges

First, go back and select your mywebapp database in the left pane. Go to Tools and select Query Tool again. In the query pane, use the following query to add more data to your table. Note that you can change names or numbers to anything you like, but we will use the information in this particular query for the examples in this book.

```
INSERT INTO bowlers (last_name, first_name, accuracy, power, consistency, bio)
VALUES (
  'Gutter',
  'Doctor',
  55,
  47,
  62,
  'Likes to talk trash, but cannot knock down many pins.'
```

```

);
INSERT INTO bowlers (last_name, first_name, accuracy, power, consistency, bio)
VALUES (
  'Shoes',
  'Cranky',
  67,
  94,
  71,
  'Can roll the ball very fast. A bit inconsistent and
  might even throw the ball at you.'
);
INSERT INTO bowlers (last_name, first_name, accuracy, power, consistency, bio)
VALUES (
  'Roller',
  'Slow',
  82,
  39,
  93, 'Might take a long time to finish a turn, but is quite accurate.
  Does not knock down many pins though.'
);
INSERT INTO bowlers (last_name, first_name, accuracy, power, consistency, bio)
VALUES (
  'Winner',
  'Prize',
  97,
  95,
  98, 'Great all-around bowler. Play someone else if you want to win.' );

```

Note that the longer strings will need to be on one line in the query pane. Once the query is entered in the pane, press `f5` to run it and add the new rows of data to your bowlers table.

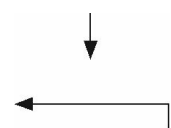
Using the pg Module

The last piece of knowledge you need about your new database (for now) is how to query your database in a Node.js program. To do this, you can use the `pg` module. First, open a command window in your project folder and then type the following to install the `pg` module:

```
npm install pg
```

Once the module is installed, open a new file and save it in your project folder as `pg_example.js`. Add the following code to the file, replacing 12345 with the password you created for mywebappuser, and then save the changes.

Creates the connection string. Be sure to replace 12345 with the password you created for mywebappuser.



```
const {Pool} = require('pg');  Requires the pg module. const connectionInfo =
```

```
`postgres://mywebappuser:12345@localhost:5432/mywebapp`; const pool = new Pool({connectionString:
connectionInfo});
```

```
pool.query(
  `SELECT * FROM bowlers`,
```

Creates a pool to query
the database using the

```

    }, connection information. function (err, result) { if (err) { console.error(err); The
query is run and process.exit(1); logged to the console.
    }
    console.log(result.rows); process.exit(0);
  }
};
```

```
C:\JP\JSABG-Ed-5-Writing\Chapter 11\768_11Project_Code>node pg_example.js
[ { last_name: 'Purswell',
  first_name: 'Coach',
  accuracy: 72,
  power: 73,
  consistency: 84,
  bio: 'Used to play a lot, but now coaches new players.' },
  { last_name: 'Gutter',
    first_name: 'Doctor',
    accuracy: 55,
    power: 47,
    consistency: 62,
    bio: 'Likes to talk trash, but cannot knock down many pins.' },
  { last_name: 'Shoes',
    first_name: 'Cranky',
    accuracy: 67,
    power: 94,
    consistency: 71,
    bio: 'Can roll the ball very fast. A bit inconsistent and might even throw the ball at you.' },
  { last_name: 'Roller',
    first_name: 'Slow',
    accuracy: 82,
    power: 39,
    consistency: 93,
    bio: 'Might take a long time to finish a turn, but is quite accurate. Does not knock down many pins though.' },
  { last_name: 'Winner',
    first_name: 'Prize',
    accuracy: 97,
    power: 95,
    consistency: 98,
    bio: 'Great all-around bowler. Play someone else if you want to win.' } ]
```

The result of running the pg_example.js script

You will notice that the pg module is included and sets up a “pool” using the connection information to query the database and return any results.

The connection information is simply a string that points to the PostgreSQL database server and provides the username, password, host, and database name.

With everything set up, you can use the `pool.query()` function to provide a query (the first argument), query parameters in an array (the second argument—in this case none, so an empty array), and the function to run when the query completes. In this function, you can have an error (`err` argument) or results (`results` argument). In this code, if there is an error, then the error will be displayed on the console and the Node process will exit. If the query was successful, the result is logged to the console. Figure 11-12 shows the result of running this script.

The result is an array of objects that contains the information for each bowler that was stored in the database!

Try This 11-2 *Test Some SQL Queries*

No files required, This project will allow you to try out some additional things you can do just pgAdmin with SQL queries to narrow the result set. So far, you simply got all of the rows in the table, but you can get fewer rows or even just a single row by altering the query just a bit.

Step by Step

1. In pgAdmin, select your mywebapp database in the left pane.
2. Go to Tools | Query Tool and enter the following into the query pane:
`SELECT * FROM bowlers WHERE accuracy > 80;`
3. Press `f5` to run the query. Notice that only bowlers whose accuracy is above 80 will be in the results list. The results in this case should be the information for Slow Roller and Prize Winner. Adding a WHERE clause can refine your results to only those that meet certain criteria.
4. Change the query to the following:
`SELECT * FROM bowlers ORDER BY power DESC;`
5. Press `f5` to run this query. Notice that all of the rows are returned, but that they are ordered from those with the highest power rating to those with the lowest power rating. ORDER BY can be used to sort the results according to a particular column's values; either is descending (DESC) or ascending (ASC) order.
6. Change the query to the

following:

```
SELECT first_name, last_name FROM bowlers WHERE consistency >= 80;
```

7. Press `f5` to run the query. Notice that now only the first and last names of the three bowlers with consistency greater than or equal to 80 are returned. Replacing `*` with one or more column names will limit the number of columns returned to those that are named.

Try This Summary

In this project, you tried some additional `SELECT` queries via the Query Tool in pgAdmin. You were able to refine your query to be able to get information in different ways or to limit the number of rows or columns that are returned. To find out more of what you can do with queries, see beginner-sql-tutorial.com/sql-select-statement.htm.

Creating a Web Server

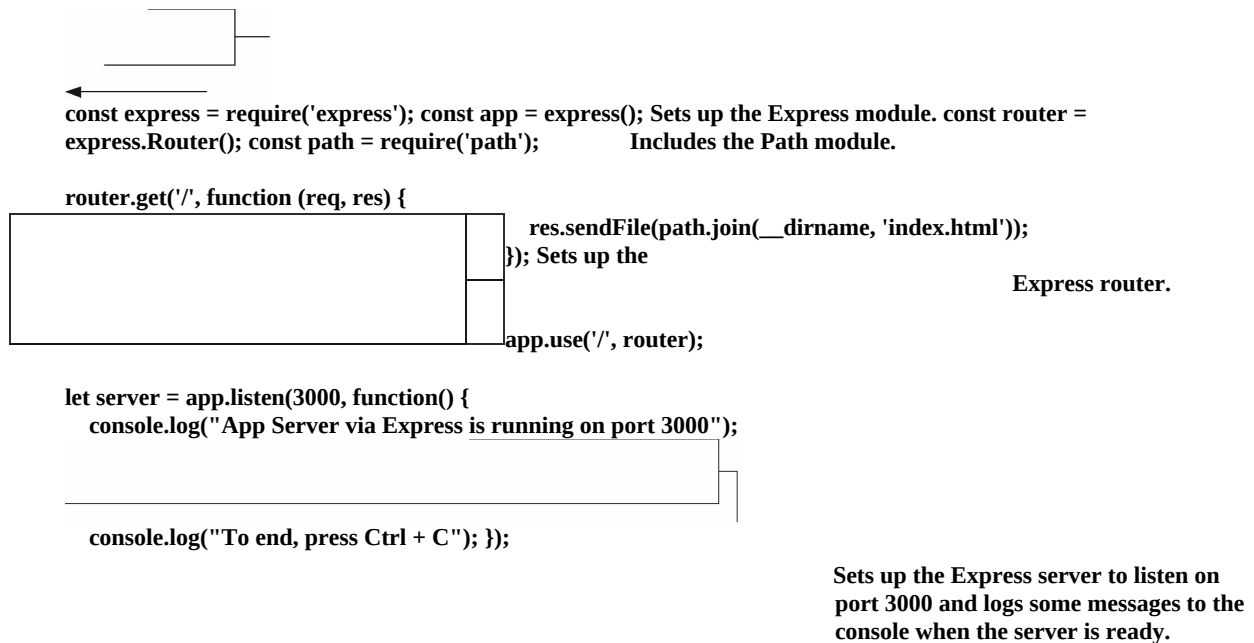
There are a number of options for creating a Web server with Node.js. In fact, you can write your own code to do so if you like. By and large, the standard is to simply use the Express module, which makes much of this quite a bit easier.

For this project, you want to create a very simple Web server that can listen on a particular port and render an HTML document when you visit a particular route—the most common being the base route `/` or the “main page.” For a simple local Web server, this means you want to be able to open <http://localhost> in your Web browser and see an HTML page displayed.

To begin, create an HTML file in your project folder with the following code and save it as [index.html](#), like so:

```
<html>
  <head>
    <title>My Web App</title>
  </head>
  <body>    The web server is working!
  </body>
</html>
```

Next, you want to create a JavaScript file that you can run as a Node script. Create another new file in your project folder with the following code and save it as `server.js`:



First, notice the first three lines of code:

```
const express = require('express');  
const app = express(); const router =  
express.Router();
```

Immediately, the Express module is included. An “app” is created by calling the `express()` function. This app will actively listen on a port once things are set up and use the router defined in the following line to direct the user to the right HTML page. As mentioned, the third line sets up an Express router, which allows you to define what routes will render what pages. A route can be anything from `/` for the main app page to `/view_statistics` route, which will open a stats page when visited. The Express router makes it easy to set up any routes you wish and respond to those routes in the proper way.

Next, the path module is included. This module allows you to easily navigate the file path on your machine. This will be used later in the script. The next portion of the code defines the routing:

```
router.get('/', function (req, res) {  
  res.sendFile(path.join(__dirname, 'index.html'));  
}); app.use('/', router);
```

As you can see, the `router.get()` function allows you to define what happens when a user tries to get information from that route. Most commonly, you respond by rendering a Web page of some kind—whether static or dynamic. In this case, a simple HTML page will do, so `res.sendFile()` is used to send the appropriate file. Here, it sends the `index.html` file you created earlier in

this section. This used `path.join` to join the folder that the `server.js` file is in (`__dirname` does this part) with the name of the file to render (in this case, the `index.html` file in your project folder). You can define additional routes the same way, but for now the router is completed by adding the **`app.use('/', router)`** line to the script.

Finally, the `app.listen()` function tells the Express app which port to listen to and whether to do anything once it is listening. To run the app (in this case, on port 3000), point your browser to <http://localhost:3000>.

With the files in place, go to your command window and type the following:

```
node server.js
```

You should get the logged information that the server is running on port 80 and how to stop the server when done. Once that is running, go to your Web browser and go to <http://localhost:3000>.

You should see the simple Web page that says that the Web server is working!

With this in place, you can now use your Web server to create a simple Web app that can even display the information you have stored in your database. You will do this as you add to this project while moving through the rest of this book and learn about sending requests from the front-end code to the server to get data!

8 MATH NUMBER AND THE DATA OBJECT

In this chapter, you will learn about the JavaScript Math, Number, and Date objects, in that order. For each object, a short introduction is provided along with a description of why the object can be useful to you. Following that is a look at the various properties and methods that you can use for that object.

Using the Math Object

The Math object can be useful when you need to perform various calculations in your scripts. To begin, take a look at what the Math object is.

What Is the Math Object?

The Math object is a predefined JavaScript object. Like the other predefined objects you have studied in this book, the Math object gives you properties and methods to use. The Math object is used for mathematical purposes to give you the values of certain mathematical constants or to perform certain calculations or operations.

How the Math Object Is Useful

As mentioned, the Math object is useful when you need to make mathematical calculations in your scripts. For instance, if you need the value of pi for a calculation, the Math object gives you a property to use so you can get that value.

Also, if you need to find the square root of a number, a method of the Math object enables you to do this. Another thing this object provides is a way to generate random numbers in JavaScript, which you will find useful in certain scripts.

Properties

The Math object gives you a number of properties that can help you if you need to perform certain mathematical calculations. Table 12-1 lists the properties of the Math object, with the values of each.

As you can see in Table 12-1, all of the properties simply hold numeric values that can be useful in mathematical calculations. Because these are irrational numbers, the values listed are nonterminating and are thus approximations. Note that these properties are in “all-caps.” This is because they are constant values, and the naming convention for such values is that they must be in all capital letters.

Property	Value
E	Value of Euler’s constant (E), which is about 2.71828...
LN10	Value of the natural logarithm of 10, which is about 2.302585...
LN2	Value of the natural logarithm of 2, which is about 0.693147...
LOG10E	Value of the base 10 logarithm of E, which is about 0.43429...
LOG2E	Value of the base 2 logarithm of E, which is about 1.442695...
PI	Value of pi, often used with circles, which is about 3.14159...
SQRT2	Value of the square root of 2, which is about 1.4142...
SQRT1_2	Value of the square root of one half, which is about 0.7071...

Properties of the Math Object

Using the Properties

The properties contain read-only values, which tend to be useful in particular types of calculations. For instance, if you want to find the area of a circle, you use the formula $\text{Area} = \pi * r^2$. Knowing that, you could write an application to determine the area of a circle based on the radius input by a viewer. You could use the following code, starting with the HTML code:

```
<form id="getarea">
<label for="radius">Enter a radius:</label><br>
<input type="text" id="radius">
<input type="submit" value="Get the Area!">
</form>
```

The form element is retrieved

Next, the JavaScript code:

by its id (getarea)

```
let area_form = document.getElementById("getarea");
area_form.addEventListener("submit", function(event) { let rad =
document.getElementById("radius").value,
the_area = 0;
if (rad.length < 1) {
```

The value of the text box (with

an id of radius) is retrieved using its value property

<pre>event.preventDefault(); alert("Please enter a radius!"); return;</pre>	<pre>Checks to see if data is } entered and alerts if not else { event.preventDefault(); rad = !isNaN(rad) ? 0 : +rad false; if (rad === false) {</pre>
	<pre> alert("Radius must be numeric!"); return;</pre>

Checks to see if the data entered is not numeric and alerts if not

```
    }
    the_area = Math.PI * (rad * rad);

    alert("The area is " + the_area + " square units. ");
}
}, false);
```

Performs the calculation
and displays the answer

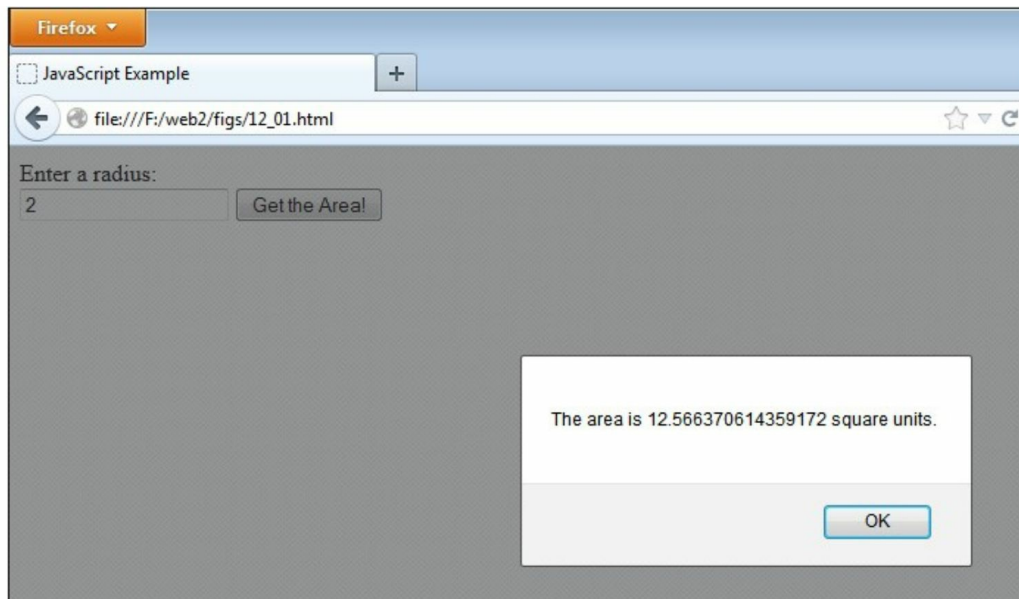
This code first grabs the form element by its id (getarea) and assigns it to a variable named get_area. When the form is submitted, it grabs the value input in the text field for the radius and assigns it to a variable named rad.

Next, the code performs two checks. It checks that the length property of rad is not less than 1, to ensure that the viewer entered data into the field. If it is less than 1, the code gives an alert asking the viewer to enter the information. Next, it checks to see if the value of rad is not numeric. This is done by using the isNaN() function, which returns true if a value is not numeric. Since you want to test whether it is numeric, adding the NOT operator in front makes sure the value does not come back as NaN and thus is numeric. If this is not the case, it is assigned a Boolean value of false. The if statement will then check the value of rad, and will send the viewer an alert saying that the radius must be numeric if it could not be coerced to a number.

Finally, if the data entered passes those tests, the calculation of the area of the circle is performed and assigned to a variable named the_area. Notice that for now you multiply the radius by itself to get the radius squared. When you get to the math methods in the next section, you will see that the method pow() may be used instead.

Once the calculation is complete and assigned to the variable, an alert is displayed to the viewer with the area in generic “square units.” The script could of course be altered to suit your needs or to offer options (centimeters, inches, or other units of measure).

Figure 12-1 shows the result of running this script in a browser and entering 2 into the text field.



The area of the circle is displayed for the viewer.

Methods

The methods of the Math object enable you to perform certain calculations that can be helpful in your scripts. Table 12-2 lists the basic methods of the Math object and briefly describes the purpose of each. Each method is discussed in more detail in the following sections. Also, some additional methods can be found at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math#Methods.

The Basic Methods

For the purpose of this book, “basic methods” are defined as the methods that take in a single number, do a simple calculation with it, and return a value. Grouping the methods in this way avoids the need to list each method with the same sort of example—it is not any sort of official organization of the methods, just a way to expedite this discussion.

Method	Purpose
abs()	Returns the absolute value of the number sent as an argument
acos()	Returns the arccosine of the number sent as an argument, in radians
asin()	Returns the arcsine of the number sent as an argument, in radians
atan()	Returns the arctangent of the number sent as an argument, in radians
atan2()	Returns the arctangent of the quotient of two numbers sent as arguments, in radians

ceil()	Returns the smallest integer greater than or equal to the number sent as an argument
cos()	Returns the cosine of the number sent as an argument, in radians
exp()	Returns the value of E to the power of the number sent to the method as an argument
floor()	Returns the largest integer less than or equal to the number sent as an argument
log()	Returns the natural logarithm of the number sent as an argument
max()	Returns the larger of the two numbers that are sent as arguments
min()	Returns the smaller of the two numbers that are sent as arguments
pow()	Returns the numeric value of the first argument raised to the power of the second argument
random()	Returns a random number between 0 and 1; does not require an argument
round()	Returns the value of the number sent as an argument rounded to the nearest integer
sin()	Returns the sine of the number sent as an argument, in radians
sqrt()	Returns the square root of the number sent as an argument
tan()	Returns the tangent of the number sent as an argument, in radians

Methods of the Math Object

The following basic methods are the ones that work in generally the same way:

abs()	exp()
acos()	log()
asin()	sin()
atan()	sqrt()
cos()	tan()

Each of these basic methods takes a numeric value and sends back another value. Since the general usage is the same, this discussion uses sqrt() as an example of how the rest could be used to get their various values. If you need to know what type of value is returned from a different method, refer to Table 12-2 to see what each method does.

The easiest way to use the sqrt() method is to input a positive number as the argument to the method, as shown in the following example:

```
window.alert(Math.sqrt(4));
```

This alerts the value of the positive square root of 4, which is 2.

Instead of calculating a static number, you could get the user to input a number, and then send an alert to the user with the square root of the number input by the user. You could do this using the following code, starting with the HTML code:

```
<form id="getroot">
<label for="sr_num">Enter a (positive) number or zero:</label><br>
```

```
<input type="text" id="sr_num">
<input type="submit" value="Get a Square Root"> </form>
```

The value of the text box (with

Next, the JavaScript code:

an id of sr_num) is retrieved

```
let root_form = document.getElementById("getroot");
root_form.addEventListener("submit", function(event) { let thenum =
document.getElementById("sr_num").value;
thenum = !isNaN(thenum) ? thenum : false;
event.preventDefault();
alert("Input is required and must be numeric!"); return;
}
else if (thenum < 0) {
alert("Hey! I said to enter a positive number! Try again.");
return;
}
else {
let theroot = Math.sqrt(thenum);
alert("The square root of " + thenum + " is " + theroot);
}, false);
```

if (!thenum || (thenum.length < 1)) {
Checks to see if the input is empty or is not
numeric

If the number entered is
negative, the viewer has to
try again

The square root is calculated
and alerted to the viewer

When the user submits the form, the value input into the text box is assigned to the thenum variable. Next, the function checks to see if the field was left empty or the data entered was not numeric. After that, it checks to see if the viewer entered a negative number. Any of these situations will alert the viewer to the error. Otherwise, the Math.sqrt() method is called using the value of thenum as an argument. This returns the number's square root, which is then assigned to the variable named theroot. Finally, an alert appears telling the user the square root information.

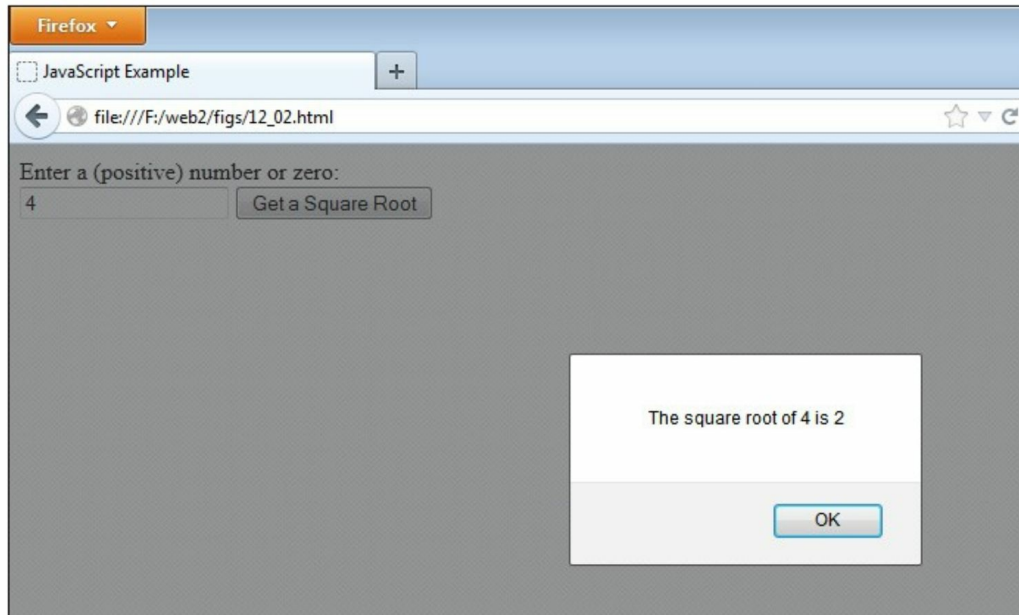
The result of this script when the viewer enters the number 4 is shown in Figure 12-2.

The other methods in this section work in much the same way; they just return different results such as absolute values, tangents, or logarithms.

The Two-Argument Methods

This section discusses the methods that take two arguments instead of just one. These methods include the following:

- atan2()
- max()
- min()
- po



The square root is displayed for the viewer.

The max() and min() methods are very similar, while the pow() method does something a bit different.

The max() and min() Methods The max() method takes two or more numbers and returns the largest number. The min() method also takes two or more arguments, but returns the smaller number. You could use these methods in a script that enables the viewer to enter two numbers and then alerts the user which number is larger.

The following example code uses both of these methods and gives the viewer the results in an alert. First, the HTML code:

```
<form id="getmax">
Enter two numbers to see which is bigger:<br><br>
<label for="num1">1st Number</label>
<input type="text" id="num1"><br>
<label for="num2">2nd Number</label>
<input type="text" id="num2"><br><br>
<input type="submit" value="Which Number is Bigger?"> </form>
```

Next, the JavaScript code:

```
let max_form = document.getElementById("getmax"); max_form.addEventListener("submit",
```



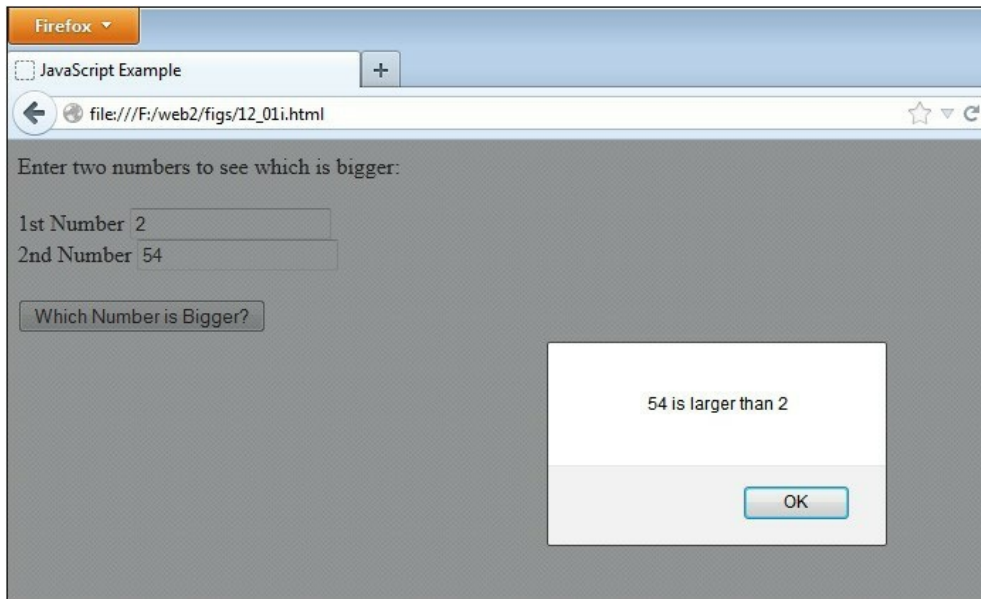
```

function(event) { let num1 = document.getElementById("num1").value,    num2 =
document.getElementById("num2").value,
    largenum = 0,    smallnum = 0;
    num1 = (+num1 === 0) ? 0 : +num1 || false; num2 = (+num2 === 0) ? 0 : +num2 ||
false;
    event.preventDefault();
    if ((num1 !== false) && (num2 !== false)) {
        largenum = Math.max(num1, num2); The results of the maximum and    smallnum =
        Math.min(num1, num2);minimum are assigned to variables    if (largenum === smallnum) {
            alert("Those two numbers are equal!"); A check in case the
            numbers are equal
        }
        else {
            alert(largenum + " is larger than " + smallnum); ← The alert will display
            if the numbers are
            not equal
        }
        else {
            alert("Please enter only numeric values."); ← The alert will display if
            either value is not numeric
        }
    }, false);

```

This form asks the viewer for two numbers and assigns them to variables. A check is performed to ensure that numeric values are entered. If so, it takes the maximum and minimum from both numbers and assigns the values to variables. Those variables are then used to check whether they are equal. If so, an alert comes up saying they are; otherwise, an alert pops up with the results.

The following illustration shows the results of this script in a browser if the viewer enters 2 as the first number and 54 as the second number.



The pow() Method The pow() method takes two arguments and calculates the value of the first argument to the power of the second argument. For instance, the following code would return the value of 2 to the third power:

```
Math.pow(2,3);
```

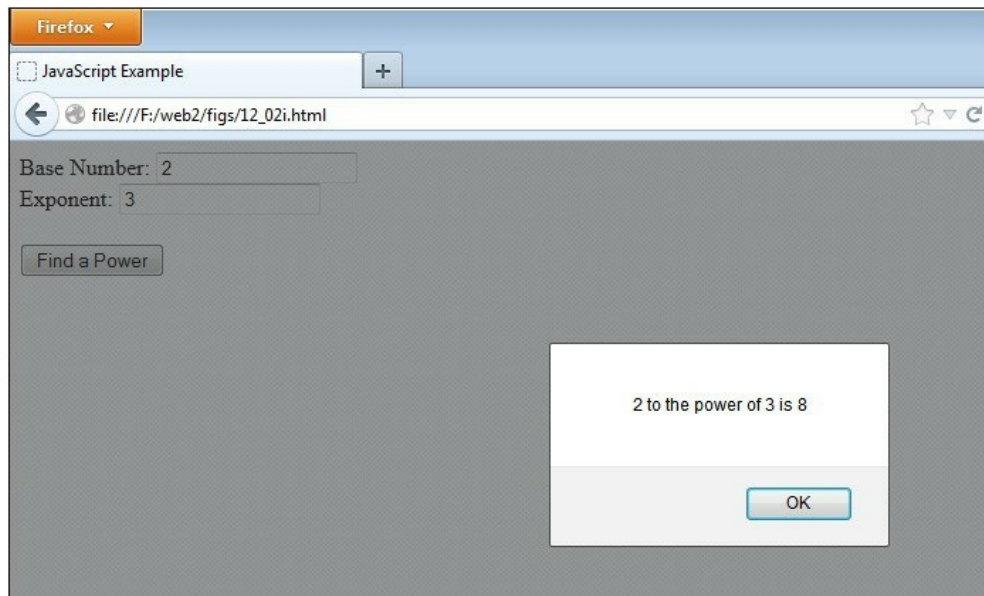
Other than its difference in calculations, you can use it in the same general way that you used the other two-argument methods by assigning the result to a variable and then using the variable in a script. As an example, you could use the following script, starting with the HTML code:

```
<form id="getpow">
<label for="base">Base Number:</label>
<input type="text" id="base"><br>
<label for="exp">Exponent:</label>
<input type="text" id="exp"><br><br>
<input type="submit" value="Find a Power"> </form>
```

Next, the JavaScript code:

```
let pow_form = document.getElementById("getpow"); pow_form.addEventListener("submit",
function(event) { let base = document.getElementById("base").value, exp =
document.getElementById("exp").value,
    result = 0;
    base = +base || false; exp = +exp || false;
event.preventDefault();
if ((base !== false) && (exp !== false)) {
    result = Math.pow(base, exp);
    alert(base + " to the power of " + exp + " is " + result);
} else {
    alert("Please enter only numeric values");
}
}, false);
```

Using this code, if the viewer enters 2 as the first number and 3 when asked for a power, the script will compute the result of 2 to the third power. The viewer then is given an alert showing the answer. Clicking the button in the HTML code is how the viewer starts the function. The result of this script when the numbers 2 and 3 are input is shown here:



Now that you know about the two-argument methods, take a look at some other methods that haven't been covered yet.

[The Other Methods](#)

These methods take a single argument, but what each does with that argument warrants a closer look. The individual methods include the following:

- `ceil()`
- `floor()`
- `round()`

The `ceil()` Method The `ceil()` method stands for *ceiling* and returns the smallest integer that is greater than or equal to the number sent as the argument. This method is used mainly when there are likely to be numbers after the decimal point in a number. It rounds the number up to the next highest integer, unless the number is an integer already. In that case, the same number is returned (because it can be equal). For instance, `Math.ceil(12.23)`; would return 13, but `Math.ceil(12)`; would return 12.

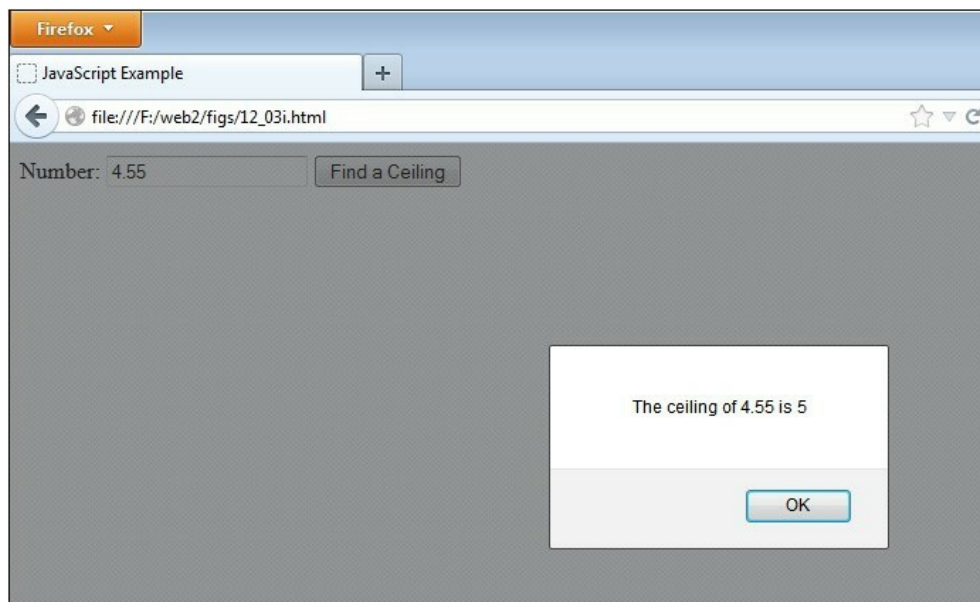
The following script shows an example of how the `ceil()` method can be used to return different values, starting with the HTML code:

```
<form id="getceiling">
<label for="num">Number:</label>
<input type="text" id="num">
<input type="submit" value="Find a Ceiling"> </form>
```

Next, the JavaScript code:

```
let ceil_form = document.getElementById("getceiling"); ceil_form.addEventListener("submit",
function(event) { let num = document.getElementById("num").value,
    ceiling = 0;
    num = isNaN(num) ? num : false;
    event.preventDefault(); if (num !== false) {    ceiling =
Math.ceil(num);
    alert("The ceiling of " + num + " is " + ceiling);
    } else {
    alert("Please enter a numeric value.");
    }
}, false);
```

This script displays an alert that states the ceiling of the number entered by the viewer. The following illustration shows the result of this in the browser when the viewer enters 4.55 at the prompt.



The floor() Method The `floor()` method is like the `ceil()` method, but it goes the opposite way. The `floor()` method returns the largest integer less than or equal to the argument sent to the method. This rounds down to the next lowest integer, unless the argument is an integer already. In that case, it returns the same integer since it is already equal to an integer. Basically, this method just removes the decimal part of a number and leaves the integer as

the result.

For instance, `Math.floor(12.23);` will return 12 and `Math.floor(12);` will also return 12. You can use the `floor()` method in the same way the `ceil()` method was used in the preceding section—by assigning the result to a variable.

The `round()` Method The `round()` method works like the previous two methods, but instead rounds the number entered as the argument to the nearest integer whether it is greater or less than the number. Any number having the decimal portion's value at .5 or greater will be rounded up, while any decimal portion with a value less than .5 is rounded down.

The .5 cutoff is strict, so `Math.round(12.49999999);` would return 12 even though your tendency may be to round it up.

The `random()` Method

The `random()` method is very useful for creating scripts that require random integers. It returns a random number between 0 and 1. This means that you get a number with a decimal that can be quite long and not useful on its own. For instance, it might return something like 0.36511165498095293.

To get a random integer that you can use, you need to do some things to get the type of value that you want to use.

Random Integers To get a random integer, the first thing you will want to do is to make the result have a greater range of values so that you are not stuck between 0 and 1. To get a greater range of values, you can multiply the result of the `random()` method by an integer to create a larger range. Like an array, the range would begin counting from 0; so, to get a range of five possible integers, you would multiply the result by 5. The following code shows how you can do this:

```
let rand_num = Math.random() * 5;
```

This gets the result between 0 and 4, but does not give you an integer yet. The number could still come out as a long decimal number.

To get an integer between 0 and 4, you need to find a way to make all of these decimal numbers convert to integers. Recall that earlier you ran through three methods, `floor()`, `ceil()`, and `round()`, that converted numbers to integers in various ways. The `floor()` method is the one you will choose here because it simply removes the decimal places after the integer and gives you the

integer portion of the number.

To use the `floor()` method, you could write the following code:

```
let rand_num = Math.random()*5; let rand_int =  
Math.floor(rand_num);
```

The `floor()` method takes in the value of the `rand_num` variable as an argument and then gives you an integer from it.

If you want to save a line of code, you could get a little fancy. You could just insert the `random()` method and calculation as the argument to the `floor()` method. You can do this because the result of the calculation, `Math.random*5`, is a number, and the `floor()` method can take a number as an argument. The following code shows how you can code this on a single line:

```
let rand_int = Math.floor(Math.random()*5);
```

Now the variable `rand_int` will have the value of a random integer between 0 and 4. As you might have noticed, this sort of number range could be quite useful with arrays. This is how you can begin to code some fun scripts with random numbers.

Note that if you want the random number to start at 1 instead of 0, you can add a `+ 1` to the end of the statement (this is helpful in user-facing code where that might be the expectation). Here is an example of this:

```
let rand_int = Math.floor(Math.random()*5) + 1;
```

Random Numbers for Scripts Now you can have a little fun with the `Math` object by using the `random()` method. By setting up some arrays, you can create a script that provides random quotes or shows a random image each time the page is loaded.

Random Quotes for Fun If you have thought about adding a quote to your page but don't want to deal with changing the quote all the time to have something different, a random-quotes script could be just the thing for you.

To make such a script, you first need some quotes to use. Suppose you want to set up ten different quotes that will be displayed in random order each time the page is loaded. Since you have a number of values that are similar (and so that you can use them with the random integer later), you should use an array so that you can store all of these values and retrieve them easily.

So, you need to set up an array with ten elements similar to the following

example, in which each element is a random (and perhaps peculiar) quote that I have thrown into the mix for you:

```
let quotes = ["Look in the mirror. Are you looking at me?",  
    "It is time for a rhyme, I guess.",  
    "Where is my JavaScript book?",  
    "If I had a buck for every dollar I spent--Oops, never mind.",  
    "I suppose you were expecting a real quote here.",  
    "Quotes are great, but don't quote me on that.",  
    "What should I write here?",  
    "Wut hapns iff eye miss spel ohn purpas?",  
    "Mark my words, I will mark my words.",  
    "This spot reserved for a better quote."    ];
```

Now that you have this odd list of quotes in an array, you can use them by generating a random integer.

You need a random integer between 0 and 9 (ten numbers), so you can use the following code to assign a random integer between 0 and 9 to a variable:

```
let rand_int = Math.floor(Math.random() * 10);
```

Now the value of the variable `rand_int` will be a random integer between 0 and 9. You can use it to access the element of the array whose index number matches the random integer in the `rand_int` variable. You just need to access the array element using the variable as the index number, as in the following example:

```
quotes[rand_int]
```

You can write this value in the body of the page using the `innerHTML` property of the `div` element, as in the following example code. The HTML document is saved as `random_quotes.html` and the JavaScript file is saved as `random_quotes.js`. The HTML code:

```
<h1>My Random Quote for You:</h1>  
<div id="my_quote"> Look in the mirror. Are you looking at me?  
</div>  
<script type="text/javascript" src="random_quotes.js"></script>
```

The JavaScript code:

```
let quotes = ["Look in the mirror. Are you looking at me?",  
    "It is time for a rhyme, I guess.",  
    "Where is my JavaScript book?",  
    "If I had a buck for every dollar I spent--Oops, never mind.",  
    "I suppose you were expecting a real quote here.",  
    "Quotes are great, but don't quote me on that.",  
    "What should I write here?",  
    "Wut hapns iff eye miss spel ohn purpas?",  
    "Mark my words, I will mark my words.",  
    "This spot reserved for a better quote."  
    ];  
let q_div = document.getElementById("my_quote"); let rand_int =
```

```
Math.floor(Math.random() * 10); q_div.innerHTML = quotes[rand_int];
```

The code writes one of the random quotes on the page based on the random integer value in the `rand_int` variable. A default quote (the first one in the array `data`) is provided for those without JavaScript, which is then overwritten with the random quote if JavaScript is available. Reloading the page enables the random number to be reset and will probably (though not necessarily, because it is random) show a different quote.

Figure 12-3 shows one of the possible results of this script when run in a browser.

Figure 12-4 shows another one of the possible results of this script when run in a browser.

You can keep getting different (or sometimes the same) results by refreshing the page.

Now that you can write random quotes into a page, how about displaying a random image? It is very similar to the last script; you just need to make some small adjustments.

Random Images for the Updated Look A random-image script can give your page the feel of being updated without requiring you to change an image all the time. Of course, the images all need to fit the content where you decide to place the randomly chosen image. A randomimage script could be useful, for example, for an art gallery to display its collection.



A possible result of using the random-quote script



Another possible result of using the random-quote script

The first thing you need is an array of image URL addresses (which can be local or absolute—local addresses are used here). The array used for this script is shown in the following example code:

```
let r_image= ["image0.gif", "image1.gif", "image2.gif", "image3.gif",  
             "image4.gif", "image5.gif", "image6.gif", "image7.gif",      "image8.gif", "image9.gif"];
```

This array sets up the addresses of images that can be displayed at random each time the page is loaded.

Next, you need a way to get a random integer between 0 and 9. This is the same as in the previous random-quote script:

```
Math.floor(Math.random() * 10);
```

The next step is to access the array in the body section of the document to show a random image from the array when the page is loaded in the browser.

The following code enables you to display the random image using the src property of an img element. The HTML file is saved as random_images.html and the JavaScript file is saved as random_images.js. First, the HTML code:

```
<h1>My Random Image for You:</h1>  
<div>   
</div>  
<script type="text/javascript" src="random_images.js"></script>
```

Next, the JavaScript code:

```
let r_image= ["image0.gif", "image1.gif", "image2.gif", "image3.gif",  
             "image4.gif", "image5.gif", "image6.gif", "image7.gif",  
             "image8.gif", "image9.gif"],    im =  
document.getElementById("my_image"),    rand_int =  
Math.floor(Math.random() * 10);  
im.src = r_image[rand_int];
```

As you can see, this is quite similar to the random-quote script. Since you are

changing an image, you can assign the result to the src property available on image elements to dynamically change the image. If you need to change text as well, you could change this to use innerHTML as you did with the random quotes.

NOTE

For this example, the images are saved as image0.gif, image1.gif, and so on; however, you could save your image files under any name you like.

This discussion of the random() method could go on for some time because there are numerous things that you could randomize. However, you now know the basics of how this feature works, so it's time to move on.

Q: numbers that I don't even use!	
A: canvas element), and more.	
Q: the properties of the Math object?	
A:	
Q:	
A:	
Q:	
A:	

Step by Step

1. Create an HTML page with script tags that point to an external JavaScript file named prjs12_1.js. Add a heading that says “Random

Link” and add a div element with an id of random_link. Inside the div element, insert a default link for those without JavaScript. Save the HTML file as pr12_1.html.

2. When complete, the body of the HTML file should look like this:

```
<body>
<h1>Random Link</h1>
<div id="random_link">
<a href="http://www.scripttheweb.com">Random Site!</a>
</div> <script type="text/javascript" src="prjs12_1.js"></script> </body>
```

3. Create an external JavaScript file and save it as prjs12_1.js. Use it for steps 4–7.
4. Create a set of five Web addresses in an array. Use the following addresses or some of your own choosing:

```
http://www.scripttheweb.com http://www.webxpertz.net
http://www.duckduckgo.com http://www.yahoo.com
http://www.google.com
```

5. Use the random() method to create a random integer you can use to access the array with the integer as an index number.
6. Display a random link on the page in the format shown next by changing the innerHTML property of the div element with the id of random_link:

```
<a href="random address here">Random Site!</a>
```

7. When complete, the JavaScript file should look like this:

```
let urls = ["http://developer.mozilla.org",
            "http://github.com",
            "http://www.duckduckgo.com",
            "http://www.yahoo.com",
            "http://www.google.com"
            ],
    rdiv = document.getElementById("random_link"),    rand = Math.floor(Math.random() *
5);
rdiv.innerHTML = '<a href="' + urls[rand] + '">Random Site!</a>';
```

8. Save the JavaScript file and open the HTML file in your Web browser. Try reloading a few times to see how the random addresses show up for the link.

Try This Summary

In this project, you were able to use your knowledge of the Math object. Using the random() method, you created a Web page that displays a link that goes to a random Web address.

Understanding the Number Object

The Number object is another predefined JavaScript object that offers several useful properties and methods for use with numbers. Its most common use is to access some of its helpful properties that represent certain values that can aid you when creating scripts. Number is slightly different from Math in that you can create a number to use as a number datatype with this object.

Properties

Table 12-3 lists the properties of the Number object and briefly describes the purpose of each. Some of the properties are described in more detail in this section.

Property	Purpose
EPSILON	The difference between 1 and the smallest floating point number greater than 1. Has an approximate value of 2^{-52} .
MAX_VALUE	Holds a constant number value, representing the largest value before JavaScript interprets a number as infinity.
MIN_VALUE	Holds a constant number value, representing the smallest value before JavaScript interprets a number as negative infinity.
MAX_SAFE_INTEGER	The maximum safe integer in JavaScript, which is $2^{53}-1$.
MIN_SAFE_INTEGER	The minimum safe integer in JavaScript which is $-(2^{53}-1)$.
NaN	Represents the value of “Not a Number.”
NEGATIVE_INFINITY	Represents the value of negative infinity.
POSITIVE_INFINITY	Represents the value of infinity.

Properties of the Number Object

The MAX_VALUE Property

The MAX_VALUE property is a constant number value, approximately 1.79E+308. The reason this property is helpful is that any number greater than its value is represented as infinity in JavaScript. Thus, using it in a comparison could provide a way to avoid calculations that are too large to display a numerical value. The following code is an example of how this works:

```
let big_num = num1 * num2 * num3; if (big_num >
Number.MAX_VALUE) {
  alert("The number is too large, try smaller numbers.");
} else {
  alert(big_num); }
```

Assuming num1, num2, and num3 were entered by the viewer, the alert, instead of displaying the word “infinity” as the answer, informs the viewer to

try entering smaller numbers for the calculation if the value of big_num is greater than the value of the MAX_VALUE property.

The MIN_VALUE Property

The MIN_VALUE property is a constant number value, approximately $5e-324$. The reason this property is helpful is that any number less than its value is represented as negative infinity in JavaScript. Thus, using it in a comparison could provide a way to avoid calculations that are too small to display a numerical value. The following code is an example of how this works:

```
let small_num = num1 - num2 - num3; if (small_num <
Number.MIN_VALUE) {
  alert("The number is too small, try larger numbers.");
} else {
  alert(small_num); }
```

Assuming num1, num2, and num3 were entered by the viewer, the alert tells the viewer to try entering larger numbers for the calculation if the result of the calculation is less than the value of the MIN_VALUE property.

The NaN Property

The NaN property is a value that represents “Not a Number.” It is displayed by the browser as a string value of NaN and is not equal to any number or another instance of NaN.

The NEGATIVE_INFINITY Property

The NEGATIVE_INFINITY property is a constant value that represents negative infinity. It can be used in a similar fashion to the way MIN_NUMBER and MAX_NUMBER are used.

The POSITIVE_INFINITY Property

The POSITIVE_INFINITY property is a constant value that represents positive infinity. It can be used in a similar fashion to the way MIN_NUMBER and MAX_NUMBER are used.

Method	Purpose
toExponential()	Returns a string value that represents the number in exponential notation
toFixed()	Returns a string value that represents the number rounded to the specified number of digits after the decimal
toPrecision()	Returns a string value that represents the number rounded to the specified number of significant digits
toSource()	Returns a string value that represents the source code of the object
toLocaleString()	Returns a language-sensitive version of the number

toString()	Returns a string value for a Number object
valueOf()	Used by JavaScript internally most often

Methods of the Number Object

NOTE

Infinity behaves differently in mathematical equations than other numbers. For more information on this, see tutorial.math.lamar.edu/Classes/CalcI/TypesOfInfinity.aspx.

Methods

lists some of the methods of the Number object and briefly describes the purpose of each. The following sections discuss some of these methods in more detail.

The toExponential(), toFixed(), toPrecision(), and toString() Methods

These methods return a string value representing what the Number object would look like formatted in a particular way. Note that these methods cannot be used with a number itself (a numeric value), as in the following code:

```
alert(10.toExponential());
```

This will cause a JavaScript error because it expects a Number object. To avoid this, use the methods by assigning numeric values to variables (which will make them Number objects), as in the following code:

```
let the_num = 10; alert(the_num.toExponential());
```

NOTE

You could also use the constructor syntax of `let the_num = new Number(10);` or simply `new Number(10).toExponential()` if you just need the value.

The toExponential() Method

The toExponential() method returns a string representing a Number object in the form of exponential notation. Thus, the following code would write the result of 1.0e+1 (or a similar notation, depending on your browser) on the screen:

```
let the_num = 10; alert(the_num.toExponential());
```

The toFixed() Method

The toFixed() method returns a string representing a Number object rounded to the specified number of places after the decimal. For instance, if you need

to format the results of calculations to appear as monetary values, you could use this method to get the result of your calculation rounded at the second digit after the decimal and displayed in your currency format. For example, this code uses dollars and cents:

```
let kidsmoney = 2000; let mykids = 7;  
let one_share = kidsmoney/mykids; alert("One share of the money is $" + one_share.toFixed(2));
```

The result of the calculation for one share is 285.7142857142857, but since it is displayed using the `toFixed()` method on it with 2 as the argument, the sentence displays as follows:

One share of the money is \$285.71

The `toPrecision()` Method

The `toPrecision()` method returns a string representing a Number object rounded to the specified number of significant digits. This is for all digits before and after the decimal. Thus, if you wanted a number like 45.57689349 rounded to five significant digits, you could use the following code:

```
let the_num = 45.57689349; alert(the_num.toPrecision(5));
```

The browser will alert the string 45.577, which is the number rounded to five significant digits (two before and three after the decimal place in this case).

The `toString()` Method

The `toString()` method returns the string value of a Number object (or a numerical variable value). This can be useful if you want to convert a numerical value to its corresponding string value (for example, change 10 to "10").

The `toSource()` Method

The `toSource()` method returns a string value that represents the source code of the object.

With the predefined Number object, this method returns the value of the constructor property.

This method is most often called by JavaScript internally and is less likely to be used in code.

The `valueOf()` Method

This is another method that is mainly used by JavaScript internally. For now, you just need to know that it is a valid method of the Number object.

The parseInt() and parseFloat() Methods

These two methods are useful when you need to get a numeric value from a mixed string. They work in a similar fashion, with parseInt() only grabbing the integer portion of a number and parseFloat() grabbing the integer and any numbers after a decimal. Here is how they work:

- If the first character in the string is anything other than a number, a plus (+), or a minus (–), then both methods return NaN.
- For parseInt(), if the first character is a number (other than zero), plus, or minus, then each character is evaluated until a non-numeric character is reached (with the exception of hexadecimal values, where certain other characters such as x and A are allowed). The number obtained from this process will be returned. If the first character is zero and the number is not hexadecimal, then zero is returned.
- For parseFloat(), if the first character is a number, plus, or minus, it will evaluate each character and continue if it finds a dot (.) followed by additional numbers, until a second non-numeric character is reached. If the number begins with zero and the number is not hexadecimal, then any leading zeros are ignored until another number is reached. The number obtained from this process will be returned. Hexadecimal numbers return zero.

Some examples of the values returned for each method will show how the rules work:

```
parseInt("14"); // Returns 14 parseInt("cool567"); // Returns NaN parseInt("+34blah"); //  
Returns 34 parseInt("-23awesome"); // Returns -23 parseInt("020"); // Returns 0  
parseInt("0xA"); // Returns 10 – the hexadecimal value
```

```
parseFloat("14"); // Returns 14 parseFloat("blah22"); // Returns NaN parseFloat("123.456");  
// Returns 123.456  
parseFloat("123.45.6"); // Returns 123.45 – second dot ends the  
process  
parseFloat("023.44"); // Returns 23.44 parseFloat("0xAF"); // Returns 0
```

When using parseInt(), you are allowed to define a second argument, called the radix. This gives you the opportunity to specify the type of number to be returned. Here are some examples:

```
parseInt("11", 10); // Returns 11 – uses the decimal value parseInt("11", 2); // Returns 3 – uses the binary value  
parseInt("11", 16); // Returns 17 – uses the hexadecimal value parseInt("11", 8); // Returns 9 – uses the octal value
```


As you can see, the same value can have different results depending on the specified radix. The `parseFloat()` method does not have this option, since by design it will only parse decimal values.

TIP

When the radix argument is left off, JavaScript decides what type of value to return. In some cases, this can lead to unexpected results. To avoid this, it is best to specify the radix value, even when working with and expecting decimal values (specify the radix as 10).

Using the Date Object

The Date object is another predefined JavaScript object. It enables you to set certain time values and to get certain time values that you can use in your scripts. To use this object, you need to create an instance of the object to which you can refer.

To create an instance of the Date object, you use the `new` keyword (as you have with a number of other objects), as shown in the following example:

```
let instance_name = new Date();
```

You would replace *instance_name* with a name that you want to use for the instance of the

Date object. So, if you wanted an instance named `rightnow`, you could use the following code:

```
let rightnow = new Date();
```

Now you have an instance of the object named `rightnow`.

When using the constructor, the default date is the current date; however, you can specify a different date if needed, as in the following code:

```
let go_back = new Date("November 5, 1955");
```

Once you have an instance of the object, you can use the properties and methods of the Date object to perform various tasks (such as create JavaScript clocks). These properties and methods are described in the following sections.

Properties and Methods

The Date object doesn't give you properties (outside of constructor and prototype), but it does have quite a large number of methods you can use.

Table 12-5 lists various methods of the Date object and the purpose of each method. Further methods can be found at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date#Methods.

Now that you have the long list of methods, take a look at them in a little more detail, beginning with the methods used to get date values in an instance of the Date object.

Method	Purpose
<code>getDate()</code>	Returns the day of the month based on the viewer's local time
<code>getDay()</code>	Returns the number of days into the week based on the viewer's local time (0–6)
<code>getHours()</code>	Returns the number of hours into the day based on the viewer's local time (0–23)
<code>getMilliseconds()</code>	Returns the number of milliseconds into the second based on the viewer's local time (0–999)
<code>getMinutes()</code>	Returns the number of minutes into the hour based on the viewer's local time (0–59)
<code>getMonth()</code>	Returns the number of months into the year based on the viewer's local time (0–11)
<code>getSeconds()</code>	Returns the seconds into the minute based on the viewer's local time (0–59)
<code>getTime()</code>	Returns the number of milliseconds since 1/1/1970 for the Date object
<code>getTimezoneOffset()</code>	Returns the time-zone offset (from Greenwich Mean Time) in minutes based on the viewer's local time zone
<code>getFullYear()</code>	Returns the full year based on the viewer's local time (four digits)
<code>getUTCDate()</code>	Returns the day of the month in Coordinated Universal Time
<code>getUTCDay()</code>	Returns the number of days into the week in Coordinated Universal Time (0–6)
<code>getUTCFullYear()</code>	Returns the full year in Coordinated Universal Time (four digits)
<code>getUTCHours()</code>	Returns the number of hours into the day in Coordinated Universal Time (0–23)
<code>getUTCMilliseconds()</code>	Returns the number of milliseconds into the current second in Coordinated Universal Time (0–999)
<code>getUTCMinutes()</code>	Returns the number of minutes into the hours in Coordinated Universal Time (0–59)
<code>getUTCMonth()</code>	Returns the number of months into the current year in Coordinated Universal Time (0–11)
<code>getUTCSeconds()</code>	Returns the number of seconds into the current minute in Coordinated Universal Time (0–59)
<code>parse()</code>	Returns the number of milliseconds since 1/1/1970 of a date sent as an argument based on the viewer's local time
<code>setDate()</code>	Sets the day of the month for an instance of the Date object
<code>setHours()</code>	Sets the hours for an instance of the Date object
<code>setMilliseconds()</code>	Sets the milliseconds for an instance of the Date object
<code>setMinutes()</code>	Sets the minutes for an instance of the Date object
<code>setMonth()</code>	Sets the month for an instance of the Date object

setSeconds()	Sets the seconds for an instance of the Date object

Methods of the Date Object (*continued*)

Method	Purpose
setTime()	Sets the time (in milliseconds since January 1, 1970, at midnight) for an instance of the Date object
setFullYear()	Sets the full year for an instance of the Date object (four digits)
setUTCDate()	Sets the day of the month in Coordinated Universal Time
setUTCFullYear()	Sets the full year in Coordinated Universal Time (four digits)
setUTCHours()	Sets the number of hours into the day in Coordinated Universal Time (0–23)
setUTCMilliseconds()	Sets the number of milliseconds into the current second in Coordinated Universal Time (0–999)
setUTCMinutes()	Sets the number of minutes into the hours in Coordinated Universal Time (0–59)
setUTCMonth()	Sets the number of months into the current year in Coordinated Universal Time (0–11)
setUTCSeconds()	Sets the number of seconds into the current minute in Coordinated Universal Time (0–59)
toDateString()	Returns the date portion of the Date object as a string in American English
toLocaleString()	Returns a string that is the date in a format based on the locale
toLocaleDateString()	Returns the date portion of the Date object as a string based on the locale
toLocaleTimeString()	Returns the time portion of the Date object as a string based on the locale
toString()	Returns a string that is the date in American English
toTimeString()	Returns the time portion of the Date object as a string in American English

Methods of the Date Object

Methods That Get Values

Methods that get values enable you to get various time and date values that you can use in your scripts. The methods that enable you to get values for an instance of the Date object include the following:

getDate()	getFullYear()
getDay()	getUTCDate()
getHours()	getUTCDay()
getMilliseconds()	getUTCFullYear()
getMinutes()	getUTCHours()
getMonth()	getUTCMilliseconds()
getSeconds()	getUTCMinutes()
getTime()	getUTCMonth()
getTimezoneOffset()	getUTCSeconds()

To use these methods, you need an instance of the Date object. Once you

have that, you can call any of the methods by using the instance name. The following is the syntax for doing this:

```
instance_name.method();
```

You would replace *instance_name* with the name of your instance of the Date object, and you would replace *method* with the method function you wish to use.

So, if you wanted to use the getDate() method with an instance of the Date object named rightnow, you would use the following code:

```
let rightnow = new Date(); let theday = rightnow.getDate();
```

This assigns the value returned from the getDate() method to a variable named theday.

Because the values returned from the Date methods are often numeric, the methods need to be explained a bit further; thus, the following sections take a look at these methods more closely.

The getDate() Method

The getDate() method enables you to get the day of the month for use in a script. The value returned is a number that represents the day of the month. So, if it is the 5th of the month, the getDate() method would return 5. If it is the 22nd, the getDate() method would return 22. This method is nice because it is fairly straightforward.

The getDay() Method

The getDay() method enables you to get the day of the week; however, rather than returning a name such as Monday or Friday, it returns a number. The number represents the number of days *into* the week (0–6) rather than the day *of* the week you would commonly have in mind (1–7). So, if it is Sunday, the method returns 0; and if it is Wednesday, the method returns 3. You have to remember that it counts from 0 when you begin using it in your scripts. Many of the methods that follow will count beginning at 0.

The getHours() Method

The getHours() method enables you to get the number of hours into the day (0–23). The count begins at 0. So, when it is midnight, the method returns 0; and when it is 2:00 p.m., it returns 14.

The getMilliseconds() Method

The getMilliseconds() method enables you to get the number of milliseconds

stored in the instance of the Date object (0–999).

The getMinutes() Method

The getMinutes() method enables you to get the number of minutes stored in the instance of the Date object (0–59). Again, the counting begins at 0. So, if it is 2:00 (either a.m. or p.m.), or any hour on the dot, the method returns 0; and if it is 2:23, the method returns 23.

The getMonth() Method

The getMonth() method enables you to get the number of months stored in the instance of the Date object (0–11). This method also begins counting at 0, which makes the result a little tricky. For instance, if it is January (the month people tend to think of as 1), the method returns 0; and if it is October (the month people tend to think of as 10), the method returns 9. This is one you have to watch a little more closely when you use it in scripts, because you will need to remember to make an adjustment if you want to use numeric dates (like 10/24/2000).

The getSeconds() Method

The getSeconds() method enables you to get the number of seconds stored in the instance of the Date object (0–59). So, if the time is 2:42:23, the method returns 23; and if the time is 2:23:00, the method returns 0.

The getTime() Method

The getTime() method gets the time (in milliseconds since January 1, 1970, at midnight) for an instance of the Date object. So, if you wanted to know the number of milliseconds since that date at your current time, you could use the following code:

```
let rightnow = new Date(); let theday = rightnow.getTime();
```

This assigns the result of the method to a variable so that you can use it later if you need it in your script.

The getTimezoneOffset() Method

The getTimezoneOffset() method gives you the number of minutes that separate the local time from GMT. So, if you are six hours apart from GMT, the method would return 360 (6×60); and if you are only one hour apart, the method returns 60.

The getFullYear() Method

The getFullYear() method returns a four-digit year:

```
let rightnow = new Date(); let theyear = rightnow.getFullYear();
```

This assigns the value returned by the method to the variable `theyear`.

The UTC Methods

These methods work the same as their counterparts (for example, `getDate()` and `getUTCDate()` work the same), but return the information in terms of Universal Time rather than the viewer's local time.

Now that you have seen the methods that get values, take a look at the methods that enable you to set values for an instance of the `Date` object.

Methods That Set Values

The methods that set values work with the same types of values as the methods that get values.

The methods that enable you to set values for an instance of the `Date` object include the following:

•• <code>setDate()</code>	•• <code>setUTCDate()</code>
•• <code>setHours()</code>	•• <code>setUTCFullYear()</code>
•• <code>setMilliseconds()</code>	•• <code>setUTCHours()</code>
•• <code>setMinutes()</code>	•• <code>setUTCMilliseconds()</code>
•• <code>setMonth()</code>	<code>setUTCMinutes()</code>
<code>setSeconds()</code>	<code>setUTCMonth()</code>
<code>setTime()</code>	<code>setUTCSeconds()</code>
<code>setFullYear()</code>	

To set these, you send them a numeric argument based on the time or date you want to use. For instance, if you wanted to set the day of the month for an instance of the `Date` object, you could use the following code:

```
let rightnow = new Date(); rightnow.setDate(22);
```

This would set the day of the month to the 22nd for the `rightnow` instance of the `Date` object.

The other methods work in the same way. In order to know what value needs to be sent to one of these methods, take a look at what type of value is returned by its counterpart in the methods that get values. The argument the method will expect will be a value like the one returned by the method.

Other Methods

The remaining methods perform various tasks that the other methods don't cover in some way.

The parse() Method

The parse() method is used to find out the number of milliseconds since January 1, 1970, at midnight for a date string (such as Dec 12, 1999) input as an argument. This is often used with the setTime() method since it needs an argument in milliseconds to set the time. You could use the parse() method to find the number of milliseconds since January 1, 1970, for the date Dec 12, 1999 at midnight (the rightnow instance of the Date object will use this as the date that all of the methods will use to return values), as shown in the following code:

```
let rightnow = new Date();  
let thenum = Date.parse("Dec 12, 1999") ← This method can be used directly from the Date object  
rightnow.setTime(thenum);
```

This code parses the date into a number of milliseconds, and then sends it to the setTime() method used with the rightnow instance of the Date object.

The toString(), toDateString(), toTimeString(), toLocaleDateString, and toLocaleTimeString() Methods

These methods return a string representing the date and time, or a portion thereof. For instance, the toString() method returns a date in string format. You can use it to get a formatted date for an instance of the Date object, as shown in the following code:

```
let rightnow = new Date(); let thedate = rightnow.toString();
```

This will assign a date string value to the variable thedate. The value of the string depends on what browser the viewer is using to view the page. It can then be written to the page or used with other methods of the Date object in a script.

The toLocaleString() Method

The toLocaleString() method returns a date string in the format of the viewer's locale. You can use it to get the locale format for an instance of the Date object, as shown in the following code:

```
let rightnow = new Date(); let thedate = rightnow.toLocaleString();
```

This will assign a date string value to the variable thedate. The value of the string depends on what browser the viewer is using to view the page. It can then be written to the page or used with other methods of the Date object in a script.

Now that you have the methods down, see if you can have a little fun with

the Date object.

How About Some Date Scripts?

With the technical overview out of the way, you are ready to create some scripts that use the methods of the Date object. First you will write a script to display the date on the page, and then you will create a script for a simple status bar clock.

Write the Date on the Page

To write the date on the page, you need to use some of the Date object methods to get the values you need. Suppose you want to write a date with the format of Tuesday, M/D/Y (month, day, year). To do this, you need to find out the day of the week, the month, the day of the month, and the year. You can do this using the `getDay()`, `getMonth()`, `getDate()`, and `getFullYear()` methods.

The following script will write the date to the page. First, the HTML code (save as `write_date.html`):

```
<h1>Today's Date</h1>
<div id="write_date">
<!-- call to a server-side script for backup could go here -->
</div>
<script type="text/javascript" src="write_date.js"></script>
```

Note the comment within the div element. If you want to make this accessible to browsers that do not support JavaScript, you can place a call to a PHP (or other server-side technology) script here for those lacking JavaScript. For example, if you had the page set up to parse PHP, you could use the following to display the date much like the JavaScript code will:

```
<h1>Today's Date</h1>
<div id="write_date">
<?PHP
$the_date = date(l, n/j/Y);
echo "$date";
?>
</div>
<script type="text/javascript" src="write_date.js"></script>
```

You will see that the PHP script is much shorter than the JavaScript script due to its built-in date-formatting capability. The date displayed may differ from the viewer's date as it displays the date on the Web server. An in-depth discussion of server-side technology is beyond the scope of this book, but this serves as an example of a way to provide the same basic

feature for those without JavaScript.

Next, the JavaScript code (save as write_date.js):

A new instance of the Date object is created The methods of the Date object are used to obtain the needed values

```
let rightnow = new Date(),
```

weekday = rightnow.getDay(), An array is created to hold the name of each day of the week, with the index

```
ds = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"],
```

themoth = rightnow.getMonth(), values corresponding to what will be returned by the getDay() method theyear = rightnow.getFullYear(); thedate = rightnow.getDate();

date_div = document.getElementById("write_date"); The div element is obtained by its id

```
date_div.innerHTML = ds[weekday]+ ", " + themoth + "/" + thedate + "/" + theyear;
```

The month value is incremented in order to numerical month value The innerHTML property is changed look like its standard to display the formatted date

This script sets the results of the methods to variables. It then creates an array to hold the days of the week, which are later accessed using the number returned from the `getDay()` method as the index. The script then makes an adjustment, adding 1 to the number returned by the `getMonth()` method, so that the month will show up the way you would expect it (recall that it counts months starting at 0 instead of 1, so this ensures that January is represented by the number 1 rather than 0, for example).

The formatted output is written onto the page for the viewer to see. The result of this script when run in a browser is shown in

Note that, in this case, if the date or month is in single digits, it won't add the 0 to the front. You will see how you can do this in the next section where you will create a clock.



Figure The date is shown

Create a Simple Clock

To create a simple clock, you need the hours, minutes, and seconds of the current time. To get these, you can use the `getHours()`, `getMinutes()`, and `getSeconds()` methods.

The following code will create a clock that is displayed on the page. First, the HTML code (save as `clock.html`):

```
<h1>Current Time:</h1>
<div id="my_clock">
<!-- call to a server-side script for backup could go here -->
</div>
<script type="text/javascript" src="clock.js"></script>
```

CAUTION

If you use a server-side script as a backup to the JavaScript clock, you almost surely do not want it to update every second because this could put undue strain on the Web server. In such a case, it is often best to simply display the time without updating it.

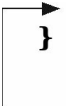
Next, the JavaScript code (save as `clock.js`):

```
function startclock() { let thetime = new Date(),
    hours = thetime.getHours(),    The methods of the Date object are used    mins = thetime.getMinutes(),
    to obtain the hours, minutes, and seconds
    secn = thetime.getSeconds(),    ap = (hours >= 12) ? "p.m." : "a.m.",    clock_div =
    document.getElementById("my_clock");
```

A variable is set so that a.m. or p.m. may be displayed after the time

```
    hours = (hours >= 13) ? hours -= 12 : hours; hours = (hours < 1) ? 12 : hours; mins = (mins < 10)
    ? "0" + mins : mins; secn = (secn < 10) ? "0" + secn : secn;
    The hours, minutes, and seconds are adjusted so that they will display like a typical 12-hour digital
    clock
```

```
    clock_div.innerHTML = hours + ":" + mins + ":" + secn + " " + ap; setTimeout(function() { startclock(); },
    1000);
```





`startclock();` ← Calling the function here starts the process The hours, minutes, seconds, and ap variables are combined and
 The script is called every second displayed as the new innerHTML using the `setTimeout()` method for the
 div element

The script creates a function that sets the results of the methods to variables. It takes the hours variable and sets the ap variable to p.m. if hours is greater than or equal to 12 and sets it to a.m. if hours is less than 12 (at this point the hours variable still holds 13 for 1 p.m., 14 for 2 p.m., and so on). Once this is done, the hours variable is adjusted so that it will display the expected value for a 12-hour clock.

The script then adjusts the values of the variables that show the minutes and seconds by adding a leading 0 when the number is less than 10. This way the clock will show 12:02:34 for 12:02:34, instead of leaving out the 0 and displaying 12:2:34 (this can also be done for the hours variable if you would like it to have a leading zero).

At the end, the function displays the output on the page. The function is initially called right after it is defined. The function is repeated at intervals of 1000 milliseconds, or 1 second. This enables the clock to stay current. The results of this script when run in a browser are shown in Figure 12-6.



The current time is displayed on the page.

Try This 12-2

Create a JavaScript Clock

pr12_2.html
prjs12_2.js

This project enables you to work more with the methods of the

Date object, as well as learn how to adjust the values that are returned so they can be used in various ways. This creates a JavaScript clock with a few more options than your simple clock in the previous section.

Step by Step

1. Create an HTML page with script tags that point to an external JavaScript file named prjs12_2.js. Add a heading that says “Current Time” and add a div element with an id of my_clock. Save the HTML file as pr12_2.html.

2. When complete, the body of the HTML file should look like this:

```
<body>
<h1>Current Time</h1>
<div id="my_clock"></div>
<script type="text/javascript" src="prjs12_2.js"></script> </body>
```

3. Create an external JavaScript file and save it as prjs12_2.js. Use it for steps 4–7.

4. Write some code that will display a clock. In this clock, include the following information: .. The time with hours, minutes, and seconds .. Whether it is a.m. or p.m.

.. The date in the form mm/dd/yyyy

5. This will be a 12-hour clock, so be sure to adjust the value of the hours so that they stay between 1 and 12. Also, note the format of the date and adjust the month and day values accordingly.
6. Begin the clock, and have it update every second.
7. When complete, the JavaScript file should look like this:

```
function startclock() { let thetime = new Date(),    hours =
thetime.getHours(),    mins = thetime.getMinutes(),    secn
= thetime.getSeconds(),    ap = (hours >= 12) ? "p.m." :
"a.m.",    themonth = thetime.getMonth(),    thedate =
thetime.getDate(),    theyear = thetime.getFullYear(),
    clock_div = document.getElementById("my_clock");    hours = (hours >= 13) ?
hours -= 12 : hours;
    hours = (hours < 1) ? 12 : hours;
    mins = (mins < 10) ? "0" + mins : mins;    secn = (secn < 10) ? "0" +
secn : secn;
    themonth++;
    clock_div.innerHTML = hours + ":" + mins + ":" + secn + " " + ap + " " +
                                themonth + "/" +
thedata + "/" + theyear;
    setTimeout(function() { startclock(); }, 1000);
} startclock();
```

8. Save the JavaScript file and open the HTML file in your browser. The time and date should appear on the page.

Try This Summary

In this project, you used your knowledge of the Date object. Using the methods of the Date object, you created a clock with a date that appears on the Web page.

Continuing Project

In Chapter 11, you used Node.js to retrieve data from a PostgreSQL database (your mywebapp database). You also used Node.js to create a simple Web server and were able to open a test page in your browser. In this chapter, you will further your project by combining the PostgreSQL query you made in Node.js with the Web server you wrote. You will then add some calculations that can be useful down the road when you begin displaying more information in the browser.

First, update your server.js file in your project folder so that it looks like the following code. This simply adds in the pg module and uses it. This will eventually allow you to get data from the database and then use any data you need to update the resulting Web page (for now you will just look at the Node.js console to see the results). Here is the new code for server.js:

```
const express = require('express');
const app = express(); const router =
express.Router(); const path = require('path');
const {Pool} = require('pg');
const connectionInfo = `postgres://mywebappuser:12345@localhost/mywebapp`;
const pool = new Pool({connectionString: connectionInfo});
```

```
app.use(express.static(path.join(__dirname, 'public')));
```

```
function getBowlerData() {    pool.query(
    `SELECT * FROM bowlers`,
    [],
    function (err, result) {
    if (err) {
        console.error(err);
    }
    }
    // This is where you will add new code
    }
```

```
);
```

This is the code that is run when the query completes

```

}      successfully. Here, you can use the data to log it, display it,  make calculations before displaying, and so on.
router.get('/', function (req, res) {
  getBowlerData();
  res.sendFile(path.join(__dirname, 'index.html'));
});
app.use('/', router);

let server = app.listen(3000, function() {
  console.log("App Server via Express is running on port 3000");
  console.log("To end, press Ctrl + C"); });

```

Notice the addition of the `getBowlerData` function and the comment indicating where code will be added. Save the updated file with the same name, `server.js`. In the following code examples, you will simply see the code to be added where this comment is, and when done, the complete file will be displayed again with the completed code.

Getting to the Needed Data

When you retrieved the data in Chapter 11, you simply logged the result to the console, which displayed an array of objects with bowler information. This time, you will go through each of the bowler objects in the array and make calculations based on the bowler attributes.

To get to the data, you will need to loop through the results array. This can be done a number of ways, most commonly using a `for` loop or using `forEach()`. For ease of reading the code, `forEach()` will be used. Where you have the comment indicating the location of new code, add the following code:

```

result.rows.forEach(function (bowler) { ←
  console.log(`Bowler Name: ${bowler.first_name} ${bowler.last_name}`); });

```

The `forEach` iteration is started, and a function is run for each item in the `result.rows` array returned from the database query.

The argument `bowler` in the function represents the value of the item for the current iteration of the array. So, the first time the function is run, the first array item's value is used, in this case `result.rows[0]`. This is the first bowler object returned from the database query. Thus, referring to this object as `bowler` lets you use an easy-to-read format to access the properties. Instead of `result.rows[0].last_name`, you can refer to `bowler.last_name`. Note that a `for` loop is just fine and may be preferred in some situations, but for this project a `forEach` fits nicely.

As you can see the `first_name` and `last_name` properties of the object are used to log each bowler's name to the console. You can use this same technique to access any of the properties, which you will do in the next section!

Running Some Calculations on the Results

Now that you can easily access the data from the query, you can make some calculations for each bowler based on the bowler's attributes. For your app, you want to give each bowler an overall rating, which will simply be an average of each of the bowler's attributes rounded to the nearest whole number. Bowler attributes are based on a 0-100 system, so a bowler with a 100 overall rating would have the best possible rating.

To begin, you need to calculate the average of each bowler's attributes. To do this, you can add them all together and divide by the number of attributes (three in this case). So, adjust your code to look like this:

```
result.rows.forEach(function (bowler) {  
  let overallAvg = (bowler.accuracy + bowler.power + bowler.consistency) / 3;  console.log(`Bowler Name:  
${bowler.first_name} ${bowler.last_name}`);  
  console.log(`Bowler Overall: ${overallAvg}`); });
```

Start your Web server from a console window (`node server.js`) and open the URL `http://localhost:3000` in your browser. Look at your console window (not the Web page) to view the result. Notice that it doesn't look particularly nice:

```
Bowler Name: Coach Purswell  
Bowler Overall: 76.33333333333333  
Bowler Name: Doctor Gutter  
Bowler Overall: 54.666666666666664  
Bowler Name: Cranky Shoes  
Bowler Overall: 77.33333333333333  
Bowler Name: Slow Roller  
Bowler Overall: 71.33333333333333  
Bowler Name: Prize Winner  
Bowler Overall: 96.66666666666667
```

The decimals are really long. This can be fixed by limiting the number of decimal places displayed. For this app, you will simply round to the nearest whole number to achieve this, which can be done with the `Math.round()` method you learned earlier in this chapter. This time, adjust your code to look like this:

```
result.rows.forEach(function (bowler) {  
  let overallAvg = (bowler.accuracy + bowler.power + bowler.consistency) / 3;
```

```

let overallRating = Math.round(overallAvg);
←
console.log(`Bowler Name: ${bowler.first_name} ${bowler.last_name}`);
    console.log(`Bowler Overall: ${overallRating}`);
    });
←

```

The variable to display is changed from overallAvg to overallRating.

A new variable, overallRating, is created and given the value of the overallAvg rounded to the nearest whole number. Shut down the Node server (ctrl-c), restart (node server.js), and refresh the Web page. If you look at the console window now, you should see a cleaner result:

```

Bowler Name: Coach Purswell
Bowler Overall: 76
Bowler Name: Doctor Gutter
Bowler Overall: 55
Bowler Name: Cranky Shoes
Bowler Overall: 77
Bowler Name: Slow Roller
Bowler Overall: 71
Bowler Name: Prize Winner
Bowler Overall: 97

```

Now you can easily see which bowlers have the best overall rating! As expected, Doctor Gutter is rated lowest, while Prize Winner has a stellar rating at 97. As you progress with this app, you will be able to display this information on the Web page instead!

If you had any trouble getting this to work or just want to see the full code for the server.js file upon the completion of the new code, here it is:

```

const express = require('express');
const app = express(); const router = express.Router();
const path = require('path'); const {Pool} =
require('pg');
const connectionInfo = `postgres://mywebappuser:12345@localhost/mywebapp`;
const pool = new Pool({connectionString: connectionInfo});

app.use(express.static(path.join(__dirname, 'public')));

function getBowlerData() {    pool.query(
    `SELECT * FROM bowlers`,
    [],
    function (err, result) {        if (err) {
        console.error(err);
    }
    result.rows.forEach(function (bowler) {
        let overallAvg = (bowler.accuracy + bowler.power +
        bowler.consistency) /
3;
        let overallRating = Math.round(overallAvg);
        console.log(`Bowler Name:
        ${bowler.first_name}
        ${bowler.last_name}`);
    });
}

```



```

        console.log(`Bowler Overall: ${overallRating}`);
    });
}
);
}

router.get('/', function (req, res) {
    getBowlerData();
    res.sendFile(path.join(__dirname, 'index.html'));
});
app.use('/', router);

let server = app.listen(3000, function() {
    console.log("App Server via Express is running on port 3000");
    console.log("To end, press Ctrl + C");
});

```

Note that you will want to ensure that the Postgres connection string is correct with your own username and password.



Which of the following correctly assigns the day of the week for an instance of the Date object named rightnow to a variable named weekday?

- A. let weekday= rightnow.getDate();
- B. let weekday= rightnow.getDay();
- C. let weekday= right now.getDay();
- D. let weekly= rightlater.getMinutes();

9 HANDLING THE STRING

The String Object

As just explained, one way to create a String object is to use the new keyword, as you've done with other objects previously. The syntax is shown here:

```
let instance_name = new String("string value here");
```

You replace *instance_name* with the name you want to use for the instance of the String object. You then replace *string value here* with the string of characters to use as the new String object.

So, if you want to create an instance of the String object named guitar_string, you could use the following code:

```
let guitar_string = new String("G");
```

This script creates an instance of the String object for the string "G".

The String Literal

You can create a string literal just by assigning a string value to a variable. This technique is a bit shorter than creating a String object using the new keyword and still allows you to use all the methods of the String object (as well as one of the properties).

A string literal is created in the code that follows. Notice that the code assigns a string value to a variable.

```
let guitar_string = "G";
```

This makes the string "G" a string literal, which you know as a regular text string. With text strings, you're also allowed to use the properties and methods of the String object.

What's the Difference?

The difference between a String object and a string literal is that a regular text string has the value of the string itself, and it can be compared against another string easily, as in the following code:

```
let guitar_string1 = "E";
```

```
    Both variables have the same string value let guitar_string2 = "E"; if (guitar_string1 === guitar_string2) {
```

```
}
```

```
    alert("The strings are the same!");
```

```
    else
```

<pre> alert("The strings are not the same!"); </pre>	

```

{
    // The strings are compared
}

```

Because this code uses regular string literals, the result is what you’d expect. An alert says that the strings are the same.

However, if you used String objects to run through the same if block, you would see something unexpected. The code that follows uses String objects instead:

let guitar_string1 = new String("E"); The String objects are given let guitar_string2 = new String("E"); the same string values if (guitar_string1 === guitar_string2) {

```

} The values are compared,

```

<pre> window.alert("The strings are the same!"); window.alert("The strings are not the same!"); </pre>	

```

{
    // but an unexpected answer is the result
}

```

This time the alert would tell you that the strings are not the same, even though the string values are both “E”—because a String object is an object value and not a literal value. Objects aren’t going to be equal to one another in the same way regular text strings would be (objects contain name-value pairs, while literal values only contain the value). For most purposes, you would probably use string literals and let them use the String object’s methods.

Property	Purpose
length	Holds the numeric value of the length of the string (its number of characters)

Properties of the String Object

A regular text string is able to use the String object’s methods because JavaScript takes the string literal and turns it into a temporary String object. Once the method’s execution is complete, it returns a string literal. This allows you to use the String object’s methods without having to create String objects.

The String object has only has one property (aside from the constructor and prototype properties, which can only be used with a String object created via the constructor function). Table 13-1 shows this property.

The length Property

This property returns the length of the string, which is the number of characters contained in the string. You can use it with both String objects and string literals. You've seen this property with other objects as well, such as the Array object. (In that case, the value was the number of elements in an array.)

The following code uses a regular text string. It writes the length of the string variable onto the page.

```
let myname = "John"; ◀— A string literal is created The length of the alert("The name has " + myname.length +  
" characters.");◀— name is alerted
```

to the viewer

Notice how the name of the variable is used like an object name, and is able to access the String properties and methods. The script writes the result to the page. Because the name has four characters, the length property has a value of 4 here.

Methods of the String Object

Some common methods of the String object are listed in Table 13-2 (grouped by function). Additional methods can be found at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String#Methods.

The charAt() and charCodeAt() Methods

The charAt() method determines which character resides at a particular position in a string. It takes a number argument representing the position of the character in the string.

Method	Purpose
charAt()	Finds out which character is at a given position in a string
charCodeAt()	Finds the character code of a character at a given position in a string
concat()	Adds two or more strings together and returns the new combined string value
slice()	Pulls out a specified section of a string value and returns a new string
substring()	Allows a portion of the string specified with a starting position and an ending position to be returned
indexOf()	Searches for a character sent as a parameter in a string: if it's found, the position of the first instance of the character is returned; otherwise, it returns -1

lastIndexOf()	Searches for a character sent as a parameter in a string: if it's found, the position of the last instance of the character is returned; otherwise, it returns -1
toLowerCase()	Converts a string to all lowercase letters and returns the result
toUpperCase()	Converts a string to all uppercase letters and returns the result
toString()	Returns the string literal value of a string
fromCharCode()	Uses character codes sent as parameters to create a new string
localeCompare()	Compares strings based on alphabetical order and returns the result
match()	Compares a regular expression and a string to see if they match
replace()	Finds out if a regular expression matches a string and then replaces a matched string with a new string
search()	Executes the search for a match between a regular expression and a specified string
split()	Separates a string into an array of strings based on a character sent as a parameter to the method
trim()	Removes white space from the beginning and end of a string

Methods of the String Object

When you want to find a character, remember that the position count begins at 0 (as with arrays) rather than 1, so the first character is at position 0. The following code shows how to get the first character in a string by using the `charAt()` method:

```
let the_text = "Character"; The charAt() method finds the character at let first_char = the_text.charAt(0); ←——
position 0 (the first character) in the string alert("The first character is " + first_char);
```

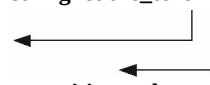
This code assigns the result of the `charAt()` method call to a variable named `first_char`, which is then used in an `alert`. The alert will tell the viewer the first character in the text string that called the method. In this case, the alert would say “The first character is C.”

If you want to find the last character, either you need to know how many characters are in the string before you use the method, or you can use the `length` property to determine the number of characters in the string. When using the `length` property, remember that it returns the number of characters, not the position of the last character.

The `length` property begins counting at 1, while you must begin counting at 0 when you use the `charAt()` method. Thus, the last character in a string will be at a position one less than the number of characters it contains. In other words, if the string has 10 characters (1–10), the last position (0–9) is at 9. For example, this code finds the last character in a string:

Subtract 1 from the length property to find the last position available in the string

```
let the_text = "Character",
```



```
position = the_text.length - 1, The charAt() method uses that last_char = the_text.charAt(position); value to find out which character alert("The last character is " + last_char); is in the last position in the string
```

This code assigns the value of the length of the string minus 1 to a variable named `position`. The `position` variable now holds the position of the last character in the string. The result of calling the `charAt()` method with the value of `position` sent as the parameter is assigned to a variable named `last_char`. Finally, an alert provides the last character in the string, which is `r`. Thus, the viewer gets an alert saying “The last character is r.”

The `charCodeAt()` returns the character code for the character at the positions sent as the argument (for further discussion on character codes, you can refer to Chapter 5 or visit www.scripttheweb.com/js/ref/javascript-character-codes.html). For example, you could use the following code:

The `charCodeAt()` method

```
let the_text = "Hello"; finds the character code for let char_code = the_text.charCodeAt(0); ← the character at position 0 alert("The character code is " + char_code); (the first character) in the string
```

An alert would then tell you the character code at position 0 in the string (the letter `H`), which is 72.

The `concat()`, `slice()`, and `substring()` Methods

These methods are provided to allow you to alter the value of a string in various ways.

The `concat()` Method This method works much like the Array object’s `concat()` method. It combines the text string that calls the method with any number of other text strings sent as arguments and returns the combined value. The text string calling the method comes first in the order, while the strings sent as arguments are added in order afterward. The following code shows an example that combines three strings using the `concat()` method:

```
let string1 = "I went to the store ",
    string2 = "then ",
    string3 = "I played a video game";
alert(string1.concat(string2, string3)); ← The three strings are combined in order from left to right
```

This code combines the strings in the order `string1`, `string2`, and then `string3`. The result is an alert that says, “I went to the store then I played a video game.” Note that MDN recommends very strongly that you use assignment operators instead of `concat` to do this. See [https:// developer.mozilla.org/en-](https://developer.mozilla.org/en-)

[US/docs/Web/JavaScript/Reference/Global_Objects/String/concat#Performance](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/concat#Performance) for details.

The slice() Method This method slices out a portion of a string and returns the value that was sliced. It works much like the slice() method of an array. The first argument tells it the position at which to start slicing, while the second argument is one greater than the position where it will stop. For instance, take a look at the code that follows:

```
let the_text = "Do not cut this short!"; let shorter_string = the_text.slice(0,7);  
alert(shorter_string);
```

This code slices the string from position 0 through position 6. Position 7 is where the *c* is in “cut”; but it isn’t sliced because the argument that tells JavaScript where to end the slicing is one greater than the position of the last character to slice. Thus, the alert will say “Do not.” Note that the original the_text value remains the same.

The substring() Method This method works much like the substr() method, but it allows you to send arguments for the beginning position and the ending position of the portion of the string you want to remove. It then returns the removed portion as a new string.

For example, take a look at the code that follows. Rather than specifying the number of characters to remove, you give an ending position. The characters are removed beginning at the starting position and ending at one less than the ending position. (Remember the slice() method.)

```
let the_text = "Do not cut this short!"; let shorter_string = the_text.substring(3,7);  
alert(shorter_string);
```

You remove everything between the beginning of the string and the beginning of the word “cut.” The alert will say “not.”

NOTE

The slice() and substring() methods are very similar, but they do have their differences in specific cases. To see more about this, go to [http://stackoverflow.com/questions/2243824/ what-is-the-difference-between-string-slice-and-string-substring-in-javascript](http://stackoverflow.com/questions/2243824/what-is-the-difference-between-string-slice-and-string-substring-in-javascript)

The fromCharCode() Method

The fromCharCode() method creates a string from a series of character codes sent as parameters to the method. The charCodeAt() method returns a numeric code for the character at a given position. This is the type of value you must send to the fromCharCode() method. Also, fromCharCode() is

called directly from the String object rather than from an existing string, because it is piecing together a string on-the-fly and doesn't require one to run. Instead, it uses the parameters sent to it to return a string.

So, if you want to alert the text string "HI" to the viewer, you could use the example code shown here:

```
window.alert(String.fromCharCode(72,73));
```

This code takes the first parameter (the character code 72) and converts it to an *H*. It then takes the second parameter (the character code 73) and converts it to an *I*. The two are combined in the order they were sent to form the string "HI," which is sent as an alert to the viewer.

[The localeCompare\(\) Method](#)

The localeCompare() method compares two strings and returns a number indicating whether or not the string comes before or after the string argument in alphabetical order. The method will use different ways of calculating this, depending on the locale (country and language) of the browser. It returns one of the following values:

- If the string should come before the argument alphabetically, it returns a negative number.
- If the strings are equal, it returns 0.
- If the string should come after the argument alphabetically, it returns a positive number.

For example, you could use the following code to determine whether the string "orange" comes before or after another string alphabetically:

```
let comp_string = "orange";  new_string = "apple";
pos = comp_string.localeCompare(new_string);
if (pos < 0) {
  alert(comp_string + " comes before " + new_string);
}
else if (pos > 0) {
  alert(comp_string + " comes after " + new_string);
} else {
  alert("The strings are equal"); }
```

In this case, the viewer will get an alert saying, "orange comes after apple."

[The indexOf\(\) and lastIndexOf\(\) Methods](#)

The indexOf() method finds out where a certain character or string begins in a string. It returns the position of only the first occurrence of the character or

string that is sent as the argument. If the character or string isn't found in the string value, a value of -1 is returned.

The following code looks for the letter *C* in the string "Cool":

```
let the_text = "Cool"; let position = the_text.indexOf("C"); alert("Your character is at position " + position);
```

Remember that the position count begins at 0, so when it finds *C* as the first character in the string, it returns 0. Thus, the alert will say "Your character is at position 0."

Note that the method is case sensitive, so *C* and *c* are two different characters to JavaScript in this case. Thus, the code that follows returns -1 (telling you the character isn't in the string), even though an uppercase *C* is in the string.

```
let the_text = "Cool"; let position = the_text.indexOf("c"); alert("Your character is at position " + position);
```

The alert would now say "Your character is at position -1 ."

If you want to check for that -1 to keep from getting it as a position, you could use this code to send a different alert in case the character you want to find isn't in the string:

```
let the_text = "Cool"; let position = the_text.indexOf("c");
if (position === -1) {
  alert("Your character is not in the string!");
} else {
  alert("Your character is at position " + position);
}
```

This time, the if statement checks to see whether the method returns -1 to the position variable. If so, the alert says "Your character is not in the string!" Otherwise, the regular alert will tell you the position. In the previous code, the lowercase *c* isn't in the string, so the "Your character is not in the string!" alert appears.

The `indexOf()` method returns the position number only for the first occurrence of the character you send as the parameter. So, if you use the code that follows, you will be alerted that your character is at position 1, even though it's also at position 2:

```
let the_text = "Cool"; let position = the_text.indexOf("o");
if (position === -1) {
  alert("Your character is not in the string!");
} else {
  alert("Your character is at position " + position);
}
```

If you use the second argument to the `indexOf()` method, the search for

your character or string will begin at that position rather than from the 0 position. Thus, one way to find that second “o” would be to skip past the first one at position 1 and start looking at position 2.

```
let the_text = "Cool";  
let position = the_text.indexOf("o",2);  
if (position === -1) {  
  alert("Your character is not in the string!");  
} else {  
  alert("Your character is at position " + position); }
```

This time, the method returns 2 as the result, since it finds it right at the specified starting position.

NOTE

If you want to find all occurrences of a string, you may wish to use a regular expression, discussed later in this chapter.

The `lastIndexOf()` method finds out where the last instance of a certain character or string is located in the string. It returns the position of only the last occurrence of the character or string that is sent as the argument. If the character or string isn’t found in the string value, a value of `-1` is returned.

The following code looks for the letter `C` in the string “Cool Cruising Car”:

```
let the_text = "Cool Cruising Car"; let position = the_text.lastIndexOf("C"); alert("Your  
character is at position " + position);
```

This code will display an alert that tells the viewer “Your character is at position 14.”

The `match()`, `replace()`, `search()`, and `split()` Methods These methods all perform some type of pattern matching.

- The `match()` method compares a regular expression and a string to see whether they match.
- The `replace()` method finds out if a regular expression matches a string and then replaces a matched string with a new string. Note that you can also do a replace with strings. This will only replace the first instance, but it is something you can do if you only need to replace the first instance, and it can be done without regex.
- The `search()` method executes the search for a match between a regular expression and a specified string.
- The `split()` method creates an array of string elements from a string based on the character sent as an argument.

The first three methods deal with regular expressions, so you will learn more about them later in this chapter in the “Regular Expressions” section.

The split() Method The split() method uses the character sent as an argument as a separator on which to split the single string into a number of parts. It returns an array with each piece of the string minus the separator character.

For instance, the code that follows has a string with a bunch of colons in it:

```
let the_text = "orange:apple:pear:grape", split_text = the_text.split(":"),
end_count = split_text.length;
for (let count = 0; count < end_count; count++) { document.write(split_text[count] + "<br>");
}
```

The string assigned to the the_text variable consists of the names of several fruits separated by colons. The next line creates an array named split_text by using the split() method on the text string the_text. The argument sent is a colon, which is what is used to separate the string into array items. In this case, the array ends up with four items.

NOTE

The separator character that is sent as a parameter won't end up in the array: it serves only as a divider between the text so that the method knows where to begin and end each item.

The next line gets the length of the split_text array and places that value in the variable end_count. This information is then used to loop through the new array and print the items on the page.

Figure 13-1 shows the result of this script in a browser, which is a listing of fruit names.

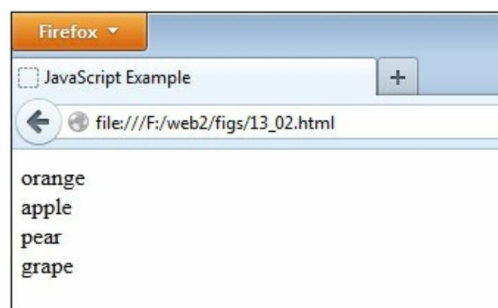


Figure 13-1 The array elements created using the split() method are

printed on the page.

The toString() Method

The toString() method returns a string literal value for a string. Here's an example of how you can use this method:

```
let string_obj = new String("Cool"); let string_lit = string_obj.toString();
```

This code takes the String object and uses the toString() method to get its string literal value. It then assigns that value to the string_lit variable.

The toLowerCase() and toUpperCase() Methods

The toLowerCase() method returns the string in all lowercase letters. Take a look at this code:

```
let the_text = "I FEEL CALM, REALLY."; alert(the_text.toLowerCase());
```

This code alerts the string in all lowercase letters, like this sample text:

i feel calm, really.

The toUpperCase() method returns the string in all uppercase. Here's an example:

```
let the_text = "I am yelling!"; alert(the_text.toUpperCase());
```

This code alerts the string in all uppercase letters, like this sample text:

I AM YELLING! The trim() Method

The trim() method removes any excess white space from the beginning and end of a string and returns the result. The original string is left intact. For example, look at the following code:

```
let text = " Once upon a time... ", trimmed_text = text.trim(); alert(trimmed_text);
```

The spaces from the beginning and end will be removed, and the string "Once upon a time..." is alerted to the viewer. Figure 13-2 shows the result of this script when run in a browser.

NOTE

This method does not work in Internet Explorer prior to version 9.

That's the last of the methods! Now you are ready to test what you've learned.

up the information stored in the cookie to make use of it. Cookies store information in long text strings, usually with some character as a separator. This is just one example of when the `split()` method can be useful to you.

Q: A lot of those methods just add tags around a text string. Couldn't I just write out the HTML for that? It seems easier.

A: These methods are not used often any more, due to the ability to use `innerHTML` or to add DOM nodes to the document. Also, they could cause the HTML code to be invalid, as some of them create tags that are now deprecated in HTML5.

Try This 13-1 Use `indexOf()` to Test an Address

pr13_1.html
prjs13_1.js

In this project, you practice using the `indexOf()` method by creating a script that performs a very basic test on an e-mail address that the viewer enters.

Step by Step

1. Create an HTML page that points to a JavaScript file named `prjs13_1.js`. Create a form with an id of "getemail." Add a text box with an id of "email" and a submit button labeled "Submit E-mail." Save the HTML file as `pr13_1.html`.
2. When complete, the body of the HTML file should look like this:

```
<body>
<form id="getemail">
<label for="email">Enter E-mail</label>
<input type="text" id="email"><br><br>
<input type="submit" value="Submit E-mail">
</form>
</body>
```
3. Create an external JavaScript file and save it as `prjs13_1.js`. Use it for steps 4–7.
4. Assign the value of the "email" text box to a variable named `email_add`.
5. Use `indexOf()` to see if the address has an at (@) character in it and to

see if the address has a dot (.) character in it.

6. If the address has both an at (@) character and a dot (.) character, send an alert thanking the viewer. If not, send an alert to the viewer saying that he or she needs these characters and to try again.
7. When complete, the JavaScript file should look like this:

```
let getemail = document.getElementById("getemail");
getemail.addEventListener("submit", function(event) { let email_add =
document.getElementById("email").value;
event.preventDefault();
if ( (email_add.indexOf("@") >= 0) && (email_add.indexOf(".") >= 0) ) {
    alert("Thank you!");
} else {
    alert("The @ and . characters are required in the e-mail address.");
}
}, false);
```

8. Save the JavaScript file and open the HTML file in your browser to try out the script.

Try This Summary

In this project, you used your knowledge of the `indexOf()` method to test an e-mail address entered by the viewer for certain characters. If one of the characters is missing, an error alert is sent to the viewer. Otherwise, an alert is sent thanking the viewer.

Using Cookies

A *cookie* is a small text file that is stored on the end user's computer. It can be referenced any time the viewer returns to your site, provided the cookie hasn't been deleted or expired. Of course, if the viewer doesn't accept cookies, then a cookie won't be able to be set or referenced later. Keep the following points in mind when using cookies:

- Cookies must be 4KB (4000 characters) each or less.
- A browser can accept up to only 20 cookies from a single domain.
- If a number of viewers don't accept cookies, this eliminates any advantages of your cookie(s) to those viewers.

Cookies can help users browse your site more effectively. For instance, if you use a script on your main page that sends one or more alerts while the page is loading, you won't want that to happen every time the viewer

goes to another page on your site and then returns to the home page. It would likely be so aggravating that you wouldn't have a visitor after it happened a few times. The alerts pop up each time the page loads because HTTP lacks state persistence. Cookies fill that gap because they allow the browser to "remember" that the viewer has seen the pop-up alert before and thus not display it on subsequent page visits.

Setting a Cookie

Setting a basic cookie is as easy as giving a value to the cookie property of the document object. The only restriction is that you can't have spaces, commas, or semicolons inside the string. For example, you can set a cookie that will store a name so that you can identify it if you set more than one cookie later. The following code sets a basic cookie:

```
function set_it() {  
    document.cookie = "name=tasty1"; }
```

CAUTION

When setting a cookie, remember not to use spaces, commas, or semicolons inside the string that sets the cookie data.

The preceding code sets a cookie with a value of name=tasty1 when the function is called. You can set any delimiter you want, though (or none at all, but setting delimiters allows you to store multiple values like a query string), so the following code would work as well:

```
function set_it() {  
    document.cookie = "name:tasty1";  
}
```

Adding the additional information isn't very difficult as long as the value does not need a space, as shown in the following code:

```
function set_it() {  
    document.cookie = "name=tasty1&fav=Sugar"; }
```

As you can see, the value of the cookie is being formatted in name-value pairs, and each pair is separated with an ampersand (&). Again, you can choose any type of separators you want. The following code is fine as well:

```
function set_it() {  
    document.cookie = "name:tasty1|fav:Sugar"; }
```

In this case, the names and values are separated with colons, while the pipe (|) symbol separates them into pairs. You can use anything that you are comfortable using.

The encodeURIComponent() Method

If you want to use spaces, commas, or semicolons in your cookie, you need a way to “escape” them so that they are translated into something a cookie accepts.

A cookie will accept character codes, as a server-side program often does. They may look like %20, %41, or something similar. To turn spaces, commas, and semicolons into these codes, you must use the JavaScript encodeURIComponent() method. It is a method under the window object, so, as you have seen in previous chapters, you can just use encodeURIComponent() rather than window.encodeURIComponent().

The following code shows how you could use the encodeURIComponent() method to set a cookie with a space in it:

```
function set_it() {
  let thetext = "name=tasty1&fav=" + encodeURIComponent("Chocolate Chip");
  document.cookie = thetext;
}
```

A string literal is created with information to be used in the cookie

The value of the escaped string is placed into a cookie }

The thetext variable is set to include the string you want to use in the cookie. The newtext variable is set to hold the result of using the encodeURIComponent() method on thetext. The escaped text is then used as the string for the cookie, which will now have the code for the space character in it (when you want to use this data, you will need to unescape it). **Allowing User Input**

By using the encodeURIComponent() method, you can prompt the viewer for the information, escape it, and then use it in the cookie. The following code shows one way to do this to get the viewer’s favorite type of cookie:

```
function set_it() {
  let thefav = prompt("Enter your favorite type of cookie", "");
  thetext = "name=tasty1&fav=" + encodeURIComponent(thefav);
  document.cookie = thetext;
}
```

The viewer is able to enter information into this prompt

The value entered in the prompt is then added to a string to be used for a cookie

The string is placed into a cookie

Now the viewer can help decide the information that will be set in the cookie, and you can use the information your viewers enter on your site on their next visit.

Setting an Expiration Date

Adding an expiration date to a cookie will keep it from being deleted once

```

function set_it() {
    let thetext = "quote=" + encodeURIComponent("I have a quote");
    expdate = ";expires=Mon, 30 Mar 2015 13:00:00 UTC";
    thetext += expdate;
    document.cookie = thetext;
}

```

The diagram illustrates the execution flow of the `set_it()` function. It shows the assignment of `thetext` and `expdate`, the concatenation of `expdate` to `thetext`, and the final assignment of the combined string to `document.cookie`. Arrows indicate the sequence of operations and the flow of data between variables.

`expires=date (in UTC format)`

In the code, an expiration date for the cookie was added by adding the date in UTC format. Then the result is added to the variable that will be used to set the string for the cookie.

Reading a Cookie

```
function read_it() {
  let mycookie = document.cookie; }
```

name=tasty1&fav=Chocolate%20Chip

Next, you must find a way to extract the information you need from the

cookie. Assuming the cookie contained the string just used as an example (name=tasty1&fav=Chocolate%20Chip), the string would now look like this:

name=tasty1&fav=Chocolate Chip

Notice that the text is divided in two different ways. The ampersand divides the string into name-value pairs, while the equal signs divide the name-value pairs into their names and values.

You can use that variable to create a new array by splitting the string on the ampersand character:

```

function read_it() {
  let mycookie = document.cookie;
  thepairs = my_cookie.split("&");
}

```

The value of the cookie
The cookie string is split into an array on the ampersand (&)

The preceding code splits the string into two array values:

thepairs[0] with a value of name=tasty1 thepairs[1] with a value of fav=Chocolate Chip

You split each of these into a new array that will have a name and a value.

Therefore, you need some code like this:

```

function read_it() {
  let mycookie = document.cookie;
  thepairs = my_cookie.split("&");
  name = thepairs[0];
  fav = thepairs[1];
  name_propval = name.split("=");
  fav_propval = fav.split("=");
}

```

The values of the array elements are assigned to variables
The new variables are split on the equal sign

Now you have all the information you need using the name_propval[] and fav_propval[] arrays:

name_propval[0] with a value of name name_propval[1] with a value of tasty1
fav_propval[0] with a value of fav fav_propval[1] with a value of Chocolate Chip

You can use them in any way you like, such as placing them in alerts. You can now also decode the values input by the user, as shown in the following example:

```

function read_it() {
  let mycookie = document.cookie;
  thepairs = my_cookie.split("&");
  name = thepairs[0];
  fav = thepairs[1];
  name_propval = name.split("=");
  fav_propval = fav.split("=");
  alert("The cookie's " + name_propval[0] + " is " +
    decodeURIComponent(name_propval[1]));
  alert("My favorite type of cookie is " + decodeURIComponent(fav_

```

The values obtained from the cookies are decoded and used

```
propval[1]));  
}
```

You might want to be sure `document.cookie` exists before you try to run all the code in the function. Any cookie you set to expire later can be read, but those without the cookie may get an error. By making the check, you can ensure the existence of the cookie or send the viewer to the function that sets the cookie. To do that, add an if/else block around the code in the function, as shown here:

```
function read_it() { let mycookie = document.cookie;  
thepairs = my_cookie.split("&");  
    name = thepairs[0];    fav = thepairs[1];    name_propval  
= name.split("=");    fav_propval = fav.split("="); alert("The  
cookie's " + name_propval[0] + " is " +  
decodeURIComponent(name_  
    propval[1]));  
    alert("My favorite type of cookie is " + decodeURIComponent(fav_propval[1]));  
}  
if (document.cookie) {  
    read_it();  
} else { set_it();  
    read_it(); }
```

The preceding code assumes that the `set_it()` function exists and will set the cookie if needed. It also assumes that this cookie is the only cookie set from this domain and that it is formatted a particular way. If you plan to use multiple cookies, you will need to do additional testing to ensure you grab the cookie you need. You can see webmonkey.com/2010/02/advanced_javascript_tutorial_lesson_2/#Reading_and_Writing_Multiple_Cookies for more information on this.

Try This 13-2

Remember a Name

pr13_2.html
prjs13_2.js

In this project, you use a cookie to remember a name when a visitor returns to the page.

Step by Step

1. Create a new HTML document that uses an external JavaScript file named `prjs13_2.js`. Create a div with an id of “greeting”. Save the HTML file as `pr13_2.html`.

2. Create an external JavaScript file and save it as `prjs13_2.js`. Use it for the remaining steps.
3. Create a function named `set_it()` that will set a cookie. Set the expiration to a future date.
Allow the viewer to enter a name in a prompt and then use the viewer's entry in the cookie.
4. Create a function named `read_it()` that will check whether the cookie exists and, if so, read the cookie and write the name on the page in the following format (replace `<name>` with the name read from the cookie):
`Welcome, <name>!`
5. If the cookie exists, call the `read_it()` function. If the cookie doesn't exist, call the `set_it()` followed by the `read_it()` function.
6. Save the JavaScript file and open the HTML file in your browser.
Enter your name and get your greeting. If you close your browser and open the page again, you should see your name without the need to enter it again.

Try This Summary

In this project, you used your knowledge of setting and reading cookies with JavaScript. You created a Web page that remembers the viewer's name when the viewer returns to the Web page.

Using Regular Expressions

Regular expressions give you much more power to handle strings in a script. They allow you to form patterns that can be matched against strings, rather than trying to use the String object's methods, which may make it more difficult to be precise.

For example, you may want to know whether the value entered in a text box for an e-mail address included at least one character at the beginning, followed by an at (@) symbol, followed by at least one character, followed by a dot (.), followed by at least two more characters (matching a traditional e-mail address like `jon@jon.com` or the shortest type of e-mail address, `j@j.jj`). The String object's methods don't provide a neat and clean way to perform this task (although with enough

tinkering, it may be possible). However, a regular expression can shorten the task or even turn a match that seemed impossible with the String object's methods into one that can be completed.

Creating Regular Expressions

To create regular expressions, you must create an instance of the JavaScript RegExp object or a RegExp literal. To create a RegExp literal, you just assign the regular expression to a variable. Instead of using quotation marks to surround the expression, you use forward (/) slashes, as shown here:

```
let varname = /your_pattern/flags;
```

You replace *varname* with the name you want to use for a variable and replace *your_pattern* with the regular expression pattern of your choice. You can follow the last slash with one or more flags (which are discussed in the upcoming section “Adding Flags”).

NOTE

JavaScript uses forward slashes to let the browser know that a regular expression is between them, the same way quote marks are used to set off strings. Thus, if a forward slash is used within the regular expression, it must be escaped with a backslash in order to work properly. For instance, instead of writing `/02/03/2009/`, you would need to write `/02\03\2009/`.

The easiest regular expression pattern to create is one that looks for an exact match of characters. For instance, if you wanted to see if the sequence *our* is present in a string, you could create the following regular expression pattern:

```
let tomatch = /our/;
```

The preceding code creates a RegExp literal named `tomatch`. Now you need a string against which to test the pattern. If you test the word *our* against the expression, it's a match.

If you test *your*, *sour*, *pour*, or *pouring* against it, then it's a match. If you test *cool*, *Our*, *oUR*, *OUR*, or *souR*, then it won't be a match. So how do you perform this test?

Testing Strings Against Regular Expressions

To test a regular expression against a string, you can use the `test()` method of the RegExp object. The basic syntax is as follows:

```
regex_name.test(string_to_test);
```

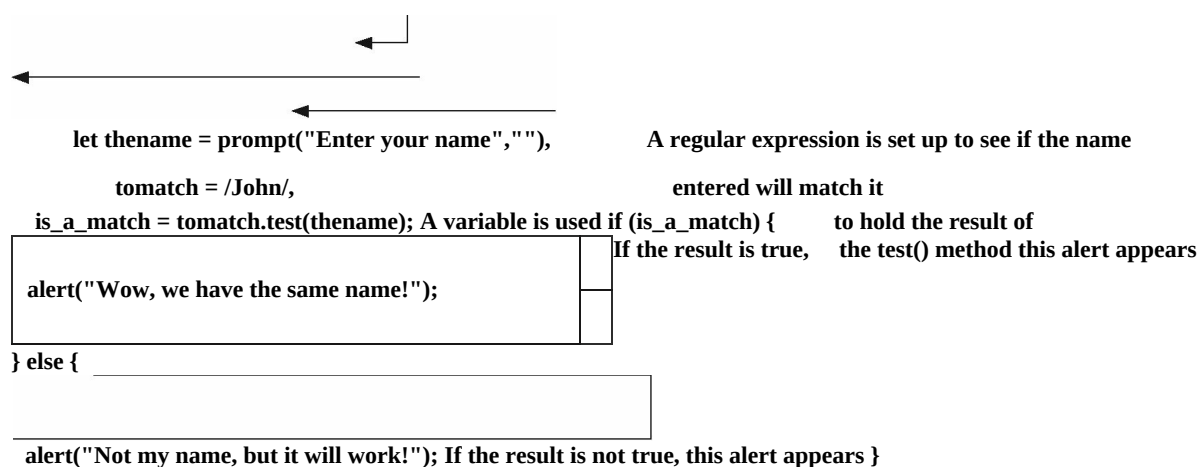
This syntax is similar to using a string method. You replace *regex_name* with the name of the regular expression and replace *string_to_test* with a string or a string variable name. For instance, look at the following example:

```
let tomatch = /our/; tomatch.test("pour");
```

This code will test the “pour” string against the regular expression named “tomatch.” It doesn’t use the result, though.

The test() method returns a Boolean value of true or false. It returns true when any portion of the string matches the regular expression pattern. Using the test() method, you can already write a short script, as shown here:

The prompt gets a name from the viewer



```

let thename = prompt("Enter your name", "");
tomatch = /John/;
is_a_match = tomatch.test(thename);
if (is_a_match) {
    alert("Wow, we have the same name!");
} else {
    alert("Not my name, but it will work!");
}

```

A regular expression is set up to see if the name entered will match it

A variable is used if (is_a_match) { to hold the result of the test() method this alert appears

If the result is true, the test() method this alert appears

If the result is not true, this alert appears }

The prompt gathers a name and holds the value in a variable. The pattern to match is John, and it is case sensitive. Thus, only an entry containing John with a capital *J* followed by lowercase *o*, *h*, and *n* will create a match and return true when it is tested (though it could contain more than just John, so entries such as Johnny or John Doe would also return true—if you want only a specific set of characters, you need to use some additional special characters, which will be discussed later in this section).

The result of the test() method is assigned to a variable named *is_a_match*. The variable is then used as the condition for the if statement. If the variable holds a value of true, then the viewer gets the “Wow, we have the same name!” alert. If it holds a value of false, the viewer gets the “Not my name, but it will work!” alert instead. Figure 13-3 shows the result of this script when “Steve” is used as the name.

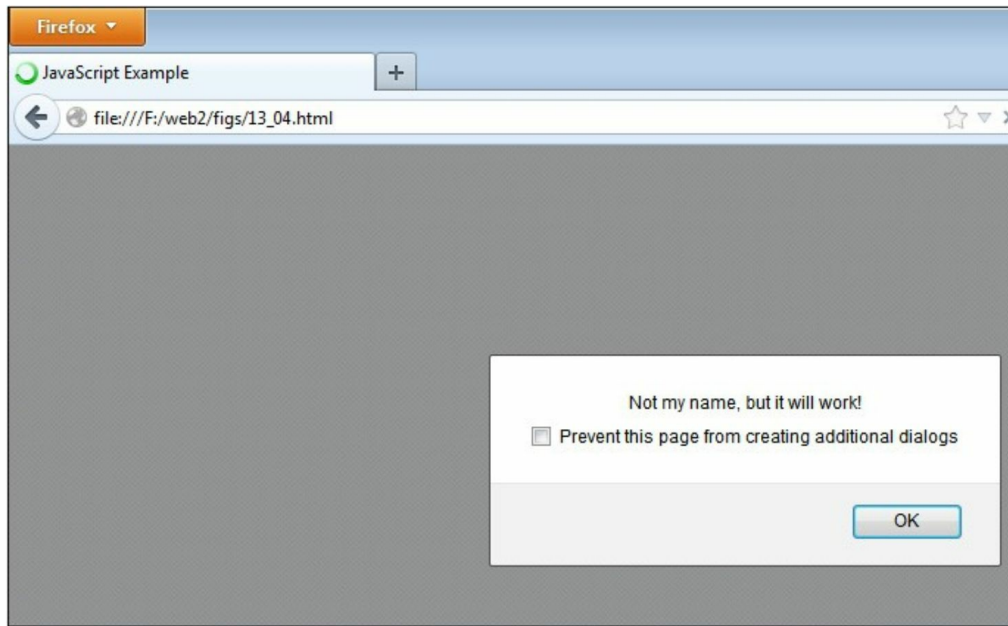


Figure 13-3 The name does not match the regular expression pattern.

If you want to shorten the script, you can just make the result of the `test()` method the condition for the if statement (rather than create another variable), as in the following code:

```
let thename = prompt("Enter your name", "");
tomatch = /John/;

if (tomatch.test(thename)) { ←————— The result of the test() method is used as the condition
    alert("Wow, we have the same name!");
} else {
    alert("Not my name, but it will work!"); }
```

Because the method returns true or false, it can be placed as the condition for the if statement on its own. (You could make it `(tomatch.test(thename)==true)` if you wanted to, though.)

Adding Flags

Flags allow you to make the match case insensitive or to look for every match in the string rather than just the first one (a global test). To add a flag, place it after the last slash in the regular expression. You can use three options, as shown in Table 13-3.

Flag(s)	Purpose
i	Makes the match case insensitive

g	Makes the match global
m	Makes the match work in multiline mode

Table 13-3 Regular Expression Flags

If you wanted to adjust the name script used previously to be case insensitive, you could add an *i* flag to the regular expression, as shown in the following code:

```
let thename = prompt("Enter your name", "");
    tomatch = /John/i; ← The i flag makes this regular
                        expression case insensitive

if (tomatch.test(thename)) {
    alert("Wow, we have the same name!");
} else {
    alert("Not my name, but it will work!");
}
```

The `test()` method will now return true as long as the pattern of John is in the string. It can be in any case, so now John, JOHN, john, and even JoHn are all matches and will cause the `test()` method to return true.

You can also use more than one flag or all three flags at once. For example, if you want to have the match be both case insensitive and multiline, you could use the following:

```
let tomatch = /John/im;
```

Creating Powerful Patterns

Although it's nice to be able to create such an exact pattern, you won't always be looking for a match that is so precise. In many cases, you will be looking to match a more general pattern, such as an entry that needs to have at least three characters or that needs to have two characters of any type followed by a special character.

By using special characters in your expressions, you can create the type of patterns you need to match a given sequence you want. JavaScript regular expressions use the syntax of Perl regular expressions as a model. Thus, if you've used regular expressions in Perl, much of this material will be familiar. Table 13-4 lists a number of the characters to help you create your patterns.

As you can see, extensive options exist for creating the pattern you need. Now you could easily verify strings according to the standards you

decide to set.

Character	Purpose	Example
^	Matches only from the beginning of a line	<code>/^c/</code> matches <i>c</i> in <i>corn</i> <code>/^c/</code> does not match <i>c</i> in <i>acorn</i>
\$	Matches only at the end of the line	<code>/r\$/</code> matches <i>r</i> in <i>Car</i> <code>/r\$/</code> does not match <i>t</i> in <i>Cat</i>
*	Matches the character preceding it if the character occurs zero or more times	<code>/co*/</code> matches <i>co</i> or <i>c</i> or <i>cooooo</i> <code>/co*/</code> does not match <i>pi</i>
+	Matches the character preceding it if it occurs one or more times	<code>/co+/</code> matches <i>co</i> or <i>cooooo</i> <code>/co+/</code> does not match <i>ca</i>
?	Matches the character preceding it if it occurs zero or one time	<code>/o?l/</code> matches <i>style</i> or <i>column</i> <code>/co?l/</code> does not match <i>cool</i>
.	Matches any individual character, excluding the newline character	<code>/./</code> matches <i>a</i> or <i>@</i> <code>/./</code> does not match <i>\n</i> or <i>l</i>
(x)	By replacing <i>x</i> with characters, matches that sequence and keeps it in memory to be used later; used for grouping of expressions	<code>/(a)/</code> matches <i>a</i> <code>/(cool)/</code> matches <i>cool</i> <code>/(cool)/</code> does not match <i>coal</i>
 	Used as a logical OR symbol to allow a match of what is on the left of the symbol OR what is on its right	<code>/cool bad/</code> matches <i>cool</i> <code>/cool bad/</code> matches <i>bad</i> <code>/cool bad/</code> does not match <i>car</i>
{x}	Using a number to replace <i>x</i> , matches when there are exactly <i>x</i> occurrences of the character preceding it	<code>/n{1}/</code> matches <i>n</i> <code>/nn{2}/</code> matches <i>nnn</i> <code>/nn{1}/</code> does not match <i>nnn</i>
{x,}	Using a number to replace <i>x</i> , matches when there are <i>x</i> or more occurrences of the character preceding it	<code>/n{1,}/</code> matches <i>n</i> <code>/n{1,}/</code> matches <i>nnnnn</i> <code>/n{3,}/</code> does not match <i>nn</i>
{x,y}	Using numbers to replace <i>x</i> and <i>y</i> , matches when there are at least <i>x</i> occurrences of the character preceding it but no more than <i>y</i> occurrences of it	<code>/n{1,2}/</code> matches <i>n</i> <code>/n{1,2}/</code> matches <i>nn</i> <code>/n{2,3}/</code> does not match <i>n</i> <code>/n{4,7}/</code> does not match <i>nnn</i>
[]	Matches a character set of your choice; will match when any one of the characters in the brackets (such as [abc]) or any one of a range of characters (such as [a-k]) is present	<code>/[abc]/</code> matches <i>a</i> <code>/[abc]/</code> matches <i>b</i> <code>/[abc]/</code> matches <i>c</i> <code>/[a-k]/</code> matches <i>j</i> <code>/[a-k]/</code> does not match <i>n</i>
[^]	Matches when the characters in your character set are <i>not</i> present; may be a set (such as [abc]) or a range (such as [a-k]); if it is not within brackets [], then it matches the beginning of the string, as in the first entry in this table.	<code>/[^abc]/</code> matches <i>d</i> <code>/[^abc]/</code> does not match <i>b</i> <code>/[^a-k]/</code> matches <i>n</i> <code>/[^a-k]/</code> does not match <i>j</i>

Regular Expression Codes *(continued)*

Character	Purpose	Example
\	Used to escape special characters or to make a normal character special	<code>\@</code> escapes the <i>@</i> character <code>\n</code> represents a newline character
[b]	Matches a backspace keystroke	<code>/[b]/</code> matches a backspace
\b	Matches when the character before or after it is located at a word boundary, such as before or after a space character; to match the beginning of a word, place the character to the right of the symbol (<code>\bc</code>); to match the end of a word, place the character to the left (<code>c\b</code>)	<code>/\bc/</code> matches <i>c</i> in <i>my car</i> <code>/\bm/</code> matches <i>m</i> in <i>my car</i> <code>/\bc/</code> does not match <i>c</i> in <i>ace</i> <code>/\bm/</code> does not match <i>m</i> in <i>Sam</i> <code>/m\b/</code> matches <i>m</i> in <i>Sam</i> <code>/c\b/</code> matches <i>c</i> in <i>Mac W</i>

		/m\b/ does not match <i>m</i> in <i>emu</i> /c\b/ does not match <i>c</i> in <i>my car</i>
\B	Matches a character that is not located at a word boundary	^Ba/ matches <i>a</i> in <i>car</i> ^Bc/ does not match <i>c</i> in <i>car</i>
\cX	Using a letter character to replace <i>X</i> , matches when the user presses the <code>ctrl</code> key followed by typing the letter <i>X</i>	^cX/ matches <i>ctrl-x</i> ^cV/ matches <i>ctrl-v</i> ^cS/ does not match <i>ctrl-z</i>
\d	Matches if the character is a single numeric character	^d/ matches <i>4</i> ^d/ does not match <i>s</i>
\D	Matches a single character if it is <i>not</i> a numeric character	^D/ matches <i>s</i> ^D/ does not match <i>4</i>
\f	Matches if there is a form feed	^f/ matches a form feed
\n	Matches if there is a new line	^n/ matches a new line
\r	Matches if there is a carriage return	^r/ matches a carriage return
\s	Matches a single character if it represents white space (such as a space or a new line)	^s/ matches the space in <i>b c</i> ^s/ matches the tab in <i>b c</i> ^s/ does not match <i>bc</i>
\S	Matches a single character if it does <i>not</i> represent white space	^S/ matches <i>d</i> ^S/ does not match a blank space
\t	Matches if there is a tab	^t/ matches the tab in <i>b c</i>
\v	Matches if there is a vertical tab	^v/ matches a vertical tab
\w	Matches any single character that is a letter, number, or underscore	^w/ matches <i>4</i> ^w/ does not match <i>@</i>
\W	Matches any single character that is <i>not</i> a letter, number, or underscore	^W/ matches <i>@</i> ^W/ does not match <i>g</i>

Regular Expression Codes Now, if you want to make sure a text field contains one or more digits, you could use the `/d` and `+` characters from with the following HTML and JavaScript code, starting with the HTML code:

```
<body>
<form id="has_digits">
<label for="user_text">Enter some text:</label>
<input type="text" id="user_text">
<input type="submit" value="Test">
</form> </body>
```

Next, the JavaScript code:

```
let has_digits = document.getElementById("has_digits"); has_digits.addEventListener("submit",
function(event) { let has_num = document.getElementById("user_text").value,
    tomatch = /\d+/; event.preventDefault(); if
(tomatch.test(has_num)) {
    alert("Your entry contained one or more numbers!");
} else {
    alert("Your entry did not contain any numbers!");
}
}, false);
```

This code simply checks to see whether any digits are in the string. If you want to ensure that the viewer typed in *only* digits without any other types of characters, you need to be sure the regular expression is written to test from

the beginning to the end of the string. Using the ^ and \$ symbols from Table 13-4, you can ensure that the string is tested for the match starting at the beginning of the string and ending at the end of the string. The + following the \d ensures that there is at least one number entered. Thus, the following patterns would allow only digits:

```
let tomatch = /^d+$/;
```

Since the only valid characters from the beginning to the end of the string are digits, this will return true only for entries containing digits without other characters present in the string. This is especially helpful for validating forms and other types of user input to ensure that it is the type of input you expect.

NOTE

Regular expressions can be quite powerful for validation because they allow less erroneous information to be accepted.

Client-side validation of form submissions with data such as e-mail addresses or phone numbers can save unnecessary trips to the server. However, users may disable JavaScript support and make form submissions directly. Therefore, client-side validation should support server-side validation (by a PHP script or CGI script, for example), but should never replace it (it could cause a great security risk to store or display data from a viewer that has not been validated).

Grouping Expressions

You will notice in Table 13-4 that an expression surrounded by parentheses indicates a group that can be stored for later use in the expression (or using a method such as the match() method where it will store each match of a group along with the overall match in an array).

For example, you might decide to use a particular sequence of numbers and to have that sequence repeat a particular number of times. To match the number of times something is repeated, you can use curly brackets ({}) along with a number or number range. For instance, if you want to match five instances of the number 3, you could use the following expression:

```
/3{5}/
```

If you wanted this to be a match if the number 3 occurs at least five times but no more than ten times, you could use the following expression:

```
/3{5,10}/
```

Now, suppose you wanted the match to start with a 3 and have any digit as the next character, and wanted to match that entire sequence five times (thus, something like 3234353637 would be a match). You might write the following:

```
/3\d{5}/
```

The trouble with this is that it gets the 3 correct, but matches five digits afterward without the need to repeat the 3. Thus, a number like 387643 would match even though you wanted to have five sets of two numbers with each set beginning with a 3. To fix this, you can group the 3 and the `\d` together with parentheses, and follow that with the number of times it should repeat:

```
/(3\d){5}/
```

This time, the 3 and the second digit are grouped together, and that sequence must be repeated five times.

Grouping is a helpful way to get more out of your use of regular expressions, and you will see more of this when you get to form validation in the next chapter.

The `replace()`, `match()`, `matchAll()`, and `search()` Methods

These methods of the String object were mentioned earlier in the chapter, and will make more sense now that regular expressions have been introduced.

The `replace()` Method

To replace information in a string, you can use regular expressions and the `replace()` method of the String object. The syntax for using the `replace()` method is as follows:

```
varname = stringname.replace(regex, newstring);
```

You replace *varname* with the name of the variable that will hold the new string value once the information has been replaced. You replace *stringname* with the name of the string variable that will undergo the replacement. You replace *regex* with the name of the regular expression to be used to match against the string. Finally, you replace *newstring* with the string or string variable to replace any matched values in the string.

As an example, the following code replaces the first instance of “car” in *mystring* with “skunk”:

A string literal is created

```

let mystring = "I like the way a new car smells, and cars are fun.",
    toreplace = /car/, newstring = mystring.replace(toreplace,
"skunk");
    alert(newstring);

```

The pattern to replace is set as a regular expression

The first instance of a match is replaced in the string, and the result is assigned to a variable

The alert shows the updated string

The preceding code replaces only the first instance of car, giving the alert “I like the way a new skunk smells, and cars are fun.” If you want to change every instance of “car” instead, the g flag is helpful at the end of the regular expression, as shown in the following code:

```

let mystring = "I like the way a new car smells, and cars are fun.", toreplace = /car/g,
    newstring = mystring.replace(toreplace, "skunk"); alert(newstring);

```

Adding the g flag causes all matches of the pattern to be replaced when the replace() method is run

The g flag will match every instance of the regular expression it finds in the string. Thus, when the replace() method is run, all instances of “car” will be replaced with “skunk.” The viewer will see this alert: “I like the way a new skunk smells, and skunks are fun.”

By using the replace() method with a regular expression, the first letter of the first and last name can be changed more easily. The following code shows how:

```

function getname() {
    let tomatch = /^[A-Za-z]+\s[A-Za-z]+$/, toreplace = /\b[A-Za-z]/gi,
        thename = prompt("Enter your first and last name", "");
    if (tomatch.test(thename)) {
        newname = thename.replace(toreplace, "Z"); alert("Now your name is " + newname);
    } else { alert("Name invalid. Please Try Again");
        getname();
    }
}
getname();

```

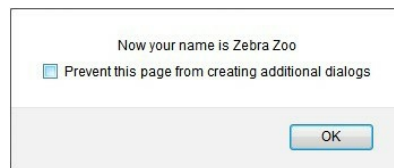
This script changes the first letter of the first and last name to Z regardless of what it was before. The regular expression for the replacement simply looks for a letter at the beginning of a word using the word boundary (\b) code. Each time a letter is at the beginning of a word, it is replaced.

The validation of the input keeps the script from getting more than two names and one space, and it also ensures that at least one letter is in each

name, with no numbers or special characters. The illustration shows the result of the script if the viewer enters Debra Loo at the prompt: the viewer's name is changed.

```
let tomatch = /^[A-Za-z]([A-Za-z]*)\s[A-Za-z]([A-  
Za-z]*)$/,    thename = prompt("Enter your first  
and last name", ""); if (tomatch.test(thename)) {  
    newname = thename.replace(tomatch, "Z$1 Z$2");  
...  

```



The `match()` and `matchAll()` Methods

The `match()` method compares a regular expression and a string to see whether they match. It returns an array containing one or more matches, depending on how it is used. If no match is found, it returns `-1`.

The basic use of the `match()` method is as follows:

```
string.match(regex);
```

The *string* will be your string literal, and *regex* will be your regular expression literal. Note the difference in the order between this method and the `test()` method. You could use it in this way:

```
let mystring = "I am Ironman!",  
    tomatch = /Iron/; if (mystring.match(tomatch)) { alert("Your string  
contains Iron!");  
} else {  
    alert("Sorry, no Iron in your string."); }
```

If this is used with the `g` flag or with grouping using `()`, it will remember each match made (including matches on groups or nested groups) and return each match as an array element. The `matchAll()` method works in this way, returning all of the matches found as an array.

The `search()` Method

The `search()` method executes the search for a match between a regular expression and a specified string. If a match is found, it returns the position in the string where the beginning of the match was found.

Otherwise, it returns `-1`. Here is an example of this method in action:

```
let mystring = "I am Ironman!",  
    tomatch = /Iron/; if (mystring.search(tomatch)) {
```

```
    alert("Iron found at position " + mystring.search(tomatch) + "!");  
  } else {  
    alert("Sorry, no Iron in your string."); }  
}
```

As you can see, the syntax is much like that of the `match()` method.

More Information

For more information on regular expressions and how to create more complex patterns, you can look at the following online resources:

- www.regular-expressions.info/
- www.regular-expressions.info/javascript.html (includes specifics on the JavaScript engine)
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

With these techniques down, you are ready to move on to working with forms in JavaScript and working on the validation of form contents.

Continuing Project

In this chapter, you will update your Node.js project to begin creating the front-end HTML and CSS for displaying your bowler data in your app. As you move through subsequent chapters, the skeleton HTML will be filled in with the actual data.

First, create a folder within your project folder named `public`. This folder will be used to store CSS and front-end JavaScript files. You may recall seeing the following line of code in your `server.js` file:

```
app.use(express.static(path.join(__dirname, 'public')));
```

This tells Express to look for static assets (such as CSS and JS files) in a folder named `public` within your project folder. You can create folders within that folder, but since this will be a relatively simple app, you will just place your assets directly within the `public` folder.

Next, create a CSS file with the following code and save it as `styles.css` inside the `public` folder:

```
body {  
  margin: 2% 5% 0 5%; }
```

You will include this CSS inside your `index.html` file.

With this in place, update your `index.html` file in your project folder to

have the following code:

```
<html>
  <head>
    <title>Bowler Stats</title>
    <link rel="stylesheet" type="text/css" href="/styles.css">
  </head>
  <body>
    <h1>Bowler Stats</h1>
    <table id="bowlers-table" border="1" cellspacing="0">
      <thead>
        <tr>
          <th>Name</th>
          <th>Accuracy</th>
          <th>Power</th>
          <th>Consistency</th>
          <th>Overall</th>
        </tr>
        <tbody id="bowlers-list">
          <!-- Fill in info here -->
        </tbody>
      </table>
    </body>
  </html>
```

Save the file. If you want to see the shell HTML page this creates, make sure to start your Node server and then go to <http://localhost:3000> to see the basic HTML that will need to be filled in once you pull the information you retrieved from the database into the front-end code. As you can see here, the plan is to fill in the tbody portion of the table with rows for each bowler's information. This will be done in Chapter 15 once you learn about AJAX.

[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
[11]