

JavaScript

THE ULTIMATE GUIDE TO UNDERSTAND JAVASCRIPT CODE AND ITS FUNDAMENTALS. DISCOVER LITERAL AND CONTROL FLOW. LEARN VARIABLES, FUNCTIONS, OBJECT AND THE BEST JQUERY

MARK GRAPH

JavaScript

*The Ultimate Guide to Understand
JavaScript Code and its Fundamentals.*

*Discover Literal and Control Flow.
Learn Variables, Functions, Object and
the Best jQuery.*

Mark Graph

© Copyright 2020 - All rights reserved.

The content contained within this book may not be reproduced, duplicated or transmitted without direct written permission from the author or the publisher.

Under no circumstances will any blame or legal responsibility be held against the publisher, or author, for any damages, reparation, or monetary loss due to the information contained within this book, either directly or indirectly.

Legal Notice:

This book is copyright protected. It is only for personal use. You cannot amend, distribute, sell, use, quote or paraphrase any part, or the content within this book, without the consent of the author or publisher.

Disclaimer Notice:

Please note the information contained within this document is for educational and entertainment purposes only. All effort has been executed to present accurate, up to date, reliable, complete information. No warranties of any kind are declared or implied. Readers acknowledge that the author is not engaging in the rendering of legal, financial, medical or professional advice. The content within this book has been derived from various sources. Please consult a licensed professional before attempting any techniques outlined in this book.

By reading this document, the reader agrees that under no circumstances is the author responsible for any losses, direct or indirect, that are incurred as a result of the use of information contained within this document, including, but not limited to, errors, omissions, or inaccuracies.

Introduction

Chapter 1 Fundamental JavaScript Concepts

Chapter 2 HTML Overview

Chapter 3 JavaScript's Control Flow Statements

Chapter 4 The Different Types of Loops in JavaScript

Chapter 5 Syntax

Chapter 6 Enabling JavaScript in Browsers

Chapter 7 Placement of JavaScript in Files

Chapter 8 Popup Message

Chapter 9 JavaScript Variables

Chapter 10 JavaScript ECMAScript Standard

Chapter 11 Working With JavaScript: A Brief HTML Guide for Beginners

Chapter 12 Changing the content of HTML elements using DOM

Chapter 13 Changing CSS using DOM

Chapter 14 Pointers

Chapter 15 Expressions and Operators

Chapter 16 What Are Some Of The JavaScript Variables?

Chapter 17 Variables, data types & constants

Chapter 18 Closures and Callbacks in JavaScript

Chapter 19 Apply, call, and bind methods in JavaScript

Chapter 20 Events

Chapter 21 Arrays in JavaScript

Chapter 22 Values, Types, and Operators

Chapter 23 Definition of Arrays in JavaScript

Conclusion

Introduction

HTML is not very smart. It mostly lets people look at text and images and allows them to move to other pages where they will do more of the same. What adds the intelligence to a web page is JavaScript. It makes the website more engaging, effective, and useful by letting pages respond to our visitors when they interact with the content.

This book assumes that we already know how to use HTML to specify web page structure and content. It will be additionally beneficial if we are familiar how pages are styled with CSS, separate from the web page structure. If this is the case then we are ready to add a little behavior to the page and make it more dynamic and interactive with JavaScript. Otherwise, without HTML and CSS, JavaScript will not do us much good. They are viewed as the three fundamental pillars of the web page: structure, presentation and behavior.

What is JavaScript?

JavaScript is the scripting language of the web with the sole purpose of adding interactivity to our pages. In addition to interactivity, modern versions of JavaScript can also be used to load and parse information from external sources or even the website's users. JavaScript is essentially a piece of programming code embedded in the HTML structure of a web page. When the web browser reads this code it activates a built-in interpreter that understands how to decipher this language and process its commands.

Although programming is involved during coding, JavaScript is not a programming language. In conventional web programming languages, like Java or .NET, the code has to be compiled before it is executed. Compiling means that the code has to be first sent to a special program that is run on the server. This program, also known as application server software, translates the code, creates the requested page and/or functionality and serves this back as HTML. Scripting languages, like JavaScript, are not compiled, but rather are interpreted on-the-fly. This means that no special software is involved as the user's own browser runs and executes the code as it is encountered.

Note: JavaScript was created during a time when Java was a very popular language. Other than that, the languages are not related and have almost nothing in common except for basic programming logic.

Implementing JavaScript

Now that we have a general idea as to what JavaScript is, we can start working with this language. As JavaScript code is part of the HTML document, we need to know how to tell browsers to run our scripts. There are two common options available to us when we want to include JavaScript in a web document and in both cases we will use the `<script>` element. The `<script>` tag is used when we want to tell the browser where the JavaScript code begins and where it ends within an HTML document. As such, this tag can be included either in the head or the body section of the page.

The first option is to place the code inline within the document structure. To do this we will begin by opening a `<script>` tag, entering the JavaScript code, and then finish by closing with the `</script>` tag. We can theoretically leave the document like this as almost all browsers will assume that the scripting language between the `<script>` tags is JavaScript by default. Nevertheless, for maximum compatibility we will extend this tag with the `type` attribute and the `text/javascript` value in order to instruct the browser how to exactly interpret the code.

```
<script type="text/javascript">  
    //A JavaScript comment  
</script>
```

The second option is to load the JavaScript code from an external file into our HTML document. For this purpose we can use the `<script>` element again, but this time in addition to the `type` attribute we will also include the URL to the external file in the `src` attribute of the `<script>` element. The external file must be a text-only file with the `.js` file extension that contains only pure JavaScript code without any HTML elements or CSS rules. For example, to call the external `scripts.js` file into our browser we would use the following code:

```
<script src="script.js" type="text/javascript">  
</script>
```

We put JavaScript in an external file and include it in the web page when we like to share the functionalities across our entire web site. Otherwise, if we just need to add some local interactive behavior, we embed the code within the page.

Note: Script files are loaded in the order in which they are placed in the HTML code

Chapter 1 Fundamental JavaScript Concepts

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".

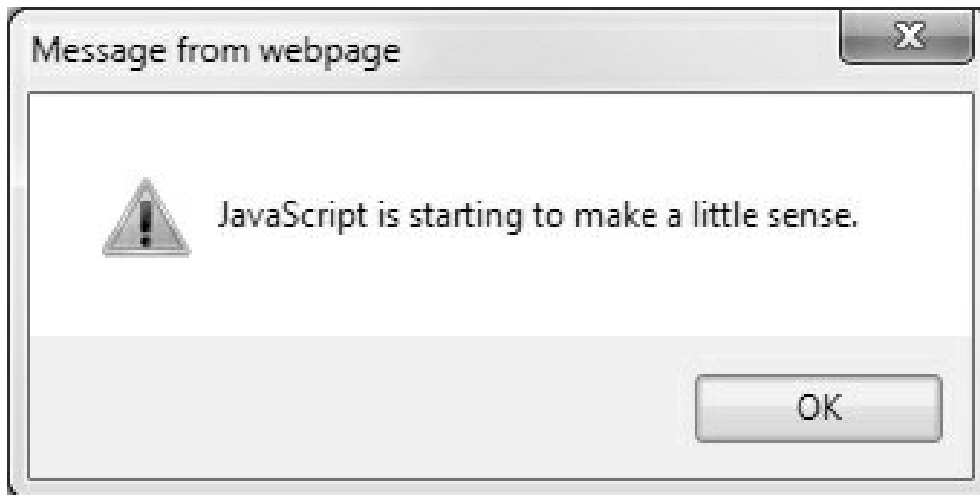


Image 1. Alert window

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>");
```

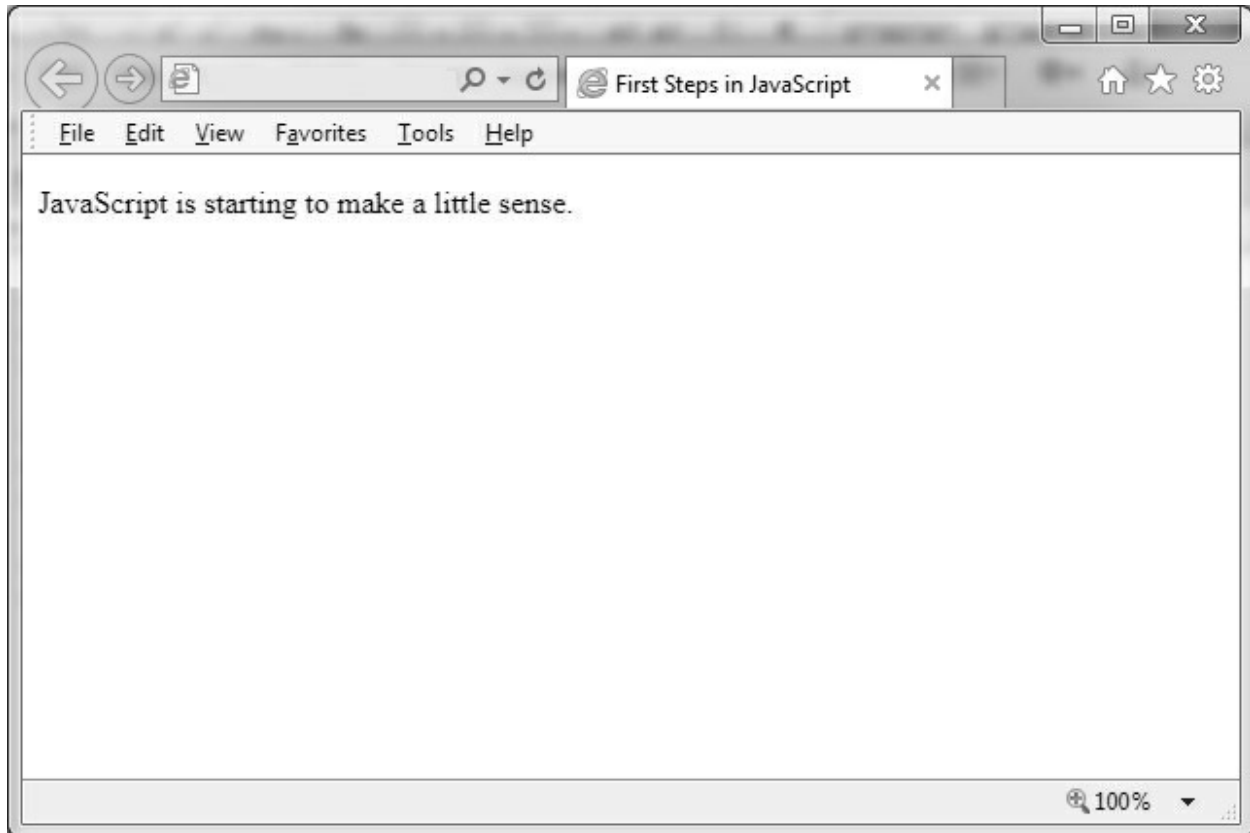


Image 2. Example of a `document.write` statement

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` in 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);
```

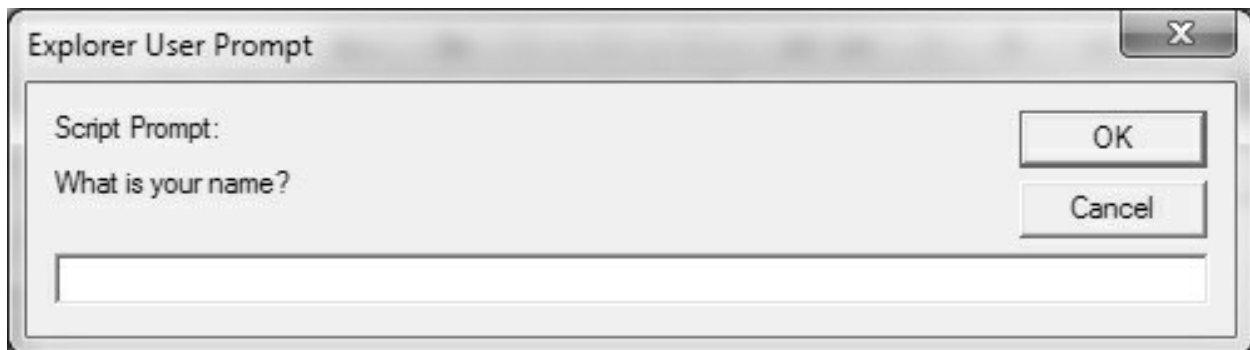


Image 3. A custom prompt dialog box

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);
```

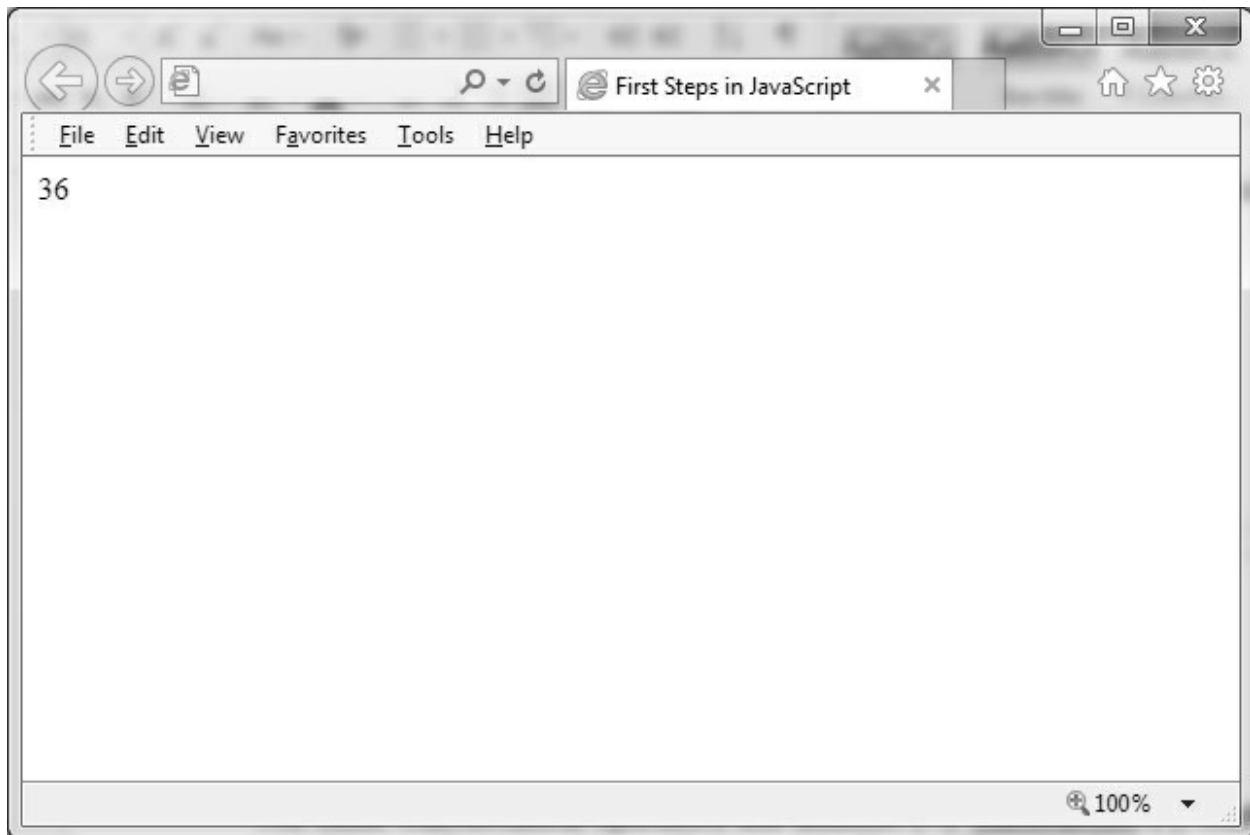


Image 4. Using variables to calculate age

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);
```

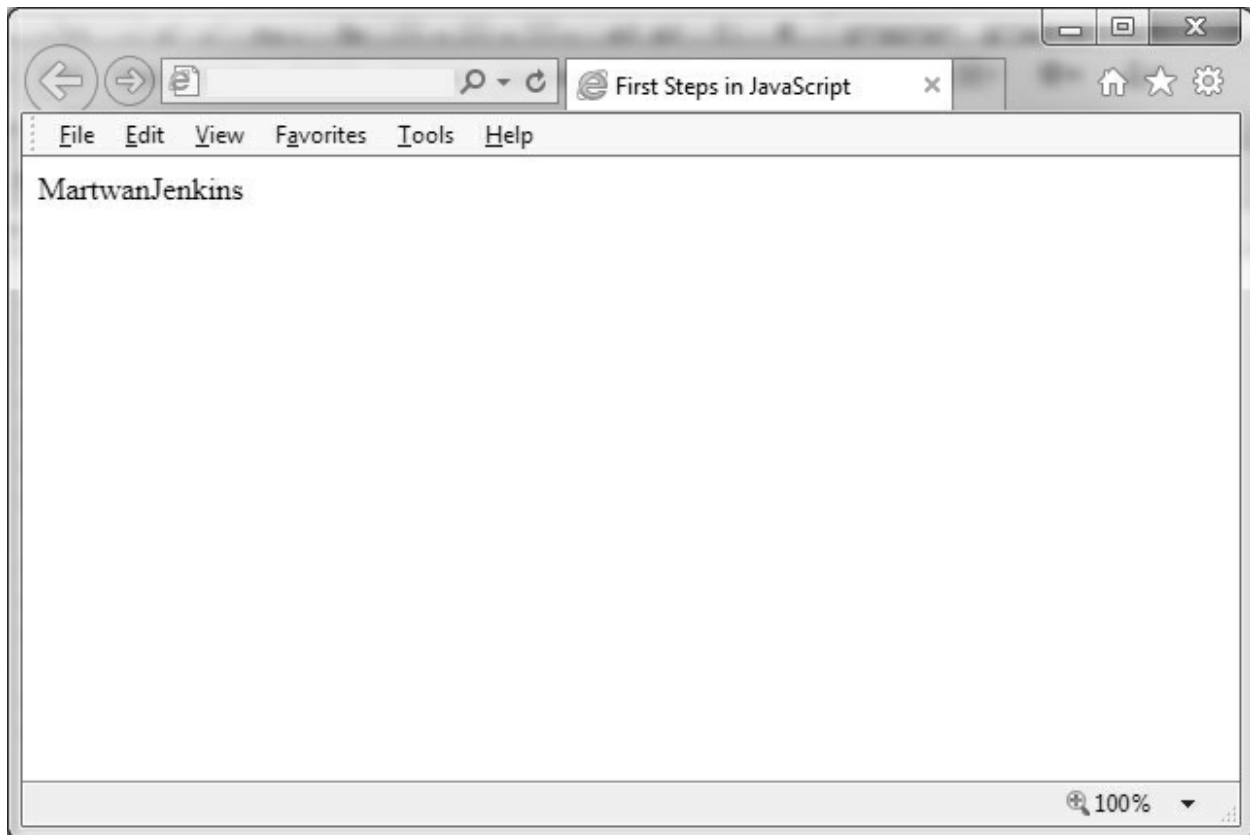



Image 5. Concatenating strings into a full name

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;
```

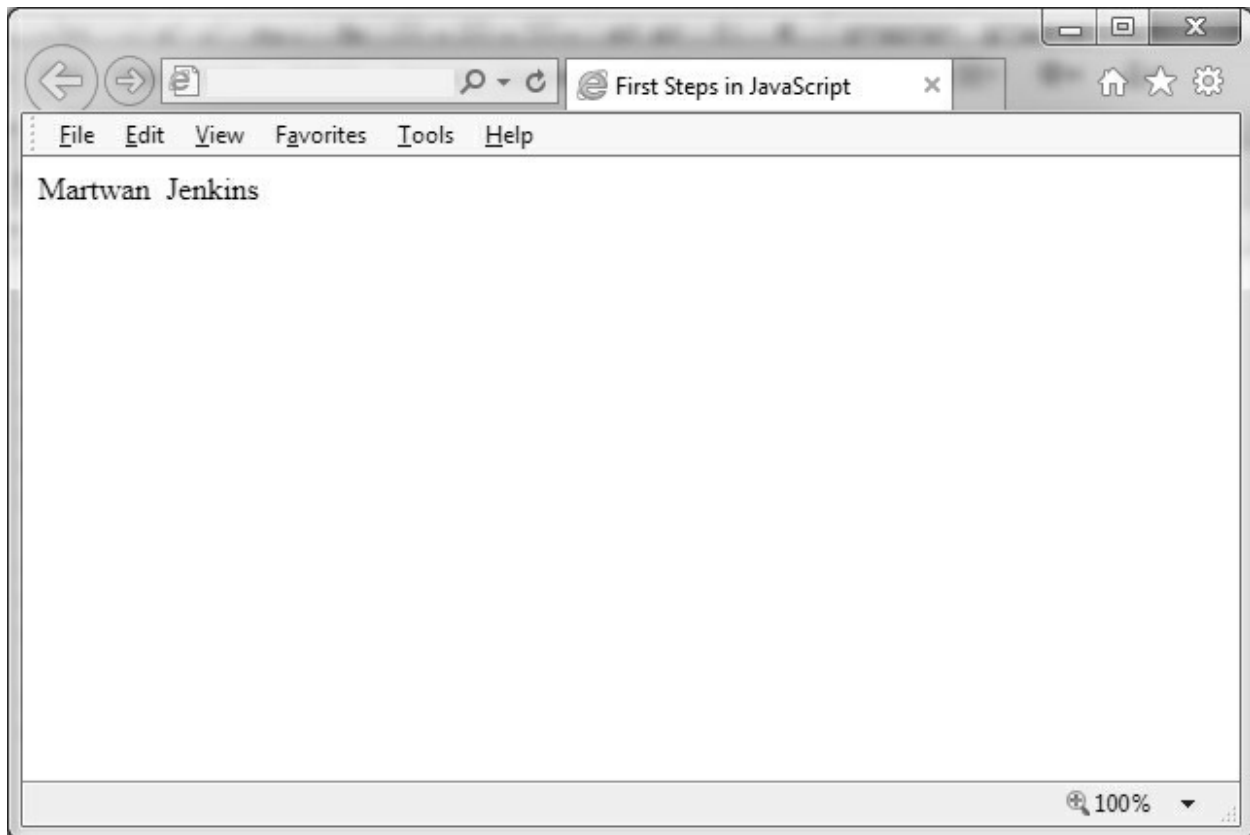


Image 6. Concatenating strings with spaces

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);
```

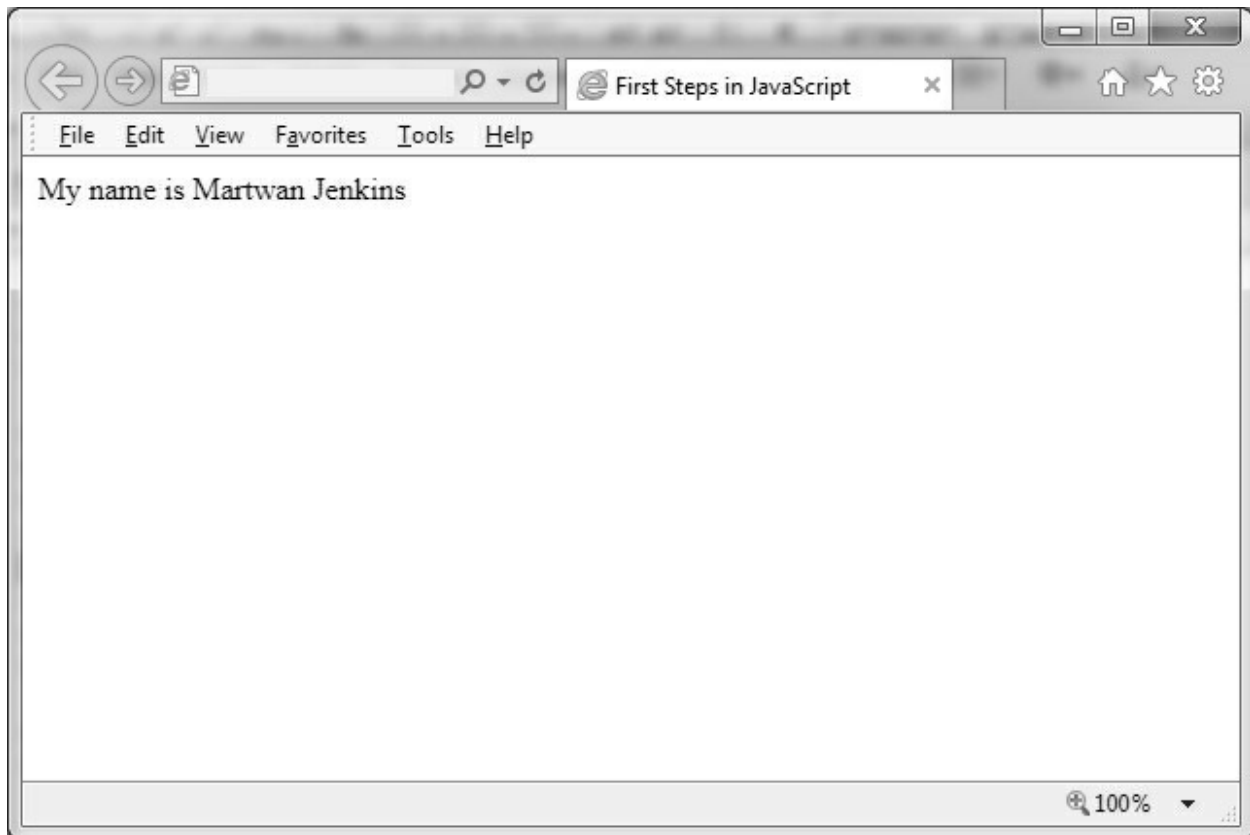


Image 7. Concatenating a full sentence

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>");
```

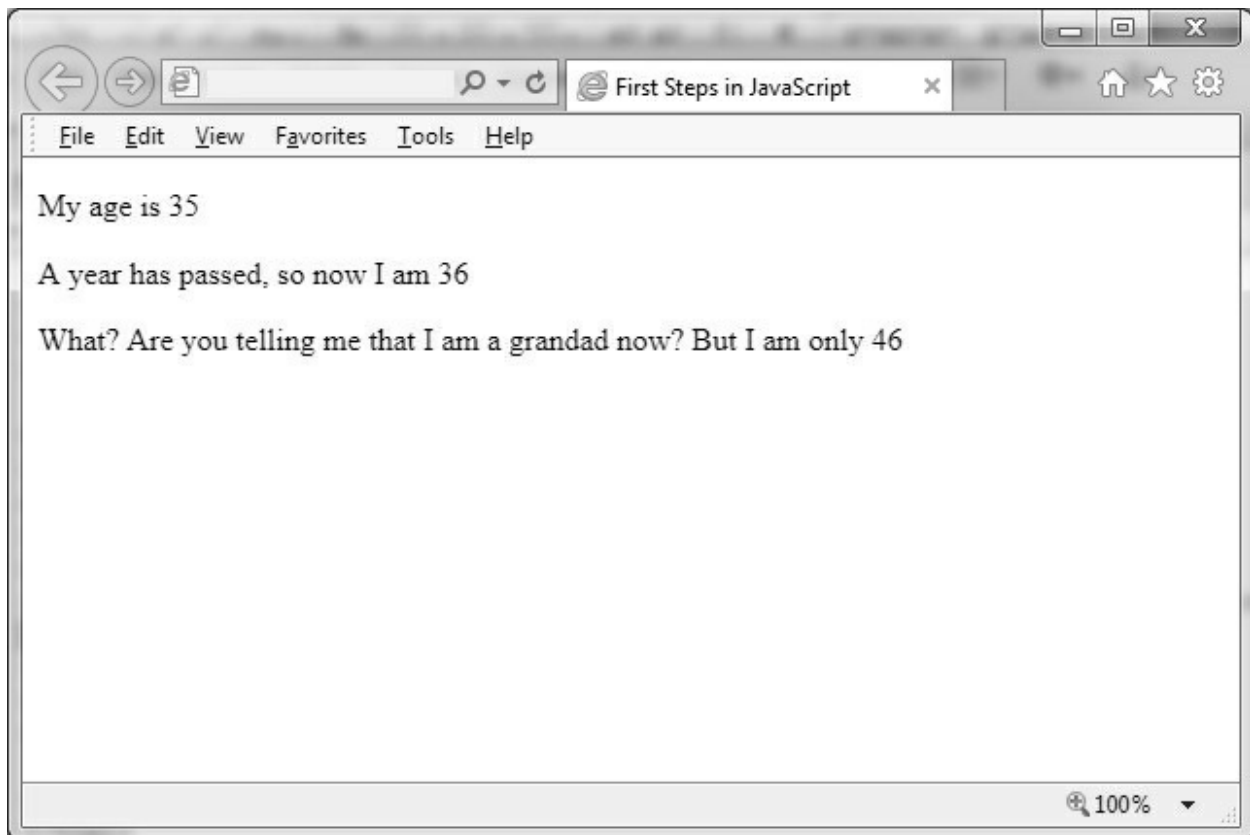


Image 8. Changing variables with assignment operators

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 asage = age - 5;

age*= 5;is the same asage = age * 5;

age/=5;is the same asage = 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 asage = age + 1;

age--;is the same asage = 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

Chapter 2 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

Chapter 3 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:

```
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.

Chapter 4 The Different 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 time-consuming. 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:

Chapter 5 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">
```

```
<!--
```

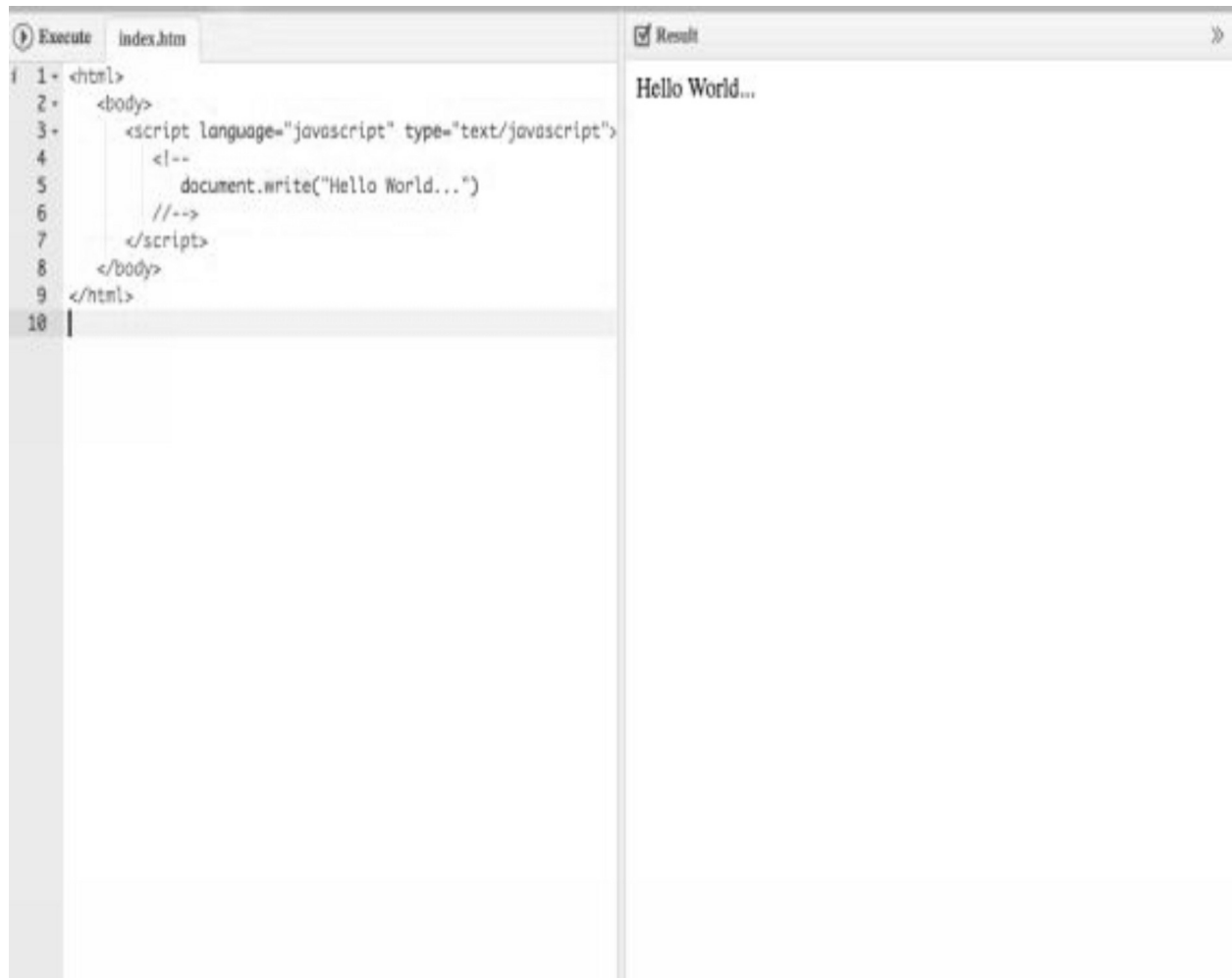


```
document.write("Hello World..")
```

```
//-->
```

```
</script>
```

```
</body>
```



```
Execute index.htm Result
1- <html>
2- <body>
3- <script language="javascript" type="text/javascript">
4- <!--
5-     document.write("Hello World...")
6-     //-->
7- </script>
8- </body>
9- </html>
10- |
```

Hello World...

```
</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>
```

Chapter 6 Enabling JavaScript in 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 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 non-JavaScript 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.

Chapter 7 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!")
```

Chapter 8 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.

Chapter 9 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:



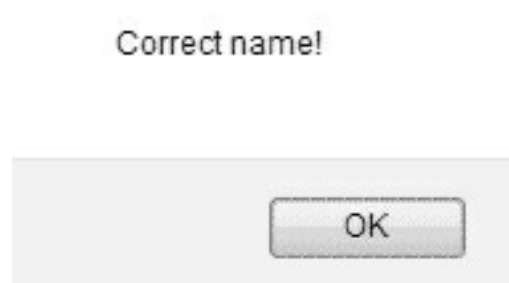
This is the values of the two values are equal, that is, 10. If not, nothing will

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.

Chapter 10 JavaScript ECMAScript Standard

ECMAScript is a conventional specification for scripting languages regulated by the ECMA International. It is core to several scripting languages such as JavaScript, ActionScript, and Jscript.

In this guide, we are not going to look at the history of the ES standardization. What we are going to do is use the ECMAScript to learn a bit more about the evolution of JavaScript.

You can however learn more about the ECMAScript standard by visiting the following resource page:

<https://bit.ly/2DZiHQb>

A Brief History of JavaScript

JavaScript is the creation of Brendan Eich, who at the time of creating the scripting language, was an employee at Netscape Communications Corporation, a corporation that Wikipedia notes had hired him to help embed their “schema programming language into the Netscape Navigator, one of the most dominant web browser of the 1990s.”

Wikipedia also notes that before Eich could start his work at Netscape in 1995, the company collaborated with Sun Microsystems with the former agreeing to use Sun’s static programming language, Java, in their Netscape Navigator.

JavaScript developed out of the collaboration mentioned above because once Netscape Communications and Sun’s Microsystem collaborated, Netscape Communications immediately realized that they needed to create a new scripting language whose syntax was similar to that of Java so that the two languages could complement each other well and allow Netscape to create web technologies platforms that could effectively compete with Microsoft, one of the giant computing companies of the 1990s.

Netscape tasked Eich with the task of creating a prototype of this scripting language, which is how, as Wikipedia notes, Eich ended up creating JavaScript in 10 days of May 1995.

During its developmental stages in the 1990s, JavaScript went by the name Mocha, and during its beta development stages in Netscape Navigator 2.0, it was initially called LiveScript a name later changed to JavaScript in December 1995 with the intent behind the name not being to cause confusion

or to mean that the language was similar to Java, but to capitalize on the popularity of Java—and Java was the most popular programming language of the time.

JavaScript follows the script version released and maintained by the ECMA Script. In this book, we are going to focus on illustration based on the latest version of JavaScript i.e. ECMA 2019 which has significant improvements over the previous versions.

Environment Setup

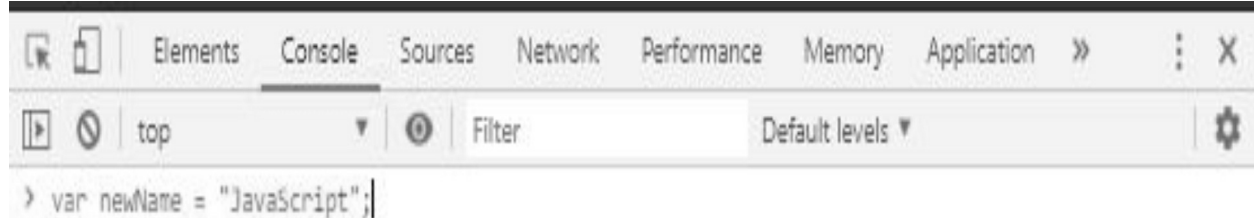
In this section, we are going to look at some of the tools we are going to use to write programs in JavaScript. JavaScript does not offer complex working environment; to work with it effectively, all you truly need is a text editor and a browser.

For the tutorials in this JavaScript guidebook, we are going to use the Latest version of Google chrome as our web browser of choice and the Brackets Text Editor as our text editor of choice for the various illustrations you are going to find in this guide. You can download these two pieces of software from the following resource pages:

<https://www.google.com/chrome/>

<http://brackets.io/>

You can use any text editor or browser you feel comfortable working with. To enable the JavaScript console in your browser, Right click inside your browser and select “developer options” then navigate to



console.

NOTE: To learn how to use JavaScript to create dynamic and responsive web pages and applications, you need to have some knowledge of how the web works, as well as how to work with HTML and CSS.

Chapter 11 Working With JavaScript: A Brief HTML Guide for Beginners

In this chapter, we are going to look at some of the basic HTML elements you need to understand in order to learn JavaScript as well as comprehend the contents as well as the various hands-on tutorials in this JavaScript book.

NOTE: This is not a HTML tutorial. It is simply a basic introduction to working with HTML because to become learn JavaScript and to become efficient at using JavaScript to create web applications and pages, you need to having some basic knowledge of how to work with HTML.

HTML, which stands for Hyper Text Markup Language, is a markup language that we normally use to define the structure of a web document such as headings, sub-headings, paragraphs and such. Since it developed at the advent of the World Wide Web, HTML is old and has advanced and changed significantly over the years. Today, we normally use HTML to format web documents for the purpose of sharing information.

To equip you with the basic HTML knowledge you need to have to start working with JavaScript effectively, we are going to look at several key aspects of a HTML document:

HTML Basic Structure

All HTML documents have a standard structure, also called the basic boilerplate. The following tags are in most documents that appear as

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
  </head>
  <body>
  </body>
</html>
```

html.

Let us explore the document above.

- The `<!DOCTYPE html>` in html is not a primitive HTML tag (discussed later). We use it to alert the browser of the version the HTML document is using. The syntax for the doctype for HTML 5 and HTML 4 is different. `<!DOCTYPE html>` is essential and should always be at the start of any html document.
- The `html` tag – This is the root tag for the entire html document. All other tags within the html document must be within the `html` tag.
- The `<head>` – In an html document, we use the `head` tag as a container for the metadata to be found on the web page. We place this tag after the `html` tag and before the `body` tag.
- The `body` tag – We use this tag to represent the beginning of an html document body. Not all tags go here, but almost every other tag such as images, text, and other media in the document are within these tags.
- The `title` tag – We use this tag to set the title of the HTML document. We place this title within the opening and closing title tag as `<title>Title page</title>`

NOTE: Most html tags require ‘closing.’ HTML5 and some browser may recognize unclosed tag and fix it. However, getting into the habit of always closing tags that require closing is a good way to improve your programming skills.

The Script Tag

Now that we have introduced the concept of tags, let us look at the most important HTML tag we are going to use in the book, the script tag.

We normally use the script tag `<script></script>` to include client-side scripts such as JavaScript code. The opening and closing of the script tag may contain a reference to an external JavaScript file or can include native JavaScript code within it. We can classify this tag as a document metadata.

The script tag also accepts other attributes. Let us look at the `src` attribute that allows you to specify the location of the external script file. For example, fire up your text editor and type in the following code. Remember that for the tutorials in this guide, we shall be using

```
<!DOCTYPE html>
<html>
  <script src="/Files/index.js"></script>
  <head>
    <title></title>
  </head>
  <body>
  </body>
</html>
```

Brackets:

This script tells the browser to locate the file in the Files directory, under the name index.js file. A script may have many other attributes; we have merely used this to illustrate the most important HTML tag whose knowledge of use you need to have before you can create JavaScript webpages and applications.

NOTE: As mentioned earlier, you cannot use JavaScript without some knowledge of HTML and CSS.

Working with JavaScript: A Brief CSS Guide for Beginners

Cascading Style Sheet or CSS is a design language used to present and modify the appearance of web pages. It allows web developers to change page features such as the fonts, image presentation and locations of various html elements.

Like HTML, JavaScript, PHP, Python, and other programming languages, CSS evolves with time with new features added every time and with each evolution. This book focuses on the latest version of CSS i.e. CSS3.

CSS Overlay

CSS files are linked externally on the html page. However, you can include the CSS code within the HTML (inline CSS) but we do not recommend that. CSS code has three main parts.

It comprises of a set of rules that mainly focus on the styling part, which are then interpreted by the browser, and then applied to their respective elements.

CSS's three main parts are:

- A selector – The CSS selector is a HTML tag that the styling is applied to. It could be a paragraph (using <p> tag).
- Property – A property is an attribute of an HTML tag. The properties include: colors, borders, navigations, etc.
- Value – values are defined values assigned to variables. The border could be 2px wide. The 2px is the value assigned to the border property.

A typical CSS syntax is:

Selector: {

Property: value;

Property2: value;

```
h1 {  
    color: aquamarine;  
    border-bottom: 2px;  
}  
  
p {  
    font-family: "Times New Roman", Times, serif;  
}  
}
```

From the above example, the h1 and p, which represent the heading one and paragraph respectively, represent the selector. Next, once we select the element we want to manipulate, we can set properties and values such as color and set it to blue, red, or aquamarine. In colors, you can set it in RGB or HEX code.

They are many types of CSS selectors and selection methods. For example, you can use the * (asterisk) as a global selector. You can also use the element id to select a specific element in case there are multiple.

NOTE: This is not a CSS book and CSS is therefore beyond the scope of this book. What this chapter sought to do is give you a basic understanding of the

most important CSS elements you need to know how to use in order to start learning how to create JavaScript code, web pages, and web applications too.

Now that you have a basic understanding of how to work with HTML and CSS, we can move on to learning how to work with JavaScript as you pursue mastery of this important scripting language especially if your intention is to learn JavaScript so that you can create dynamic webpages and responsive web applications and websites.

Chapter 12 Changing the content of HTML elements using DOM

We need to change the content of the HTML elements very often for making a dynamic web application or website. Some example scenarios include:

- Giving user feedback. Replacing an HTML element's content instead of using the 'alert' function is a better alternative to give feedback, as you can modify how an element looks and thus make your feedback display better visually. This will, in turn, lead to a more pleasurable user experience.
- Have you seen a live clock, or countdown, on a web application? They both replace their HTML element's content when the unit of time they function on passes by.
- Changing a page's content on a button click without refreshing.

The '.innerHTML' property

This is perhaps the easiest and the most frequently used way of changing the content of an HTML element. This property of an element's object in the DOM represents the current content of that element, and can be used to get or set the content of that element. If a value is assigned to this property of an element's object, then that element's content changes to the newly assigned value.

Syntax:

```
elementObject.innerHTML = 'new value';// setting
```

```
var x = elementObject.innerHTML;// getting
```

Note that this property represents everything within the element's tag, even it's children and sub-children.

Example demonstrating getting an element's content:

```
<body>
```

```
<div id='outer'>
```

```
<div id='inner'>
```


Hey there

```
</div>
```

```
</div>
```

```
<script>
```

```
var innerContent = document.getElementById('inner').innerHTML;
```

```
    var outerContent = document.getElementById('outer').innerHTML;
```

```
console.log( innerContent );
```

```
console.log( outerContent );
```

```
</script>
```

```
</body>
```

The above code produces the following output on the console screen:

Hey there

```
<div id="inner">
```

Hey there

```
</div>
```

As you can see, the output contains the white spaces and the tabs.

TIP: If you wish to get rid of the extra white spaces and tab spaces at the start and the end of the content you can use the `.trim()` method from the string class on them!

Example demonstrating the changing of an element's content:

```
<body>
```

```
<div id='outer'>
```

```
<div id='inner'>
```

Hey there

```
</div>
```

```
</div>
```

```
<script>
    var innerElement = document.getElementById('inner');
    innerElement.innerHTML = 'Hola amigos';

// the content of <div> with id 'inner' has been changed to 'Hola amigos'
</script>
</body>
```

The `textContent` property

This property is almost same as the `innerHTML` property. The difference between this property and `innerHTML` is that this property is only used to get or set the textual content (meaning the children and sub-children element's tag name is not included when getting the content of the element), whereas in the case of the `innerHTML` property, the full content of the element including the children and sub-children element's tag names and attribute were also returned.

Let us understand this by using the same example as we used in the case of the `innerHTML` property – but instead, we will use `textContent` property in place of the `innerHTML` property here.

```
<body>

<div id='outer'>

<div id='inner'>

Hey there
</div>

</div>

<script>

var innerContent = document.getElementById('inner').textContent;
    var outerContent = document.getElementById('outer').textContent;
```

```
console.log( innerContent );
```

```
console.log( outerContent );
```

```
</script>
```

```
</body>
```

The above code produces the following output on the console screen:

```
Hey there
```

```
Hey there
```

The white spaces and the tab spaces at the start and at the ending are included in the output of this result as well, but note that the `<div>` tag is not!

Another difference between the two properties is that the content that we get using the `.innerHTML` property has the characters `'&'`, `'<'`, or `'>'` replaced by `'&'`, `'<'` and `'>'` respectively, whereas the in the content obtained using `.textContent` property these characters remain as they are.

The setting action of both properties will work the same. Here is an example:

```
<body>
```

```
<div id='outer'>
```

```
<div id='inner'>
```

```
Hey there
```

```
</div>
```

```
</div>
```

```
<script>
```

```
    var innerElement = document.getElementById('inner');
```

```
    innerElement.textContent = 'Hola amigos';
```

```
// the content of <div> with id 'inner' has been changed to 'Hola amigos'
```

```
</script>
```

```
</body>
```

Yet another major difference between the two properties is that when you set an element's content using `.innerHTML` property, it is interpreted as HTML code whereas in the case of `.textContent`, it is interpreted as raw text.

Example:

```
<body>
```

```
<div id='outer'>
```

```
<div id='inner'>
```

```
Hey there
```

```
</div>
```

```
</div>
```

```
<script>
```

```
    var innerElement = document.getElementById('inner');
```

```
// no alert box will pop up in this case:
```

```
    innerElement.textContent = "<script>alert('hey')</script>";
```

```
// A alert box will pop up in this case:
```

```
    innerElement.innerHTML = "<script>alert('hey')</script>";
```

```
// this was because, in the latter, the text was interpreted as HTML code
```

```
</script>
```

```
</body>
```

Therefore, it is safer to use the `.textContent` property to set the content of an element instead of the `.innerHTML` property when you are expecting the new content to be plain text only.

Chapter 13 Changing CSS using DOM

CSS is used to style our web pages. Almost everything you see on modern sites has been styled using CSS. Thus, it becomes important for us to learn how to change CSS, which can help us to dynamically change the styling of our web page. Doing so will make our web page more interactive, giving the user a better experience. An example scenario – when a user is writing in a text box, you can make textbox background red to indicate that input is invalid, and green to indicate that it is valid. Have you ever noticed that while registering on some sites, when filling in the password, the password input box border color changes from red to orange and then to green to indicate the strength of password? This is done with the help of JavaScript.

The `element.style` property

We don't directly assign a string containing CSS style definition to this property. Instead, we assign values to “sub-properties” of this property. Technically, this property is an object itself. You can access a property by using the convention “`element.style.propertyName.`” The naming convention of these “sub-properties” is almost the same as in CSS. For example: to change text color, we use the property “color” in CSS. To access or modify it in DOM, we can use `element.style.color`. Also, note that the naming convention of these CSS “sub-properties” in DOM is in camelcase not in kebab-case as in CSS, so the property ‘border-bottom’ becomes ‘borderBottom.’

The values assigned to these “sub-properties” should be in the same format as in CSS.

Discussed below are some of the important CSS properties and how you can get and set them using DOM.

Changing an element's background

You can get or set the background of an element using the ‘.background.’ sub-property of the ‘.style.’ property of an element.

Syntax:

```
var getVal = element.style.background// GET
```

```
element.style.background = set;// SET
```

Example usage:

```
<body>
<p style='background:red' id='changeBg'>
This paragraph background color was originally red but will change to blue
by the JavaScript code!
</p>
<script>

var pElement = document.getElementById('changeBg');

console.log( pElement.style.background ); // prints 'red' on the console
screen

// now let's change its background to blue

pElement.style.background = 'blue';

console.log( pElement.style.background ); // prints 'blue' on the console
screen
</script>
</body>
```

You can directly assign values to the .background sub-property in shorthand format too, as you do in CSS. Example:

```
pElement.style.background = "#00ff00 url('image.png') no-repeat fixed
center";
```

NOTE: If a color is specified in HEX format(#RRGGBB) in any of the CSS property, then while accessing it from DOM it will appear in 'rgb(rrr, ggg, bbb)' format.

Example:

```
var pElement = document.getElementById('changeBg');

pElement.style.background = '#0000ff'; // HEX representation of blue color
```

```
console.log( pElement.style.background ); // prints 'rgb(0, 0, 255)' on the console screen
```

Changing an element's text color

You can get or set the text color of an element using the `‘.color’` sub-property of the `‘.style’` property of an element.

Syntax:

```
var getVal = element.style.color// GET
```

```
element.style.color = set;// SET
```

Changing an element's text size

You can get or set the text size of an element using the `‘.fontSize’` sub-property(note the camel case here) of the `‘.style’` property of an element.

Syntax:

```
var getVal = element.style.fontSize// GET
```

```
element.style.fontSize = set;// SET
```

NOTE: When you assign a size to any size-related property like font size, height width, etc., the sizes should be assigned along with their units (px, em, vh, etc). Not doing so will result in an error.

Example:

```
element.style.fontSize = 40;// WRONG
```

```
element.style.fontSize = '40'; // WRONG
```

```
element.style.fontSize = '40px';// RIGHT
```

The `‘float’` property, an exception in property naming convention in DOM

The `‘float’` property in CSS is used to specify the position of an element in the layout. But `‘float’` is also a reserved word in JavaScript. Owing to this, we cannot use `‘float’` as a property name in JavaScript. We use the name `‘cssFloat’` in JavaScript to refer to the `‘float’` property.

Syntax:

```
var getVal = element.style.cssFloat// GET
```

```
element.style.cssFloat = set;// SET
```

Modifying CSS style using element.setAttribute() method

The .setAttribute() method is used to change attributes of elements. We can change the CSS styling of an element by modifying their 'style' attribute. This attribute's value is plain CSS.

Syntax:

```
element.setAttribute('style', 'PLAIN CSS HERE')
```

Example:

```
element.setAttribute('style', 'background:red; height:15px');
```

Modifying CSS style by adding a CSS class to it

You can also modify CSS styling of an element through JavaScript by adding a predefined CSS class to it. You can add classes to an element by adding them to 'className' property of the element.

Syntax for adding a CSS new class:

```
element.className += ' newClassName'
```

Note that there must be a space present in the string before the CSS class name as this property is merely a string containing all the name of all the CSS classes of the element separated by a space.

Example:

```
<body>
```

```
<p style='background:red' id='changeClass'>
```

This paragraph's styling will be changed by JavaScript by adding a CSS class to it!

```
</p>
```

```
<style>
```

```
# our custom CSS class:
```

```
.myClass {
```

```
background:blue;
```

```
height:10px;
```

```
}
```

```
</style>
```



```
<script>  
  
var pElement = document.getElementById('changeClass');  
  
pElement.className += ' myClass';  
</script>  
</body>
```

You may use any way you like to change the CSS styling, but doing it via CSS class names is considered a better practice. This helps by making a more structured codebase similar to stateful architecture, where each class conceptually represents a state. Consider the scenario of weak and strong passwords as described in the beginning of this chapter. When the password has weak strength, you can add a class named 'weakPassword' to the password textbox and a class named 'mildPassword' when it is of mild strength. This helps us to write more organized code.

Chapter 14 Pointers

Pointers make certain complex tasks easier and they are fairly simple to use. A pointer is defined as a variable that holds the address of another variable. Much like a variable, before its use it must be defined.

```
type *var-name;
```

When using a pointer, the type is whatever the type of the base type is. For example, if my pointer is pointing to a double variable, my pointer must be a double. The var-name is the name of your pointer. The asterisk must be used in order to denote a pointer.

```
int *int_point; // pointer to an integer
double *dou_point; // pointer to a double
float *flo_point; // pointer to a float
char *ch_point; // pointer to character
```

The following example shows the information which is stored in a pointer, run the following code to get a better understanding of pointers.

```
#include <iostream>
using namespace std;
int main () {
    int variable = 10; // actual variable declaration.
    int *ipointer; // pointer variable
    cout << "Value of variable: ";
    cout << variable << endl;
    // access the value at the address available in pointer
    cout << "Value of *ipointer variable: ";
    cout << *ipointer << endl;
    return 0; }
```

Null Pointers

A pointer that is assigned to NULL is referred to as a null pointer. A null pointer is assigned at the time of declaration. The null pointer is generally

constant and holds a value of zero.

Chapter 15 Expressions and Operators

A JavaScript expression can be loosely defined as a block of code that yields a value. You've already seen examples of simple expressions:

- A literal value, such as 5, "Hi!", true, null, and undefined. (Later in the book, you'll learn how to formulate literal values for objects, arrays, and functions. These also constitute expressions.)
- A variable. For example, if you've declared a variable `x`, then an occurrence of `x` in your code is an expression. The value of the expression is the current value assigned to the variable.
- A function call. The value of a function call is the value returned by the function, or undefined if the function doesn't return a value. For example, the function call `parseFloat(str)` is an expression that has a numeric value.

Note

When an expression forms the left side of an assignment statement, such as the `x` in the statement `x = 5`; it's more accurate to say that the expression yields the location or address of a value. The value at that location is replaced by a new value through the assignment.

Don't confuse an expression with a statement. A statement is a basic unit of JavaScript code, usually terminated with a semicolon.

Some statements do not contain any expressions, such as a variable declaration:

```
var x;
```

An expression often forms part of a statement, such as the occurrence of the variable `valid` in the following `if` statement:

```
if (valid)
```

```
/* do something... */
```

And, finally, it's legal for any expression to appear as a complete statement, although it might not serve a useful purpose:

```
/* this literal string expression forms a legal but useless statement: */
```

```
"Hello";
```

```
/* this statement, which consists of an occurrence of the previously declared  
variable 'x', is also legal but futile: */
```

```
x;
```

```
/* this function call expression likewise forms a legal statement, but has a  
useful side effect: */
```

```
alert ("an alert message");
```

```
~ ~ ~
```

You can combine expressions using operators to form more complex expressions. For example, assuming the variable `x` contains a number, the following expression consists of two numeric expressions, one a variable and one a literal number, combined with the `+` (addition) operator:

```
x + 5
```

If the value of `x` is 2, then the value of the resulting, more complex expression is 7. The expressions that are combined using an operator (in this example, the `x` and the 2) are known as operands.

Operator Precedence

You can include more than one operator in a single expression:

```
2 * x + 3 * y + z
```

The value of such an expression depends upon the order in which the operations are performed. To define the order of operations, the JavaScript operators have been assigned different levels of precedence. For example, the `*` operator (multiplication) has a higher precedence than the `+` operator (addition), so in the above expression the multiplications are performed before the additions.

You can specify any order of evaluation by using parentheses. You might want to parenthesize an expression if you're uncertain of the precedence of the operators, or if you simply want to clarify your code by making the order of operations explicit:

```
/* here the parentheses clarify the default evaluation order: */
```

```
(2 * x) + (3 * y) + z
```

You can also use parentheses to change the evaluation order from the default. For instance, if you wanted the additions to be performed before the multiplications in the example expression, you could parenthesize it like this:

```
/* here the parentheses change the evaluation order: */
```

```
2 * (x + 3) * (y + z)
```

The following table describes the basic JavaScript operators that are used in this book, listing them in precedence order. The horizontal bars divide the table into precedence sections. Operators in the same section (such as `.` and `[]`) have the same precedence. Operators in different sections (such as `()` and `++`) have different precedence, with the operator in the first section having the higher precedence. The "A" column is explained following the table. You'll find a complete operator precedence table that includes the more esoteric operators at <http://bit.ly/OpTable>.

Operator	Description	A

.	Access a property of an object	L
[]	Access an element of an array	L
()	Call a function	L
new	Create a new object	R
++	Increment	R
--	Decrement	R
-	Negate (unary)	R
!	Logical NOT	R
delete	Remove a property from an object	R
typeof	Get data type	R
* / %	Multiply, divide, get remainder	L
+ -	Add, subtract	L
+	Concatenate string	L
< <=	Less than, less than or equal	L
> >=	Greater than, greater than or equal	L

instanceof	Test object inheritance	L
in	Test property existence, or construct a for/in loop	L
==	Test for equality, loosely	L
!=	Test for inequality, loosely	L
===	Test for equality, strictly	L
!==	Test for inequality, strictly	L
&&	Logical AND	L
	Logical OR	L
? :	Conditional operator	R
=	Assign	R
+= -= *= /= %=	Assign, with operation	R
,	Return right operand	L

If operators have the same precedence, their order of evaluation is determined by their associativity, which is given in the "A" column. Operators with "L"

associativity are evaluated left-to-right. For example, because the + operator has left-to-right associativity, the expression

$$a + b + c + d$$

is evaluated like this:

$$((a + b) + c) + d$$

Operators with "R" associativity are evaluated right-to-left. Because the = operator has right-to-left associativity, the expression

$$p = q = r = s;$$

is evaluated as

$$p = (q = (r = s));$$

(As you'll see later, the value of a single assignment expression such as $r = s$ is simply the right operand.)

Note

The book uses the term "assignment statement" as a short description for what is more accurately termed "a statement consisting of an assignment expression."

The number of operands that an operator expects is either 1 (a unary operator), 2 (a binary operator), or 3 (a ternary operator). The unary operators, the one ternary operator (?:), and the various assignment operators all have right-to-left associativity. The remaining binary operators have left-to-right associativity.

The following sections explain many of the operators given in the table. Some of the operators are used in conjunction with objects, arrays, or functions, and are discussed in later chapters that cover those topics. The typeof operator was explained in the previous chapter.

Numeric Operators

The operators `+` `-` `*` `/` and `%` perform addition, subtraction, multiplication, division, and the modulus operation, as in the following expression:

```
2 * a + b / 3 - 7 * c
```

Recall that multiplication, division, and the modulus operation are performed before addition and subtraction.

In some languages, if both operands are integers, the `/` operator performs integer division—that is, it returns an integer result. For example, `5 / 2` would equal 2. JavaScript, however, does not truncate the result of a division to an integer simply because the operands are both integers, so in JavaScript `5 / 2` equals 2.5.

The modulus operator (`%`) divides the first operand by the second, and returns the remainder of the division. For example:

Expression	Value
<code>10 % 4</code>	2
<code>-10 % 4</code>	-2
<code>10 % -4</code>	2
<code>-10 % -4</code>	-2

Notice that the sign of the result is always the same as the sign of the first operand.

As an example of using the `%` operator, if `n` is a number, the following expression could be used to test whether `n` is odd:

```
if (n % 2)
/* then n is an odd number */
```

You can reverse the sign of a numeric value by placing the `-` operator in front of it. In this context, `-` acts as a unary negation operator rather than a subtraction operator.

```
var x = 5;
```

```
/* displays "-5": */  
alert (-x);  
/* displays "5": */  
alert (-(x - 10));  
x = -7;  
/* displays "7": */  
alert (-x);
```

Note

JavaScript also provides a set of functions for performing operations on numbers, many of them more advanced than the actions performed by the numeric operators, such as calculating a square root or sine.

If you want to increment a variable, you can of course do it like this:

```
x = x + 1;
```

A more concise way to increment a variable, however, is to use the ++ (increment) operator:

```
++x;
```

The ++ operator is an example of an operator that has a side effect (incrementing the variable) and returns a value (as do all operators). If you place the ++ in front of the variable, the operator returns the incremented value. If you place the ++ after the variable, it returns the original, unincremented value:

```
var x = 5;  
/* displays 6: */  
alert (++x);  
/* displays 6: */  
alert (x);  
/* displays 6: */  
alert (x++);  
/* displays 7: */
```

```
alert (x);
```

To decrement a variable, you can use the `--` (decrement) operator, which is analogous to `++`.

Note

You can use the `++` and `--` operators with any expression that specifies a data item whose value can be changed. Such an expression is known as an lvalue, the `l` referring to the fact that the expression can appear on the left side of an assignment operator. The most common lvalue is simply a variable. You'll learn about two other lvalues later in the book: a property of an object and an element of an array.

Numeric Conversions

If an operand of a numeric operator is not numeric, JavaScript automatically converts it to a numeric value, following somewhat complex and arbitrary rules.

Arithmetic Errors

If you perform an illegal arithmetic operation, JavaScript, in its usual forgiving manner, simply returns a special value rather than generating a runtime error.

An expression that divides a positive number by zero returns `Infinity`. An expression that divides a negative number by zero returns `-Infinity`:

```
var x = -5;  
/* displays "-Infinity": */  
alert (x/0);
```

The following expressions return the special value `NaN`, meaning not-a-number:

- `0 / 0`
- `Infinity / Infinity` (also equals `NaN` if either or both operands are `-Infinity`)
- An expression that attempts to take the square root of a negative number (for example, calling `Math.sqrt (-5)`)

- An expression that attempts to convert an invalid value to a number (for example, calling `parseInt("bad");`)
- An expression in which any operand has the value NaN

The value NaN is considered unequal to any other value including itself, so the expression `NaN === NaN` is actually false. To test for NaN, you can use the global function `isNaN()`:

```
/* get a number as a string: */  
var s = prompt("Enter a number:", "");  
/* attempt to convert the string to a number 'n': */  
var n = parseFloat(s);  
/* check whether the conversion was successful: */  
if (isNaN(n))  
/* 'n' is not a valid number: */  
alert("You did not enter a valid number.");  
else  
/* 'n' is a valid number; use it... */
```

Alternatively, you can use the `isFinite()` function to test for either NaN or \pm Infinity:

`isFinite(x)`: Returns true if x is neither NaN nor \pm Infinity; returns false if x is equal to either of these special values.

The String Concatenation Operator

The + operator can also be used to combine two strings into a single string:

```
var s = prompt("Enter your name.", "");  
alert("Your name is " + s + ".");
```

One common use for string concatenation is to convert a non-string value to a string and combine it with another string, all in a single operation:

```
var count = 0;  
/* code that increments count... */  
alert("Final count: " + count);
```

If only one of its operands is a string, + converts the other operand to a string and joins the two strings. (The + operator favors a string over a number. If one operand is a string and the other a number, + does not attempt to convert the string to a numeric value and add the two numbers!)

Recall from that the automatic conversion of a number, boolean value, null, or undefined to a string works as expected and is one of the benign forms of automatic conversion. An easy way to make this conversion is by using the + operator. If you don't want to add any characters, you can just concatenate with an empty string:

```
var count = 0;  
/* code that increments count... */  
/* done incrementing 'count'; now convert it to a string: */  
var s = count + "";
```

Boolean Operators

The boolean operators all return boolean values, and are of two types: relational and logical.

Relational Operators

`==` and `!=` are the standard equality and inequality operators. For example, if `a`, `b`, and `c` are declared and initialized as follows,

```
var a = 1;
```

```
var b = 1;
```

```
var c = 2;
```

then:

<code>a == b</code>	<code>true</code>
<code>a == c</code>	<code>false</code>
<code>b != c</code>	<code>true</code>

The problem with the standard equality and inequality operators is that if the two operands are not of the same type, the operator follows a complex set of somewhat arbitrary and hard-to-remember rules that determine whether a conversion takes place, which operand is converted, and the way a conversion is performed.

Although you could make it a practice to use operands of the same type with the standard operators, a safer approach is to use the strict equality and inequality operators, `===` and `!==`, which do not perform type conversions. The `===` operator returns true only if all three of the following conditions are true:

- Both operands have the same type.
- Both operands have the same value.
- Neither operand is NaN.

The `!==` operator returns true if one or more of these conditions fails.

Provided that both operands have the same data type, you can use the strict

equality and inequality operators with any type—for example:

5 === 5	true
5 !== 10	true
"abc" === "abc"	true
"ABC" === "abc"	false
true === true	true
true !== false	true
undefined === undefined	true
null === null	true
NaN === NaN	false

You can compare two strings or two numbers using the comparison operators:

<	less than	5 < 12 (true) "5" < "12" (false)
<=	less than or equal	5 <= 5 (true)
>	greater than	"abc" > "ABC" (true)
>=	greater than or equal	"1" >= "A" (false)

Strings are compared alphabetically, based on the Unicode values of the characters. For example, within the set of standard ASCII characters, all numbers are less than the letters, and all capital letters are less than the lowercase letters.

Numbers are of course compared numerically. When comparing numbers, if

either operand is NaN, the operator always returns false.

JavaScript doesn't provide strict versions of the comparison operators. If you supply operands that are neither both strings nor both numbers, the comparison operators make conversions according to fairly complex and arbitrary rules. Again, to avoid errors and to create clear code, it's best to first do any necessary conversions explicitly so that you can make both operands the same type.

Logical Operators

Sometimes you need to express a truth condition that is more complex than a simple relational test. For example, in code that searches for an item, you might want to display the results only if the item was found (i.e. the boolean variable `found` is true) and the user wants results shown (i.e. the boolean variable `showResults` is true). This could be expressed using the logical AND operator, `&&`:

```
if (found && showResults)
```

```
/* then display the search results... */
```

The following are the three logical operators:

Op	Description	Example
<code>&&</code>	AND (true if both operands are true)	<code>a && b</code>
<code> </code>	OR (true if either or both operands are true)	<code>a b</code>
<code>!</code>	NOT (true if the single operand is false, false if the operand is true)	<code>!a</code>

If an operand is not boolean, it's converted to boolean according to the straightforward rules that were explained in `Any type to boolean`.

Several logical operands can of course be used in a single expression, as in the following example:

`!(a && b) === (!a || !b)`

This particular expression is always true regardless of the truth values of its operands (it's one of DeMorgan's laws). Recall from the table of Operator Precedence that `!` has the highest precedence, then `===`, then `&&`, and finally `||`, but that the parentheses override the default order of evaluation.

Short Circuiting

When an `&&` expression is evaluated, if the first operand is false, the second operand is guaranteed not to be evaluated. This policy can be useful: The first operand can test whether some object exists before the second operand attempts to use that object, thus avoiding the runtime error that would occur if you attempted to use a non-existent object.

Likewise, when an `||` expression is evaluated, if the first operand is true, the second operand is guaranteed not to be evaluated.

Note

The Truth About the `&&` and `||` Operators

The `&&` and `||` operators don't necessarily return an actual boolean value. Consider an `&&` expression:

`A && B`

If `A` is false or converts to false (that is, it's "falsy"), the expression simply returns the value of `A`. If `A` happens to be a boolean variable, then the expression does in fact return a boolean value (i.e. false). If `A` isn't boolean, the expression doesn't return a boolean value, but rather `A`'s actual value, which is some falsy value: undefined, null, 0, NaN, or "". (Conversions to boolean values were explained in `Any type to boolean`.) If `A` is "truthy," then the expression returns the value of `B`, which might or might not be boolean, but like all values will be either truthy or falsy.

Now consider an `||` expression, which works in a similar way:

`A || B`

If `A` is truthy, the expression returns the value of `A`. If `A` is falsy, it returns the value of `B`.

Because of the way it's evaluated, the `||` operator is often used to select the first truthy value in a series of values. Say, for example, that you want to use

the current value of variable `x` if it has a non-zero value, but you want to use a default literal value (100) if `x` is set to 0 (or undefined). You could accomplish this through the following compact `||` expression:

```
x = x || 100;
```

Assignment Operators

The standard assignment operator (=) has the following form,

```
lval = expression;
```

where the left operand, lval, is assigned the value of the right operand, expression. For example:

```
x = 2 * y + z + 5;
```

Like the operand of the ++ or -- operator explained previously, the left operand of the assignment operator must be an lvalue—that is, an expression that specifies a data item whose value can be changed, such as a variable. (You'll learn about two other kinds of lvalues later in the book: a property of an object and an element of an array.)

The assignment operator is an example of an operator that has an important side effect (namely, making the assignment). Like all operators, it also returns a value, which is simply the value of the right operand. This method of evaluation, coupled with the operator's right-to-left associativity), allows you to chain assignments like this:

```
a = b = c = 5;
```

When JavaScript evaluates this expression, it first assigns 5 to c, then takes the value of that assignment (namely 5) and assigns it to b, then takes the value of that assignment (5 again) and assigns it to a. As a result, all variables get set to 5.

JavaScript also provides a set of shortcut operators that perform an operation and make an assignment. Here are the most common ones:

Operator	Example	Equivalent to
+=	x += 5; s += " ft";	x = x + 5; s = s + " ft";
-=	x -= 5;	x = x - 5;
*=	x *= 10;	x = x * 10;

/=	x /= 2;	x = x / 2;
%=	x %= 4;	x = x % 4;

Other Operators

The final two operators discussed in this chapter are the conditional operator and the comma operator.

The Conditional Operator

The conditional operator (?:) is JavaScript's only ternary operator (an operator with three operands). An expression looks like this:

```
bool_val ? val1 : val2
```

If `bool_val` is true (or converts to true), JavaScript evaluates and returns `val1`. If `bool_val` is false (or converts to false), JavaScript evaluates and returns `val2`. Note that only one of the second two subexpressions (`val1` or `val2`) is evaluated, never both. For an explanation of the way values are converted to boolean, see *Any type to boolean*. You'll see examples of the use of this operator later in the book.

The Comma Operator

Finally, there is the strange comma operator (,), which is a binary operator that is used like this:

```
op1, op2
```

The operator evaluates `op1`, evaluates `op2`, returns the value of `op2`, and discards the value of `op1`. The operator actually has a common use in the `for`-loop statement.

What Are Some Of The JavaScript Variables?

There are a lot of different parts that you are able to work on when you are creating your JavaScript code. You will need to have all of these in place if you would like to make a great code, but learning each of the parts on their own, before combining them together, will make a big difference in how well you understand them. In this chapter, we are going to look at how you can write out variables inside of JavaScript.

The variables are basically the parts in JavaScript that will have a value, and they are only going to have this value once you assign it. They are going to be the different kinds of characters that you use inside of the code. The nice thing about this is you will be able to make the variable equal to any value

that you would like, as long as you set it inside of the compiler. There are a few types of data that are available for you to use when you are working on the variables inside of your code. Some of these data types include:

- **Text:** this is the type of data that will be a sequence of words that then make up a statement. If you would like the program to say something on your screen when the user enters information, it would be done with the text variable. You could keep it something simple like, “Hello!” or you can make it something longer like, “Thank you for visiting my page. Come back soon!”
- **Numbers:** you can turn your variables into a number. You would be able to make a choice between using whole numbers or decimal numbers inside of the code to make it work the way that you would like.
- **Boolean:** the third data type that you are able to use in your JavaScript variables are the Boolean variables. These are going to work on a true and false basis and will be based on the information that you place into your code. For example, these are helpful if you would like something to come up on the screen only if the input is considered true.

One thing that you should keep in mind when using the JavaScript language is that there is going to be hardly any difference in the floating point or integer values that you are using. You will find that JavaScript doesn't see a difference between these, and you will be able to use them the same inside of your code. You won't have to spend much of your time worrying about how they work inside of the code.

Working with your variables

For the most part, you will see that the variables inside of JavaScript are going to be pretty easy to work with. When working with these variables, you should remember that they are the containers that you are using to hold all the data inside the code. You will be able to choose how you are using these containers and what information is going to go inside each of them. The thing that you need to remember here is to use the perfect keyword, which is going to be the “var” keyword in this language, so that you can tell the compiler

that you are working with variables at this time. It also helps the compiler to know the container that it should bring out for you to use.

It is fine if this seems a little bit difficult to understand right from the beginning. Here is a good example of a code that uses the variable keyword so that you can see how things are going to work for your needs:

```
<!DOCTYPE html>

<html>
<head>
<meta charset = "ISO-8859-1">
<title>JavaScript Variables</title>
<script type = "text/javascript">
var name = "Appy";
var age = 21
var salary = 10000, expenses = 12000;
alert("Name:" +name + "Age:" + age + "/nSalary:" + salary
+"Expenses:" + expenses)
</script>
</head>
<body>
</body>
</html>
```

Despite this code not being all that long or big, there is actually quite a bit of information that you can find inside of it. When you look at the variable, you will need to start out with the variable initialization. This is going to help you to create a new variable that has a value attached to it. You can also do this when you choose to use the variable, rather than at the beginning of your code. The syntax that we listed above is the one that will make it easier to bring up some of the variables that you want to use, such as age, name, and salary. Keep in mind that you are able to use as many of these variables as

you would like based on how you are setting up the code.

When you are working with the code above, there are several variables that you will notice and the user will be able to place the input that they want inside of there. For example, the user could choose to put in any answer that they want when the system asks what their age is. When this particular code is executed onto the screen, using the inputs that are placed in the code above, the following information is going to be the output:

Name: Appy

Age: 21

Salary: 10000

Expenses: 12000

Learning the scope of your variable

The next thing that we will discuss about your variables is their scopes. This is going to make a difference as it determines where the variable is going to be located and where you are able to see that variable within the code. It is going to help you to find where the variable is located and can give you a bit of control over who has permission to see these variables as well. There are two types of scopes that can be placed on the variable and these include global variables and local variables:

- **Local variables:** the first type of variable that you will work with are the local variables. These are the ones that you will only be able to see when you look at their exact location inside of the code. You will not be able to see them anywhere else, which can help to give you a bit of security with the code and makes it easier to set up the parameters for that particular variable.
- **Global variable:** the other variable that you are able to use. With this option, you will be able to see the variable at any point, rather than just in its regular location. This kind of code is viable for anyone to change it, which could cause issues if you would like to make sure that everything stays the same. They will be present in many locations inside of the HTML document.

When you are choosing which of these two variable types that you would like to use, you should make sure that they are given names that not only help you to determine what is in them, but also that the names are not similar. For example, if you have a variable that is local as well as global, you should name them separate things, otherwise the program is always going to choose the local variable over the global variable.

Naming the variable

While we have spent some time learning about the variables and how they work within your system, it is also important to learn how you should name them. Naming the variables may seem simple, but it is important that you do it the proper way to help the program work the way that you want. You don't want to end up naming the variable the wrong thing and creating an error down the road. Some of the things that you will need to keep in mind when naming your variables include:

- Pick a name that makes sense: there are a lot of names that would be cool for a variable, but if you look at the name later on and have no idea what it holds inside, it is probably not the best name to choose. Go with a variable name that tells you what is inside and makes it easier to find and use the variable later on.
- Keep away from the keywords: in every programming language that you can use, there are some keywords that are reserved for giving commands. If you use these as the names of your variables, you are going to have some trouble because the compiler will get confused at what you are using. Make sure to stay away from the reserved keywords in JavaScript when you are naming your variables.
- Keep away from numbers: while you are allowed to use numbers within the name of your variable, you are not allowed to use them to start out the name. You can write out the name of the number so that it is all letters, but no numbers are allowed at the beginning. Start out the variable name with a letter or an underscore, and then the rest can be any combination of numbers, letters, and underscores that you

wish.

- Remember that when you are writing out the name of the variable that they are going to be case sensitive. There are differences between the upper case and lower case in this language so make sure to use them properly or you may confuse the compiler or not get the results that you want.

If you are able to remember some of these simple rules inside of your variables when you are naming them, you will find that it is easier than you can imagine to name some of the variables. These rules help you to get the results that you want, help you to remember what the variable is holding, and even ensures that the other programmers who are working in your program have an idea of what the variable is doing.

During this time, you may want to go through and assign the value that you would like to go with the variable you are working on. The variable is empty if you don't add in a value that is going with it. The nice thing here is that you get some freedom in terms of the value that you would like to assign. You can keep it simple and give the variable a number as a value or you can give it a word, such as a color, a name, or something else that will be assigned to the variable. The value that you will assign to your variable is going to be different based on what you would like to get done within the code that you are writing.

Working with variables is important when you are inside of the JavaScript programming language. They are going to help you to do some classification within your programming and can help you to store different categories together all in one place. As long as you learn how to use the variables in the correct way, using some of the codes that are available inside this chapter, and you name the variables the right way, you will find that the variables are going to work out perfectly for your needs.

Chapter 16 Variables, data types & constants

Variables

Variables are always an important aspect of any programming language. They are used to store values which can be later addressed using those variable names. Same goes with JavaScript. Let us look at the syntax for creating a variable:

Syntax

var <variableName>;//Declares a variable

<variableName> = value;//Assigning value to the variable

Or

var <variableName> = value; //Declaration and assignment in same line

In the first method, we declare the variable first and then assign a value to it at some later point in the program. The only thing one must keep in mind here is that declaration must always come before the assignment. However, in the second method, both the declaration as well as the assignment happen in a single line.

Let us look at some examples here

```
var abc;//Declaration
```

```
abc = "New variable";// Assignment
```

```
var num = 33;//Declaration and assignment
```

In above example, the variable "abc" will store the string value "New variable" while the variable "num" will store the number value 33.

Let us look at one more example:

```
var marksScience = 85; //Variable name
```

```
var marksMaths= 100;
```

```
var total = marksScience+ marksMaths; //add and store the result
```

In the above example, the variable total will store the

calculated value i.e. 185.

All the variables which are used in the program must have unique names or else it might end up creating a ruckus while executing the code. These unique names are called identifiers. For naming a variable, certain rules must be followed.

Rules for naming a variable

- Names must start with a letter, dollar (\$), or underscore (_).
- You can include letters, digits, underscores, and dollar signs in a variable name.
- Variable names in JavaScript are case-sensitive.
- You cannot use reserved words / keywords as variable names.

Some of the examples of valid variable names are as follows:

Marks1

\$abc

_fun

marks2

You would have noticed that we make use of an “equal to” symbol to assign some values to a variable. This “equal to” symbol is called assignment operator and is used to initialize a variable.

Few important points to remember:

1. When you simply declare a variable, it holds no value.

For example:

```
var abc;
```

Here the variable “abc” will not contain any value until we assign it some value.

2. Many variables can be declared in a single statement. To do that, we start the statement with var and separate the variables by a comma.

For example:

```
var firstName="Amitabh", lastName ="Kumar", age=70;
```

3. The declaration of variables can span multiple lines.

```
var firstName="Amitabh",  
lastName ="Kumar",  
age=70;
```

4. Most of the time, variables are simply declared and are initialized at some later point in the program. As mentioned in the first point, after the declaration, the variable holds no value. To be precise, it holds the value undefined. For example:

```
var name;
```

The variable “name” will contain the value undefined after the above statement is executed.

5. Even if you re-declare a JavaScript variable, it retains its previous value.

For example:

```
var name="Amitabh";  
var name;
```

Even after the execution of these statements, the variable “name” will still have the value “Amitabh.”

Data Types

Data types, as the name suggests, signifies the type of data a variable can hold. A JavaScript variable can hold several types of data. For example string, number, object, Boolean, etc.

```
var age = 70//stores a Number
```

```
var lastName = "Kumar"// stores a String
```

```
var name = {firstName:" Amitabh",  
lastName:"Kumar"};  
stores an object
```

```
var case=true; //Stores a Boolean
```

Data types is an important aspect of programming because it specifies what kind of data a variable is holding. This helps the compiler to understand what kind of operations can be performed on them. The addition operator will have different results if applied to two numbers and a radically different result if applied on two strings. For example:

```
var name ="Amit" + ""+ "Kumar";
```

```
var age = 30+40;
```

The variable “name” will store the concatenated value of both the strings i.e. “Amit Kumar” while the variable age will store the added value of 30 & 40 i.e. 70.

Now consider the next example:

```
var abc = 70 + "Amit";
```

If we try and relate the above example with normal arithmetic, it does not makes much sense. Adding a number to a string is not possible in normal arithmetic. However, it would be interesting to see how JavaScript will treat the above example. JavaScript will interpret the above example as follows:

```
var abc = “70” + "Amit";
```

Whenever we try to add a number with a string, JavaScript will treat the number as a string. However, what if our expression is a mix of several numbers and several strings! What will be the result in that case? Let’s look at the following two examples:

```
var temp = “Amit” + “ “ + “Kumar” + 30 + 40;
```

```
var temp1 = 30 + 40 +“Amit” + “ “ + “Kumar” ;
```

In the above examples, the temp will store the value “Amit Kumar3040” whereas the variable temp1 will store “70Amit Kumar”. How similar looking expressions produced different results? Let’s see how JavaScript treats the above two example. JavaScript starts evaluating expressions from left to right. So in the first case when it encountered a string, it considers all the remaining as a string itself. Therefore 30 and 40 becomes two separate strings. However, in the second example, since the numbers are written first, it first adds them and then concatenates with the remaining string.

Before we dive into each type of JavaScript data types, I must point out that

JavaScript has dynamic types. Let me explain it to you what that means. In C language, we use data types such as int, float, char, etc. Those are static data types which mean they cannot take any other type of value during the execution of the program. However, we see no such types here. We simply declare a variable using the keyword “var” and then go on to mutate it the way we want. Even if we have stored a string value in the variable, at some later point in the program, we can assign a number to it. Therefore the types in JavaScript are dynamic in nature. For example:

```
var rand;//rand is undefined
```

```
var rand= “Random”;//rand stores an string
```

```
var rand= 70;//Now rand stores a number
```

Now that we have looked at properties of data types let’s look at each data types in detail.

There are two kinds of data types in JavaScript:

- a. Primitive data
- b. Non-primitive or complex data

Primitive is the predefined types of data which are inbuilt to a language.

There are five primitive data types in JavaScript:

- String
- Number
- Boolean
- Null
- Undefined

Non-primitive or complex data are not provided by default but are defined by the programmer. For example arrays, objects, etc.

Strings

Strings are nothing but a series of characters. They are either enclosed in single or double quotes. Addition operation on them is nothing but a simple concatenation.

For example:


```
var name = "Amit Kumar";
```

```
var name2= 'Javed';
```

Quotes inside a string are not a problem as long as they are different than the quotes surrounding the string. For example:

```
var xyz = "It's done";
```

```
var xyz1 = 'Friends call him "James"';
```

Numbers

Unlike most of the other programming language, JavaScript does not have a concept of integer, float, etc. Instead, it just has a single concept of numbers which may or may not contain decimals

```
var x = 74.00; // Decimals
```

```
var y = 33; // Without decimals
```

If the number is extremely large, it can be written using scientific notation. For example:

```
var x = 333e5; // 33300000
```

```
var y = 23e-5; // 0.00023
```

Booleans

The concept of Boolean is almost same as that of other programming languages. Booleans have only two values – “true” and “false.” The major use of these comes while forming conditional statements.

```
var a = true;
```

```
var b = false;
```

Null

Null means “nothing.” The value null signifies the intentional absence of any object value. An object can be emptied by setting it to null.

```
var name=null; //value becomes null
```

Undefined

The variables which are not assigned any value will contain undefined. For example:

```
var abc; // value is undefined
```

We can also assign the value “undefined” to a variable.

```
abc= undefined;
```

People generally get confused between null and undefined. I will explain the difference at some later point in this chapter.

Those were the primitive types. Now let us look at some non-primitive data types.

Arrays

Arrays are nothing but a collection of some values. The set of those values is written using curly braces. Like all other programming language, the index starts with 0.

```
var names[ ] = { “Ram”, “Rahim”, “Jaspaal”};  
//names=”Ram”;
```

Objects

Objects in JavaScript are written as name- value pairs. For example:

```
var name = {firstName:"Amit",  
lastName:"Kumar",  
age:70};
```

Before I close in on the discussion on data types, I must explain about an operator called “typeof.” Since JavaScript is dynamically typed, at various point in the program we need to know which type of data the variable contains. We can determine that using typeof operator.

typeof Operator

Like I mentioned before, typeof operator is used to determining the type of data a variable is holding. For example

```
typeof "Amit"           // "string"  
typeof ""              // "string" (empty string)  
typeof 314              // "number"  
typeof 3.14            // "number"
```

```
typeof (3 + 4)      // "number"  
typeof false      // "boolean"  
typeof [1,2,3,4]   // "object"because arrays are considered as object in  
JavaScript  
typeof {name:'Amit', age:70} // "object"
```

Now that I have explained typeof operator let us look at what exactly is the difference between undefined and null.If we operate typeof on null, it returns an object whereas if we operate typeof on undefined, undefined is returned.

Constants

Constant as the name suggests are those variables whose value cannot be changed. These are declared using const Keyword and must be assigned a value during the time of declaration. It can either be local to a function or can be a global variable. Constants once assigned cannot be changed. Their value remains read-only throughout the program. For example:

```
const pi= 3.14; //Declaring and initializing a constant
```

This chapter explained variables, data types, and constants. I hope you enjoyed reading it and found it helpful.

Chapter 17 Closures and Callbacks in JavaScript

The chapter will deal with two very basic concepts of JavaScript, closures and callbacks. They allow us to make our code concise, crisp and creative. You will encounter these concepts in your development repeatedly. Let us have a look.

Closures in JavaScript

Closures, in their most basic form, are inner functions that have access to variables from outside its scope. The concept of closures requires a legit understanding of nested functions.

Nested functions are “functions containing functions”. Let’s look at an example to expand upon the point:

```
//function 1 is an outer function
function outer(){ //definition of function 1
//body of outer()
//function 2 is an inner function
function inner(){//definition of function 2
//body of inner()
}
}
```

In above example, function 2(inner()) resides inside the body of function 1(outer()) and hence is called a nested function. Similarly, we can have more than one nested function inside an outer function.

```
//function 1 is an outer function
function outer(){ //definition of function 1
//body of outer()
//function inner1 is an inner function
function inner1(){//definition of inner function 1
//body of inner()
}
//function inner2 is an inner function
```

```

function inner2(){//definition of inner function 2
//body of inner()
}
.
.
//function innerN is an inner function
function innerN(){//definition of Nth inner function
//body of inner()
}
}

```

The ability of the inner function to evaluate or perform calculations on the variables or parameters defined in outer function is closure.

Let us take a very basic example of JavaScript code.

//Example 1

```

//define a variable a = 5
var a = 5;
//define function add which adds 5 to a and returns value
var add = function(val){
a = a + val;//adding val to a
return a;//return a
}
var result = add(5); //will make result = 10

```

In above code, the function add() will take the value of variable a from its outer scope, and will add 5 to it and return the result. So, our output will show 10. This is the most basic use of closures. Surprised? You might have used it multiple times while developing code by now.

Closures are popular, as the inner functions are able to retrieve variables from outside their scope, modify their values, and return the result. The closures have access to only the following variables:

- variables inside its scope, i.e. variables defined inside the function body
- variables of outer function, i.e. variables of containing function
- global variables

Let us move to a more complicated scenario:

//Example 2

//define function addTo() which takes a parameter

```
var addTo = function(param_outer){
```

```
//define a variable sample of value 10
```

```
var sample = 10;
```

```
//define function add which takes another parameter
```

```
var add = function(){
```

```
return sample + param_outer;
```

```
}
```

```
//return the inner function
```

```
return add;
```

```
}
```

Here, we have a nested function add() which can take in parameters from the outer function addTo(). When user makes a call to addTo() with some parameter, he will get the inner function as a return for it, which in turn will execute the addition and return the value.

```
var test = addTo(10);
```

Internally, param_outer will be equal to 10, and add will take value of addition of param_outer and sample and will evaluate to 20.

```
var add = 20 ;
```

When the addTo() call returns add, the value of test will equal to that of add, which is 20.

Let us take an even more complicated example:

//Example 3

//define function addTo() which takes a parameter

```
var addTo = function(param_outer){
//define function add which takes another parameter
var add = function(param_inner){
return param_inner + param_outer;
}
//return the inner function
return add;
}
var test = addTo(10); //value will not be 20
```

Here, test will take the value of inner function and will become a function itself, which will have a preserved variable, param_outer equal to 10, set by us.

So, test will internally look like this:

```
//this is only a representation of how it will internally look
var test = function(param_inner){
return param_inner + 10;
}
```

If we want to add 10 to any number, we can simply write:

```
var x = 10;
var y = test(x);
```

This will make the value of y equal to 20, or 10 + 10.

A good use case is when we want to add a constant to some numbers. We can use this approach for closures.

Example:

```
var addThree = addTo(3);
var a = addThree(10); //a will have 3 + 10 = 13
var a = addThree(20); //a will have 3 + 20 = 23
var a = addThree(30); //a will have 3 + 30 = 33
```

Features of Closures

- The inner functions have access to outer function variables even after they have been executed, as we saw in our third example. We called the outer function before calling the inner function and even after returning the value, we could use the outer function parameters inside the inner function as it was preserved.
- Closures store references and not actual values to outer functions variables and can be edited, if needed (refer example 4).
- Closures provide a secure way of dealing with certain data, as we can save references to variables and prevent accidental changes.

//Example 4

//define a variable a = 5

var a = 5;

//define function add which adds 5 to a and returns value

var add = function(val){

a = a + val;//adding val to a

return a;//return a

}

a = 10; //update value of a

var result = add(5); //will make result equal to 15 and not 10 as //value of a is updated before the closure is called so saved //reference is updated with new reference

Closures are powerful but due care needs to be taken, as updated references can make your code full of bugs. Always make sure to use the capability carefully.

Note: Closures are functions or inner function which have preserved data referenced inside them from global or outer scopes.

Callbacks in JavaScript

JavaScript functions are first-class objects. “First-class objects” mean that they can be used like other objects in the body. Like variables, we can save their references, pass them as arguments, declare nested functions (refer to previous sections), and can be return values of functions as we saw earlier.

The capability of a function to be passed as argument to another function makes JavaScript functions special. A passed function can be executed at a later stage, or be returned as a function to be executed later by some other function. This is what constitutes the JavaScript Callbacks.

Callbacks are functions which are passed as parameters to another function and is executed from the body of the containing function. Since the scenarios where such capabilities are of use is limited, they can be referred to as callback patterns.

```
//define a callback function sum()
var sum = function ( num1 , num2 ){
return num1 + num2;
}
//define a function that will use sum() as parameter
var calling_func = function ( a , b , callback ){
return callback ( a , b );//calling the callback function with //parameters
}
var sample = calling_func( 5 , 5 , sum);//new will be assigned 10
```

Let us walk through the code above. First, we declare a simple function that adds two values and return the result. Next, we define a function which takes in two parameters and a third parameter which is “callback”. Callback is basically a variable which will store the name of function we will pass and execute it accordingly. It can be called any name. For ease of understanding, I have named it callback.

The important thing to note is that when we pass a function as variable, we only send the definition of callback function as in the name (say sum), not the function call with parenthesis. The function is executed only when it is called inside the body of the containing function.

As we have seen, callback functions are not called immediately, but are preserved as references and are only called when specified by the calling function, thus making them closures as detailed in the previous section.

Callback functions not only can take existing functions as parameters, but can also define a completely new function in the parameter. It will be called as a

callback by the containing function. Let's go through an example:

```
//no callback function is defined
//define a function that will use sum() as parameter
var sum = function(a,b){
return a+b;
}
var some_func= function( a , b , callback ){
return callback(a,b);
}
console.log(some_func(5,5,function(a,b){
return a-b;
}));
//output
0
```

Here, the function is specified directly with a return value. This works fine, and is used when we have situations where we know a certain function will only be used once or twice during our development.

Another important point to learn is that we can pass parameters to callback functions from inside the containing function.

```
var sum = function(a,b){
return a+b;
}
var b = 10; //define a global variable
var some_func= function( a , callback ){
return callback(a,b);//pass global var as a parameter
}
console.log(some_func(5,sum));
//output
//15
```

Here, we can pass in some reserved or pre-defined parameters to the callback function which can consume it. This results in better code quality and more security.

There is a necessary condition that whatever the callback function is should legitimately be a function. To prevent accidentally passing callback variables as something other than a function, we can put an “if” condition to check `typeof callback`, and only execute further if it is of type “function”.

```
If(typeof callback === "function"){  
  //body of our function  
}
```

Callbacks can be more than one function and can be passed as a parameter, as for a single callback.

```
var sum = function(a,b){  
  console.log("Callback")  
}  
var sum1 = function(a,b){  
  console.log("Callback2")  
}  
var b = 10;  
var some_func= function( smthn , callback ){  
  console.log("containign func")  
  callback();  
  smthn();  
}  
some_func(sum,sum1);  
//output  
containing func  
Callback  
Callback2
```

Here, we are passing in two callbacks and executing them one after another, as per our needs. This is the basic concept of callbacks.

Closures and callbacks are developed to facilitate ease of development for us and should be used as needed. Due attention needs to be paid while using them. Remember, closures are nothing but functions with preserved data. Similarly, callbacks can be generalized to “functions containing functions as parameters”.

Chapter 18 Apply, call, and bind methods in JavaScript

The `apply()`, `call()`, and `bind()` methods in JavaScript allow us to change the value of the "this" keyword with the help of some arguments. These methods are useful for calling a specified or targeted function. All three of these methods are closely related.

In this section, we will review these three methods of JavaScript in brief.

The 'this' keyword

Before diving into the details of `apply()`, `call()`, and `bind()` methods, we need to know about the keyword "this" in JavaScript.

In terms of global context

The keyword "this" refers to the global object outside of any method. In the following example, we will see the scope of this keyword in the global context. We will just `console.log` the "this" keyword in the browser.

```
console.log(this);
```

The result in the console is "window", which is nothing but the global object. This result implies that:

```
console.log(this === window); // is true
```

In the example above, we entered `console.log(this)` which gave us window object.

In this example, we will use strict mode and check the value of the "this" keyword.

```
function abc(){
```

```
'use strict'; // the strict mode
```

```
return this;
```

```
}
```

```
console.log(abc()); // undefined
```

This time, the value of the "this" keyword is undefined because it was not defined inside the function `abc()`.

In terms of function context

Addressing the "this" keyword in terms of function context, we have the three methods `call()`, `apply()`, and `bind()`.

Let's talk about these methods in detail.

The apply method

The `apply()` method invokes a function with the value of “this” keyword, and parameters provided as an array (or an array-like object).

Syntax:

The method `apply()` has the following syntax:

```
functionName.apply(thisArgument, argsArray[])
```

Parameters

Let's have a look at the arguments provided to the `apply()` method.

- `thisArgument`: The value of keyword "this" provided for the invoke the `functionName`.

Note: The value of the keyword "this" may not be the original value seen by the function when working in non-strict mode. The primitive values and the global object will be replaced by “undefined” and “null”.

- `argsArray[]`: An argument of an array-like object that specifies the arguments with which `functionName` should be called, or undefined or null if no arguments should be provided to the method.

Example:

In the following example, we will take a look at the basic functionality of the `apply()` method.

```
var student1 = {firstName: 'Abhishek', lastName: 'Kumar'};
var student2 = {firstName: 'Ashok', lastName: 'Mehta'};
function text(hello) {
    console.log(hello + ' ' + this.firstName + ' ' + this.lastName + ' ? ');
}
```

```
text.apply(student1, ['How are you']); // How are you Abhishek Kumar ?
```

```
text.apply(student2, ['How are you']); // How are you Ashok Mehta ?
```

The call method

The `call()` method invokes a particular function with the help of the "this" keyword and an individual argument.

Syntax:

The method call() has the following syntax:

```
functionName.call (thisArgument, argument1, argument2, ....)
```

Parameters

Let's have a look at the arguments provided to the apply() method.

- **thisArgument:** The value of keyword "this" provided for the invoke the functionName.

Note: As with the apply() method, the value of keyword "this" may not be the original value seen by the function when working in non-strict mode. The primitive values and the global object will again be replaced by undefined and null.

- **Argument1, 2 ... :**Arguments for the different functions

Example:

```
var student1 = {firstName: 'Abhishek', lastName: 'Kumar'};
var student2 = {firstName: 'Ashok', lastName: 'Mehta'};
function text(hello) {
    console.log(hello + ' ' + this.firstName + ' ' + this.lastName + ' ? ');
}
text.call(student1, ['How are you']); // How are you Abhishek Kumar ?
text.call(student2, ['How are you']); // How are you Ashok Mehta ?
```

The bind method

The method bind() calls a newly created function (created by the method itself), which has the value of the keyword "this" set to the specified value, with a given pattern of parameters.

Syntax:

The method call() has the following syntax:

```
function.bind (thisArgument[, argument1[, argument2[, ...]])
```

Parameters

Let's have a look at the arguments provided to the apply() method.

- **thisArgument:** The value to be passed as the keyword this argument to

the specified function when the bound function is invoked.

If the bound function is created using the new operator, the value will be ignored.

- `argument1[, argument2[, ...]]`: Arguments attached to arguments provided to the bound function when calls the specified function.

Example:

```
var student1 = {firstName: 'Abhishek', lastName: 'Kumar'};
var student2 = {firstName: 'Ashok', lastName: 'Mehta'};
function text() {
    console.log('hello' + ' ' + this.firstName + ' ' + this.lastName + ' ?');
}
var textHelloAbhishek = text.bind(student1);
var textHelloAshok = text.bind(student2);
textHelloAbhishek(); // How are you Abhishek Kumar ?
textHelloAshok(); // How are you Ashok Mehta ?
```

Difference between apply, call and bind

The `apply()` method calls the function and allows you to pass in parameters as an array or an array-like object.

The `call()` method calls the function and allows you to pass in parameters one by one.

The `bind()` method returns a newly created function and allows you to pass in any number of arguments.

Now that we know what the `call()`, `apply()`, and `bind()` methods in JavaScript are, the question still remains “when do I use which methods”?

In this section, we will clear this confusion by comparing all of them based on a few parameters.

The method `call()`, and `apply()` are somewhat interchangeable, depending on how you as the programmer decide to utilize them. You have a choice to determine whether it is easier to send a list of arguments separated by commas, or an array of arguments.

The `bind()` method is a bit different from the other two.

Where the `apply()` and `call()` methods execute the current function on the spot, the `bind()` method returns a newly created function.

A `bind()` method can be used for events such as `onClick`, where the firing of such events is uncertain.

We as programmers can't predict when those events will be fired, but we can serve the context to them.

Chapter 19 Events

So far, all the applications and scripts that have been created have something in common: they are executed from the first instruction to the last sequentially. Thanks to the flow control structures (if, for, while) it is possible to slightly modify this behavior and repeat some pieces of the script and skip other pieces depending on some conditions.

These types of applications are not very useful since they do not interact with users and cannot respond to the different events that occur during the execution of an application. Fortunately, web applications created with the JavaScript language can use the event-based programming model.

In this type of programming, the scripts are dedicated to waiting for the user to "do something" (press a key, move the mouse, close the browser window). Next, the script responds to the user's action by normally processing that information and generating a result.

Events make it possible for users to transmit information to programs. JavaScript defines numerous events that allow a complete interaction between the user and the web pages/applications. Pressing a key constitutes an event, as well as clicking or moving the mouse, selecting an element of a form, resizing the browser window, etc.

JavaScript allows you to assign a function to each of the events. In this way, when any event occurs, JavaScript executes its associated function. These types of functions are called "event handlers" in English and are usually translated as "event handlers".

Event Models

Creating web pages and applications has always been much more complex than it should be due to incompatibilities between browsers. Although there are dozens of standards for the technologies used, browsers do not fully support them or even ignore them.

The main incompatibilities occur in the XHTML language, in the support of CSS stylesheets and, above all, in the implementation of JavaScript. Of all of them, the most important incompatibility occurs precisely in the browser's event model. Thus, there are up to three different models to handle events depending on the browser in which the application runs.

Basic Event Model

This simple event model was introduced for version 4 of the HTML standard and is considered part of the most basic level of DOM. Although its features are limited, it is the only model that is compatible in all browsers and therefore, the only one that allows you to create applications that work the same way in all browsers.

Standard Event Model

The most advanced versions of the DOM standard (DOM level 2) define a completely new and much more powerful event model than the original. All modern browsers include it, except Internet Explorer.

Internet Explorer Event Model

Internet Explorer uses its own event model, which is similar but incompatible with the standard model. It was first used in Internet Explorer 4 and Microsoft decided to continue using it in the other versions, despite the fact that the company had participated in the creation of the DOM standard that defines the standard event model.

Basic Event Model

Types Of Events

In this model, each XHTML element or tag defines its own list of possible events that can be assigned. The same type of event (for example, clicking the left mouse button) can be defined for several different XHTML elements and the same XHTML element can have several different events associated.

The name of each event is constructed using the prefix `on`, followed by the English name of the action associated with the event. Thus, the event of clicking an element with the mouse is called `onclick` and the event associated with the action of moving the mouse is called `onmousemove`.

The most used events in traditional web applications are `onload` to wait for the page to load completely, the events `onclick`, `onmouseover`, `onmouseout` to control the mouse and `onsubmit` to control the submission of forms.

The typical actions that a user performs on a web page can lead to a succession of events. Pressing for example on a button of type `<input type = "submit">` triggers the events `onmousedown`, `onclick`, `onmouseup` and

onsubmit consecutively.

Event Handlers

A JavaScript event by itself lacks utility. For events to be useful, JavaScript functions or code must be associated with each event. In this way, when an event occurs, the indicated code is executed, so the application can respond to any event that occurs during its execution.

The functions or JavaScript code defined for each event are called "event handler" and since JavaScript is a very flexible language, there are several different ways to indicate the handlers:

- Handlers as attributes of the XHTML elements.
- Handlers as external JavaScript functions.
- "Semantic" handlers.

Event handlers as XHTML attributes

This is the simplest and least professional method of indicating the JavaScript code that should be executed when an event occurs. In this case, the code is included in an attribute of the XHTML element itself. In the following example, we want to show a message when the user clicks on a button:

```
<input type = "button" value = "Click me and you will see" onclick = "alert ('Thanks for clicking');" />
```

In this method, XHTML attributes are defined with the same name as the events to be handled. The previous example only wants to control the event of clicking with the mouse, whose name is onclick. Thus, the XHTML element for which you want to define this event must include an attribute called onclick.

The content of the attribute is a text string that contains all the JavaScript instructions that are executed when the event occurs. In this case, the JavaScript code is very simple (alert ('Thanks for clicking');), since it is only about displaying a message.

In this other example, when the user clicks on the <div> element a message is displayed and when the user hovers the mouse over the element, another message is displayed:

```
<div onclick = "alert ('You clicked with the mouse' ); " onmouseover = "alert
```

```
('You just ran over the mouse');">
```

You can click on this element or just hover over the mouse

```
</div>
```

This other example includes one of the most used instructions in older JavaScript applications :

```
<body onload = "alert ('The page has been fully loaded');" > ...
```

```
</body>
```

The previous message is displayed after the page has been fully loaded, that is after it has been loaded downloaded your HTML code, your images and any other object included in the page.

The onload event is one of the most used since, as seen in the DOM chapter, the functions that allow access and manipulation of the nodes of the DOM tree are only available when the page has been fully loaded.

Event handlers and 'this' Variable

JavaScript variable defines a special variable called this that is created automatically and used in some advanced programming techniques. In events, the variable this can be used to refer to the XHTML element that caused the event.

Event handlers as external functions

The definition of event handlers in XHTML attributes is the simplest but least advisable method of dealing with events in JavaScript. The main drawback is that it is complicated in excess as soon as a few instructions are added, so it is only recommended for the simplest cases.

If complex applications are made, such as the validation of a form, it is advisable to group all the JavaScript code into an external function and call this function from the XHTML element.

Following the previous example that shows a message when clicking on a button:

```
<input type = "button" value = "Click me and you will see" onclick = "alert ('Thanks for clicking');" /> Using external functions can be transformed into:
```

```
function sample Message () {
```

```
alert ('Thanks for clicking');  
}
```

```
<input type = "button" value = "Click me and you will see" onclick =  
"sample Message ()" />
```

This technique consists of extracting all JavaScript instructions and grouping them into an external function. Once the function is defined, the function name is included in the attribute of the XHTML element, to indicate that it is the function that is executed when the event occurs.

The function call is made in the usual way, indicating its name followed by the parentheses and optionally, including all the necessary arguments and parameters.

The main drawback of this method is that in the external functions it is not possible to continue using the variable `this` and therefore, it is necessary to pass this variable as a parameter to the function.

In the previous example, the external function is called with the parameter `this`, which within the function is called `element`. The complexity of the example is mainly due to the way in which different browsers store the value of the `borderColor` property.

While Firefox stores (in case the four edges match in color) the black value, Internet Explorer stores it as black black black black and Opera stores its hexadecimal representation # 000000.

Semantic event handlers

The methods that have been seen to add event handlers (as XHTML attributes and as external functions) have a serious drawback: they "dirty" the XHTML code of the page.

As is known, one of the basic good practices in the design of web pages and applications is the separation of content (XHTML) and its appearance or presentation (CSS). Whenever possible, it is also recommended to separate the contents (XHTML) and its behavior or programming (JavaScript).

Mixing the JavaScript code with the XHTML elements only helps to complicate the source code of the page, make it difficult to modify and maintain the page and reduce the semantics of the final document produced.

Fortunately, there is an alternative method to define JavaScript event

handlers. This technique is an evolution of the method of external functions, since it is based on using the DOM properties of XHTML elements to assign all external functions that act as event handlers. So, the following example:

```
<input id = "clickable" type = "button" value = "Click me and you will see"
onclick = "alert ('Thanks for clicking');" />
```

It can be transformed into:

```
// External function function sample Message () {
alert ('Thanks for clicking');
}
// Assign the external function to the document.getElementById ("clickable")
element. Onclick = sample Message;
// XHTML element
```

```
<input id = "clickable" type = "button" value = "Click and see" />
```

The technique of semantic handlers consists of:

1. Assign a unique identifier to the XHTML element using the id attribute.
2. Create a JavaScript function responsible for handling the event.
3. Assign the external function to the corresponding event in the desired element.

The last step is the key to this technique. First, you get the element to which you want to associate the external function: `document.getElementById ("clickable");`

Next, a property of the element with the same name as the event to be handled is used. In this case, the property is `onclick`: `document.getElementById ("clickable"). Onclick = ...`

Finally, the external function is assigned by its name without parentheses. The most important thing (and the most common cause of errors) is to indicate only the name of the function, that is, dispense with the parentheses when assigning the function: `document.getElementById ("clickable"). Onclick = sample Message;`

If the parentheses are added after the name of the function, the function is actually running and saving the value returned by the function in the element `onclick` property.

```
// Assign an external function to an event of a document.getElementById
("pinchable") element. Onclick = sample Message;
// Execute a function and save its result in a property of a
document.getElementById ("pinchable") element. Onclick = sampleMessage
();
```

The great advantage of this method is that the resulting XHTML code is very "clean" since it does not mix with the JavaScript code. In addition, within the assigned external functions, the variable `this` can be used to refer to the element that causes the event.

The only drawback of this method is that the page must be fully loaded before the DOM functions assigned by the handlers to the XHTML elements can be used. One of the easiest ways to ensure that certain code is to be executed after the page is fully loaded is to use the `onload` event:

```
window.onload = function () {
document.getElementById ("clickable"). Onclick = sample Message ; }
```

The prior art uses the concept of anonymous functions, which is not going to be studied, but which allows to create a compact and very simple code. To ensure that a JavaScript code is to be executed after the page has been fully loaded, you only need to include those instructions between the `{` and `}` symbols:

```
window.onload = function () {...
}
```

In the following example, you add events to elements of type `input = text` of a complex form:

```
function highlights () {
// JavaScript code}
window.onload = function () {
var form = document.getElementById ("form"); var fieldsInput =
form.getElementsByTagName ("input");
for (var i = 0; i <fieldsInput.length; i ++) {
if (fieldsInput [i] .type == "text") {fieldsInput [i] .onclick = highlights;
}
```



```
}  
}
```

Obtaining Event Information (Event Object)

Normally, event handlers require additional information to process their tasks. If a function, for example, is responsible for processing the onclick event, you may need to know what position the mouse was at the time of clicking the button.

However, the most common case in which it is necessary to know additional information about the event is that of the events associated with the keyboard. Normally, it is very important to know the key that has been pressed, for example, to differentiate the normal keys from the special keys (ENTER, tab, Alt, Ctrl., Etc.).

JavaScript allows you to obtain information about the mouse and keyboard using a special object called event. Unfortunately, different browsers have very notable differences in the treatment of information about events.

The main difference lies in the way in which the event object is obtained. Internet Explorer considers that this object is part of the window object and the rest of browsers consider it as the only argument that the event handling functions have.

Although it is a behavior that is very strange at first, all modern browsers except Internet Explorer magically and automatically create an argument that is passed to the managing function, so it is not necessary to include it in the call to the managing function. Thus, to use this "magic argument", it is only necessary to assign it a name, as browsers automatically create it.

In summary, in Internet Explorer type browsers, the event object is obtained directly by: `var event = window.event;`

On the other hand, in the rest of browsers, the event object is obtained magically from the argument that the browser automatically creates:

```
function handlerEvents (theEvent) {  
var event = theEvent;  
}
```

If you want to program an application that works correctly in all browsers, it

is necessary to obtain the event object correctly according to each browser. The following code shows the correct way to obtain the event object in any browser:

```
function handlerEvents (elEvento) {  
var event = elEvento || window.event; }
```

Once the event object is obtained, all the information related to the event can be accessed, which depends on the type of event produced.

Information about the event

The type property indicates the type of event produced, which is useful when the same function is used to handle several events: `var type = event.type;`

The type property returns the type of event produced, which is equal to the name of the event but without the prefix `on`.

Using this property, the previous example in which a section of contents was highlighted when you hover the mouse over.

Information about keyboard events

Of all the events available in JavaScript, keyboard-related events are the most incompatible between different browsers and therefore, the most difficult to handle. First, there are many differences between browsers, keyboards, and user operating systems, mainly due to differences between languages.

In addition, there are three different events for the keystrokes (`onkeyup`, `onkeypress`, and `onkeydown`). Finally, there are two types of keys: normal keys (such as letters, numbers and normal symbols) and special keys (such as `ENTER`, `Alt`, `Shift`, etc.)

When a user presses a normal key, three events occur in a row and in this order: `onkeydown`, `onkeypress` and `onkeyup`. The `onkeydown` event corresponds to the fact of pressing a key and not releasing it; the `onkeypress` event is the key press itself and the `onkeyup` event refers to the release of a key that was pressed.

The easiest way to obtain information about the key that has been pressed is through the `onkeypress` event. The information provided by the `onkeydown` and `onkeyup` events can be considered as more technical since they return the internal code of each key and not the character that has been pressed.

Below is a list with all the different properties of all keyboard events in both Internet Explorer and other browsers:

Keydown event:

- Same behavior in all browsers:
- KeyCode property: internal code of the key
- CharCode property: not defined

Keypress event:

Internet Explorer:

- keyCode property: the character code of the key that was pressed
- charCode property: not defined

Other browsers:

- keyCode property: for normal keys, not defined. For special keys, the internal code of the key.
- charCode property: for normal keys, the character code of the key that was pressed. For special keys, 0.

Keyup event:

- Same behavior in all browsers:
- KeyCode property: internal code of the key
- charCode property: not defined

To convert the code of a character (not to be confused with the internal code) when character representing the key that was pressed, the `String.fromCharCode ()` function is used.

When you press the a key in the Firefox browser, the following sequence of events is displayed:

Event type: keydown KeyCode

property: 65

charCode property: 0

Character pressed:?

Event type: keypress

KeyCode property: 0

charCode property: 97

Character pressed: a

Event type: keyup KeyCode

property : 65

charCode property: 0 Character pressed:?

Pressing the A key (the same key, but having previously activated the capital letters) shows the following sequence of events in the Firefox browser:

Event type: keydown KeyCode

property: 65

charCode property: 0 Character pressed:?

Event type: keypress

KeyCode property: 0

charCode property: 65

Character pressed: A

Event type: keyup KeyCode

property : 65

charCode property: 0 Character pressed:?

In the keydown and keyup events, the keyCode property is still valid in both cases. The reason is that keyCode stores the internal code of the key, so if the same key is pressed, the same code is obtained, regardless of the fact that the same key can produce different characters (for example, upper and lower case).

In the keypress event, the value of the charCode property varies, since character a is not the same as character A. In this case, the value of charCode matches the ASCII code of the pressed character.

Following the Firefox browser, if a special key is pressed, such as the tab, the following information is displayed:

```
-----  
Event type: keydown KeyCode  
property: 9  
charCode property: 0 Character pressed:?
```

```
-----  
Event type: keypress  
KeyCode property: 9  
charCode property: 0  
Character pressed:?
```

```
-----  
Event type: keyup KeyCode  
property: 9  
charCode property: 0 Character pressed:?
```

The special keys do not have the charCode property, since only the internal code of the key pressed in the keyCode property is saved, in this case, code 9. If the Enter key is pressed, code 13 is obtained, the key the upper arrow produces code 38, etc. However, depending on the keyboard used to press the keys and depending on the arrangement of the keys depending on the language of the keyboard, these codes may vary.

The result of the execution of the same example above in the Internet Explorer browser is shown below. Pressing the a key, the following information is obtained:

```
-----  
Event type: keydown KeyCode  
property: 65
```

charCode property: undefined Character pressed:

Event type: keypress

KeyCode property: 97

charCode property: undefined Character pressed:

Event type: keyup KeyCode

property: 65

charCode property: undefined Character pressed:

The keyCode property in the keypress event contains the ASCII code of the key character, so the character can be obtained directly using `String.fromCharCode(keyCode)`.

If the A key is pressed, the information shown is identical to the previous one, except that the code that shows the keypress event changes to 65, which is the ASCII code of the A key:

Event type: keydown KeyCode

property: 65

charCode property: undefined Character pressed:

Event type: keypress

KeyCode property: 65

charCode property: undefined Character pressed:

Event type: keyup KeyCode

property: 65

charCode property: undefined Character pressed :

When you press a special key like the tab, Internet Explorer displays the following information:

Event type: keydown KeyCode

property: 9

charCode property: undefined Character pressed:

The codes shown for the special keys match those of Firefox and other browsers but remember that they may vary depending on the keyboard that is used and in the function of the arrangement of the keys for each language.

Finally, the altKey, ctrlKey and shiftKey properties store a Boolean value that indicates whether any of those keys were pressed when the keyboard event occurred. Surprisingly, these three properties work the same way in all browsers:

```
if (event.altKey) {  
  alert ('The ALT key was pressed'); }  
}
```

Below is the case in which the Shift key is pressed and without releasing it, you press on the key that contains the number 2 (in this case, it refers to the key that is at the top of the keyboard and by therefore, it does not refer to the one found on the numeric keypad). Both Internet Explorer and Firefox show the same sequence of events:

Event type: keydown KeyCode

property: 16

charCode property: 0 Character pressed:?

Event type: keydown KeyCode

property: 50

charCode property: 0 Character pressed:?

Event type: keypress

KeyCode property: 0

charCode property: 34

Character pressed: "

Event type: keyup KeyCode

property : 50

charCode property: 0 Character pressed:?

Type of event: keyup

Property keyCode: 16

Property charCode: 0 Character pressed:?

The keypress event is the only one that allows to obtain the really pressed character, since when pressing on key 2 having previously pressed the Shift key, the character is obtained ", which is precisely the one that shows the keypress event.

The following JavaScript code allows you to correctly obtain in any browser the character corresponding to the key pressed:

```
function handler (elEvento) {  
    var event = elEvento || window.event; var character = event.charCode ||  
    event.keyCode; alert ("The clicked character is:" + String.fromCharCode  
    (character));  
} document.onkeypress = handler;
```

Information about mouse events

The most relevant information about mouse-related events is the coordinates of the mouse pointer position. Although the origin of the coordinates is always in the upper left corner, the point taken as a reference of the coordinates may vary.

In this way, it is possible to obtain the position of the mouse with respect to the computer screen, with respect to the browser window and with respect to the HTML page itself (which is used when the user has scrolled over the page). The simplest coordinates are those that refer to the position of the pointer with respect to the browser window, which are obtained through the clientX and clientY properties:


```
function showsInformation (the Event) {  
var event = the Event || window.event; var coordinateX = event.clientX; var  
coordinateY = event.clientY; alert ("You clicked on the position:" +  
coordinate X + "," + coordinate Y);  
} document.onclick = sampleInformation;
```

The coordinates of the position of the mouse pointer with respect to the full screen of the user's computer are obtained in the same way, using the screenX and screenY properties:

```
var coordinateX = event.screenX; var coordinateY = event.screenY;
```

In many cases, it is necessary to obtain another pair of different coordinates: those corresponding to the position of the mouse with respect to the origin of the page. These coordinates do not always coincide with the coordinates regarding the origin of the browser window since the user can scroll over the web page. Internet Explorer does not provide these coordinates directly, while other browsers do. In this way, it is necessary to detect if the browser is Internet Explorer type and if so, perform a simple calculation.

The ie variable is true if the browser in which the script is run is of type Internet Explorer (any version) and false otherwise. For the rest of the browsers, the coordinates regarding the origin of the page are obtained using the pageX and pageY properties. In the case of Internet Explorer, they are obtained by adding the position with respect to the browser window (clientX, clientY) and the page scrolling (document.body.scrollLeft, document.body.scrollTop).

Chapter 20 Arrays in JavaScript

What is an array?

An array is a collection of items. It is basically a JavaScript object that is used to store an ordered collection of items. All the items can be accessed by using a single name which is the array name. Unlike Java, arrays in JavaScript can have non-homogenous data stored in them, i.e., items in the array can be of different data types.

An array is one of the most useful entities in any programming language; a programming language is incomplete without an array. Often we find the need to have a list of items, and it is easier to do so with arrays, instead of having a separate variable for each item, you can refer to the whole list with just one variable if you use arrays.

Initializing arrays in JavaScript

There are two ways of initializing an array in JavaScript which is listed below.

- Method 1:

This is the most commonly used method to create an array, in this method array is declared with the help of square brackets [], where the items of the array are placed between these square brackets separated by commas.

Example:

```
var myArray = []; //initialzing an empty array  
var myArray2 = [ 'this', 'is', 'an', 'array'];
```

- Method 2:

This method involves initialization of array with the help of 'new' keyword which is used to initialize an object of Array 'class'.

Example:

```
var myArray = new Array(); //initialzing an empty array  
var myArray2 = new Array('this', 'is', 'an', 'array');
```

Accessing elements of array

Array's elements or items are accessed by referring to their index number

which is the position of element or item in the array. The index number in JavaScript starts from 0, not from 1, so index number of the first element will be 0 and that of the second one will be 1 and so on.

Elements can be accessed by typing array name followed by the index number of the desired element enclosed within square brackets [].

Example:

```
var myArray = [ 'this', 'is', 'an', 'array'];
```

```
console.log( myArray.0 ); // WRONG SYNTAX, array's elements are not object's properties
```

```
console.log( myArray ); // CORRECT SYNTAX, prints 'this' on the console screen
```

```
console.log( myArray ); //CORRECT SYNTAX,prints 'an' on the console screen
```

Unlike objects, in arrays, elements cannot be referred to by string indexes. But arrays can store objects as one of its elements which can have string index.

Example:

```
var myArray = [ 'this', 'is', { 'an' : 'example' }];
```

```
console.log( myArray.an );// WRONG SYNTAX
```

```
console.log( myArray.an );//CORRECT SYNTAX, prints 'example' on the console screen
```

Nested array

It is possible to have nested arrays, i.e., arrays within arrays, in JavaScript. You just have to declare an array, as you normally do, as an element of another array.

Example:

```
var myArray = [ 'an array', ['nested', 'array']];
```

```
console.log( myArray ); //prints 'an array' on the console screen
```

```
console.log( myArray ); //prints 'nested' on the console screen
```

Modifying and adding elements to array

You can directly assign values to array elements by referring them with their index number, the same way as you do to access them.

Example:

```
var myArray = [ 'this', 'is', 'an', 'array'];  
console.log( myArray ); // prints 'an' on the console screen  
myArray = 'change';
```

```
console.log( myArray ); // prints 'change' on the console screen
```

You can add values to array in the exact manner as you assign values to array elements, if that particular index is bigger than array's size then it gets created automatically.

Example:

```
var myArray = [ 'this', 'is', 'an', 'array'];  
myArray = 'change'; // index 12 doesn't exist but gets created when we  
assign value to this index
```

```
console.log( myArray ); // prints 'change' on the console screen
```

You can also add elements to the array by using `.push()` method about which you will read later in this chapter.

Size or length of an array

You can get the size/length of an array by using the 'length' property of the array object. As the index starts from 0, the valid indexes for an array of length 'n' is from 0 to (n - 1);

Example:

```
var myArray = [ 'this', 'is', 'an', 'array', 'example'];  
console.log( myArray.length ); //prints '5' on the console screen
```

```
console.log( myArray[ myArray.length - 1 ] ); //prints 'example' on the  
console screen
```

When you add a new value to an array by using an index number that did not exist previously, the array length is updated accordingly.

Example:

```
var myArray = [ 'this', 'is', 'an', 'array', 'example'];  
console.log( myArray.length ); //prints '5' on the console screen  
myArray = 'change';
```

```
console.log( myArray.length ); //prints '13' on the console screen
```

Looping through array

Looping through an array in JavaScript is rather simple, there are many approaches to do so, some of the most common and easy ways to do so are discussed below.

- Using array's length:

This method is simple and straightforward; it involves getting an array's length using 'length' property of the array object and then looping from 0 to the obtained length.

Example:

```
var myArray = [ 'this', 'is', 'an', 'array', 'example'];  
var len = myArray.length;  
// notice that we use '<' operator below not '<=' here  
for( var i = 0; i < len; i++) {  
  console.log( myArray[i] );  
}
```

//above example prints each element of the array in separate line on the console screen

- Using .forEach() method:

Example:

```
var myArray = [ 'this', 'is', 'an', 'array', 'example'];  
myArray.forEach(function(item, index, array) {  
  console.log( index, item );  
});
```

//above example prints index along with the corresponding element of the array in separate line on the console screen

- for-of method:

This method involves usage of the 'of' keyword in for loop.

Example usage:

```
var myArray = [ 'this', 'is', 'an', 'array', 'example'];  
for (var item of myArray) {  
  console.log(item );  
}
```

//above example prints each element of the array in separate line along with other properties of the object on the console screen

Some commonly used array object methods

- .pop()

This method removes the last element from the array and returns it.

Example usage:

```
var myArray = [ 'this', 'is', 'an', 'array', 'example'];  
console.log( myArray.length );//prints '5' on the console screen  
console.log( myArray.pop() );// prints 'example' on the console screen  
console.log( myArray.length );//prints '4' on the console screen, as the last  
element has been removed using .pop() method, so now length becomes 4
```

- .push(item1, item2 ... itemN)

This method adds element(s) to the end of the array. The element(s) to be added are passed as an argument of this array.

Example:

```
var myArray = [ 'this', 'is', 'an', 'array', 'example'];  
console.log( myArray.length );//prints '5' on the console screen  
myArray.push('more', 'elements');//adding 2 new elements at the end of the
```

array

`console.log(myArray.length);`//prints '7' on the console screen, as two new elements have been added

- `.shift()`

Same as `.pop()` method but instead of removing the last element of the array, this method removes the first element of the array and returns it.

Example:

```
var myArray = [ 'this', 'is', 'an', 'array', 'example' ];
```

```
console.log( myArray.length );
```

//prints '5' on the console screen

```
console.log( myArray.shift() );
```

// prints 'this' on the console screen

```
console.log( myArray.length );
```

//prints '4' on the console screen, as the first element has been removed using `.shift()` method so now length becomes 4

- `.unshift(item1, item2 ... itemN)`

Same as `.pop()` method but instead of adding element(s) to the end of the array, this method adds the element(s) at the start of the array.

Example usage:

```
var myArray = [ 'this', 'is', 'an', 'array', 'example' ];
```

```
console.log( myArray.length );
```

//prints '5' on the console screen

```
myArray.unshift('more', 'elements');
```

//adding 2 new elements at the end of the array

```
console.log( myArray );
```

//prints 'more' on the console screen

```
console.log( myArray.length );
```

//prints '7' on the console screen, as two new elements have been added

- `.join(separator)`

This method returns a string consisting of the elements of the array with the argument passed as the separator between them. The separator argument is optional if it is not passed then the separator is taken as a comma.

Example:

```
var myArray = [ 'this', 'is', 'an', 'array', 'example'];
console.log( myArray.join('-') );
//prints "this-is-an-array-example" on the console screen
```

- `.slice(startIndex, endIndex):`

This method extracts the part of the array starting from `startIndex` and ending at `(endIndex - 1)` and returns it as a new array; the original array stays unaffected. Both of the arguments are optional, if not specified then whole array copy is returned.

Example:

```
var myArray = [ 'this', 'is', 'an', 'array', 'example'];
var newArray = myArray.slice(1, 3);
console.log( newArray ); // prints ['is', 'an'] on the console screen
```

- `.indexOf(searchItem)`

This method returns the first index of the element which is being searched for based on the argument passed, returns -1 if no such element was found.

```
var myArray = [ 'this', 'is', 'an', 'array', 'an', 'example'];
console.log( myArray.indexOf('an') );//prints '2' on the console screen
console.log(myArray.indexOf('random') );//prints '-1' as no such element is there in the array
```

- `.lastIndexOf(searchItem)`

This method is same of `.indexOf()` method but instead of returning the first index of the searched element it returns the last index of it, if a match was found, else it returns -1.

```
var myArray = [ 'this', 'is', 'an', 'array', 'an', 'example'];
console.log( myArray.lastIndexOf('an') );//prints '4' on the console screen
console.log(myArray.indexOf('random') );//prints '-1' as no such element is there in the array
```

Arrays are very useful. So learn the array concepts very well. I hope this chapter was helpful.

Chapter 21 Values, Types, and Operators

In the world of computers, there is data. You can create new data, read data and change data, which are all stored as a look-alike long succession of bits. Define bits as zeros and ones that take a strong or weak signal, and high or low electrical charge from inside the computer. All data and pieces of information are described as a succession of zeros and ones and represented in bits.

Values

Take a deep breath and think of an ocean of bits. The latest PCs contains more than 30 billion bits in its data storage. We use bits to create values. The computer can function correctly because every bit of information is split into values. Every value consists of a type that influences its role, and values can be numbers, text or functions, etc. To generate value, you need to invoke its name, and it appears from where it was stored.

Arithmetic

Arithmetic is the major thing to do with numbers. The multiplication, addition, and subtraction of more than one number to produce another number is an arithmetic operation. This is an example of what they look like in JavaScript:

```
100 + 4 * 11
```

The + and * symbols are called operators, the first means addition while the other means multiplication. An operator inserted between two values will produce another value. The - operator is for subtraction and the / operator is for the division. If operators show together without parentheses, the precedence of the operators decides the way they are applied. If several operators with the same precedence show right next to each other like 1 - 2 + 1, apply them left to right: (1 - 2) + 1.

Special numbers

JavaScript consists of three unique values that do not act like numbers but are regarded as numbers. Infinity and -Infinity are the first two, which mean the positive and negative infinities, and the last value is the NaN. NaN says “not a number,” although it is a value of the number type.

Strings

A string is the succession of numbers. The string is the next data type, and they represent text. Strings confine their content in quotes.

```
`Down the walkway path. `
```

```
"On top of the roof."
```

```
'I am at home.'
```

Quotes are used in different types like the double quotes, single quotes, or the backticks to mark strings. It is essential that the strings match. The elements within the quotes create a string value by JavaScript. JavaScript uses the Unicode standard to assign a number to every character needed, including Arabic, Armenian, Japanese, etc. You cannot subtract, multiply, or divide strings but you can use the + operator, which will not add but concatenates two strings together. Concatenation means to glue strings together.

Unary operators

Symbols do not represent all the operators. You can write some operators in words. A clear example is a type of operator, and this operator creates a string value with the name of the type of its attached value.

```
console.log(typeof 4.5)
```

```
// → number
```

```
console.log(typeof "x")
```

```
// → string
```

The second displayed operator is called the binary operator because they use two values, while operators that use one value are the unary operator.

```
console.log(- (10 - 2))
```

```
// → -8
```

Boolean values

Boolean is a value that differs only between two possibilities such as “on” and “off” and so on. The Boolean consists of only true and false.

Comparison

Comparison is a way to create Boolean values:

```
console.log(3 > 2)
```

```
// → true
```

```
console.log(3 < 2)
```

```
// → false
```

The > and < characters are the signs that represent “is greater than” and “is less than,” accordingly. You can use binary operators in a Boolean value that determines if the contained value is true or false.

You can compare strings in the same manner.

```
console.log("Aardvark" < "Zoroaster")
```

```
// → true
```

You can instruct strings in alphabetical order, uppercase letters are often “less” than lowercase, so "Z" < "a," and non-alphabetic characters (! -, and so on) are also present in the ordering. When JavaScript wants to compare strings, it recognizes characters from left to right, differentiating the Unicode codes individually.

Here are other related operators <= (less than or equal to), >= (greater than or equal to), == (equal to), and != (not equal to).

```
console.log("Itchy" != "Scratchy")
```

```
// → true
```

```
console.log("Apple" == "Orange")
```

```
// → false
```

In JavaScript, there is only one value that is not equal to itself, which is the NaN (“not a number”).

```
console.log(NaN == NaN)
```

```
// → false
```

Logical operator

JavaScript endorses only three operators that you can apply to Boolean values, they are and, or, and not. The && operator signifies logical and and its results depend on where the values imputed are true or false.

```
console.log(true && false)
```

```
// → false
```

```
console.log(true && true)
```

```
// → true
```

The `||` operator indicates logical or. This operator outputs true if the given value is true.

```
console.log(false || true)
```

```
// → true
```

```
console.log(false || false)
```

```
// → false
```

An exclamation mark (!) indicates Not. It is an operator that overturns the set value, and it changes from true to false and false to true.

The `||` consists of the lowest precedence of all operators, then `&&`, the comparison operators (`>`, `==`, etc.), and so on. The example below states that parentheses are necessary:

```
1 + 1 == 2 && 10 * 10 > 50
```

Empty values

Null and undefined are the only two types of special values that are used to indicate the non-appearance of an important value. They contain zero data.

Automatic type conversion

Earlier I said that JavaScript accepts practically all given programs, including programs with odd behaviors. Automatic type conversion illustrates in the following expressions:

```
console.log(8 * null)
```

```
// → 0
```

```
console.log("5" - 1)
```

```
// → 4
```

```
console.log("5" + 1)
```

```
// → 51
```

```
console.log("five" * 2)
```

```
// → NaN
```

```
console.log(false == 0)
```

```
// → true
```

When you assign an operator the wrong value, JavaScript silently returns that value to the exact type it requires using the type coercion rule. In the first expression, the null turns to 0, and the 5 in the second remains 5 (from string to number). In the third expression, there was a string concatenation before the numeric addition, which converts the 1 to 1 (from number to a string). When odd numbers such as "five" or undefined changes to the number, it gets the value of NaN. If you want to differentiate between values of the same type using ==, the output should be true if the values are similar except NaN. If you want to test if a value contains a real value, use the == (or! =) operator to compare it. To avert unexpected type conversions, use the three-character comparison operators.

Short-circuiting of logical operators

The && and || are called the logical operators. They are used to hold several types of values in a specific way. They change the values contained in the left side to Boolean type to decide, although it depends on the operators and the type of generated result, but will always reinstate the left or right-hand value. The || operator sends back value to the left when it can be changed to true and will reinstate the value to the right.

```
console.log(null || "user")
```

```
// → user
```

```
console.log("Kate" || "user")
```

```
// → Kate
```

This function is used to return values to its default value and placed within an empty value as a replacement. Strings and numbers to Boolean value conversion rules indicate that 0, NaN, and empty string (") count as false while other values are true. Therefore 0 || -1 outputs -1, and "" || "!" yields "!?". The && operator operates identically but the other way around. When values to the left can be changed to false, return the value, or it sends the value to the right. Both operators evaluate the value to the right only when it is required. For example, we have the following values set as true || X, the value of X will be true and will not consider it. The same rule applies to the false && X, which the x is false and will overlook it. You can call this process the short-circuit evaluation.

Summary

This chapter looks at the four types of JavaScript values, which are strings, numbers, undefined values and Booleans. These values are developed by inserting their names as true, null or value (13, "ABC"). Operators can integrate and change values. We looked at binary operators for arithmetic (+, -, *, /, and %), string concatenation (+), comparison (==, !=, ===, !==, <, >, <=, >=), and logic (&&, ||), and also various unary operators (- to nullify a number, ! to nullify logically, and type of to search for a value's type) and a ternary operator (?:) to choose one of two values depending on a third value. You will get sufficient information to use JavaScript like a small calculator, and you will improve in the following chapters.

Exercise

Write a JavaScript practice to build a variable through a user-defined name.

Solution

Chapter 22 Definition of Arrays in JavaScript

Another thing that we need to take a look at when we work in this language is what an array is all about. To keep it simple, an array is just going to be a structure of data that is able to contain our group of elements. In most coding languages including JavaScript, we will find that these elements are going to be the same type of data, such as a string or an integer. Arrays are going to be used in order to help a computer program, or your coding language, organize data so that a set of values that are related is able to be sorted through or searched through well.

No matter what kind of coding language you decide to work with, you will find that keeping the data and the objects organized is going to be important. If you don't take care of these objects and types of data, they are going to float around the work that you are doing, and you will run into some troubles in the code. The part of the code that you are in will not be able to find the objects and classes that they need and your code won't work.

But when you are working with many of the modern types of coding languages out there, including JavaScript and a few others, you are going to rely on a bit more organization than what we are going to find in some of the older languages. You need to make sure that things are organized, and that the set of values that are related are going to be easy to search through and sort through as much as you would like.

For example, when we take a look back at the arrays, we may find that a search engine is going to rely on one of these arrays to help it store some of the web pages found in one of the many searches that the user was able to perform. When this same search engine displays its results, the program is going to output one element of the array at a time. This is going to be done in several different methods. The search engine is going to either do this with a specified number of values, with the most important on top, or it will keep on doing this until all of the values that are stored up in the array that are showing up.

While the program is able to create a new variable, if it wanted, for all of the results that are found in it, this can really take up a lot of time and space in some of the codings that you are doing, and it is definitely not going to be the most efficient method that you can use here. Instead, you will find that storing all of the results (while we are still on the search engine example) in an array is going to be a more efficient method to use to pull up the results

that you want, while also managing your memory.

Now, there are going to be a few different parameters and properties that are able to come with the array that you would like to create. Being able to make this come together and work the way you want is important, and understanding what each of these properties is all about is going to be a great way to get a better understanding of what you are able to do with them. Some of the properties that come along with your array object, along with information on what all of these means, includes:

1. **Constructor:** This is going to help us get back a reference to our array function, the one that was able to create the object.
2. **Index:** This is going to be the property in our array that is going to represent the zero-based index that will match up to that part inside of our string.
3. **Input:** This is going to be a property that we are only going to see in certain arrays. When an array has been created by a regular expression and it all matches, then we are going to have this input.
4. **Length:** This is going to show us the number of elements that we are going to be able to see in our array.
5. **Prototype:** We will discuss this one more in this guidebook, but this particular property is going to allow us a way to add methods and properties to our object.

This may not make a lot of sense right now, but we are going to expand upon it a bit and see more about what we will be able to do with these arrays and the different parts that come with it. Before we move into this though, we need to take some time to talk about some of the array methods that are important to our code. There are a lot of different methods that happen with our object of an array, so let's take a look at what all of these mean.

1. **Concat:** This one is going to return to us a new array that is going to be comprised of not only this array but with the other values or arrays that we need.
2. **Every():** this one is going to return true if we find that each and every element that falls in our array is going to satisfy the

testing function that we provided.

3. `Filter()`: This one is responsible for creating a new array with all of the elements that fall into the current array for which the provided filtering function is going to give us truly as the result.
4. `forEach()`: This one is responsible for calling up a function for each of the elements that are inside of our array.
5. `indexOf()`: This one is going to return the first (or the least) index that comes with the element that is inside of our array, as long as it is equal to the value that we specify. Or it will work with -1 if nothing is found to match in it.
6. `Join()`: This one is going to join together all of the elements of an array and can help turn these into a string.
7. `lastIndexOf()`: This one is going to return the last, or the greatest, index of your element in the array as long as that element is equal to the value that we are trying to specify, or we will work with -1 if we can't find a match in the array.
8. `Map()`: This one is responsible for helping us to create a brand new array with the results of calling up a provided function of all the elements that fall into this array.
9. `Pop()`: This one is going to help us remove the last element out of our array and then will return the element that we want.
10. `Push()`: This one is nice because it is able to add in at least one, but sometimes more, elements to the end of your array and then will return to you the new length that this makes your array.
11. `Reduce`: This is going to apply the function that you want at the same time against two values of the array, going from the left side over to the right side, in order to reduce it by one value.
12. `reduceRight()`: This is going to be the same idea, but instead of going from the left side to the right side, we are going to reverse this and go from the right side to the left side.
13. `Reverse`: This is going to be responsible for reversing the order that your array elements are going to be in. The first element in the

array is going to become the last element, and then the element that was the last one will go to the beginning of the array.

14. Shift(): This one is going to remove the first element out of your array and then will give you a return of that element.

15. Slice: This one is going to extract a section of your array and will give you a return that has a brand new array in it.

16. Some(): This one is going to return true as long as you have one or more elements in the array that is able to satisfy the testing function that you provided.

17. toSource(): This one is going to help us to represent the source code that comes with our object.

18. Sort(): This one can be useful because it is going to help us to sort through all of the elements that come with our array.

19. Splice: This one is going to be there to help us to either add or remove the elements of the array.

20. toString(): This one is going to return a string that is going to represent the array and all of the elements that happen in it.

21. Unshift: This final one is going to take some time to add in one or more elements to the beginning of our array and then will let us know what the new length of this array is going to be when it is done.

As we can see here, there are a lot of possibilities that we are able to work with when it comes to handling the arrays that show up in our codes. These arrays are going to help us to hold onto some of the data that we have more efficiently and will ensure that we are able to put it all together and pull out the elements that we need and want without a lot of struggle along the way. Make sure to take a look at some of the methods and functions that are available with these arrays, especially with JavaScript, so you know how to use them for some of your own needs as well.

Conclusion

The next step is to get started with the JavaScript language and all of the neat things that we are able to do with this kind of coding language. JavaScript is a bit higher level for the coding that you want to do compared to some of the other options, which is going to allow us to have a chance to handle all of the websites and web pages that you are going to create. There isn't another language that can work as well with all of the operating systems while ensuring that we are able to get the results that we will see.

In this book, I have provided you with the basic knowledge that you will need to start your journey in programming using JavaScript. The different concepts taught here, such as functions, loops, branches, and objects will equip you with the skills that you need to create your first JavaScript project.

Also, continue practicing and taking on small projects to start improving your skills. Through the knowledge imparted in this book, coupled with practice, you will be able to work on building your own websites or coding your own projects.

In your further study, I recommend that you learn and take on advanced topics such as troubleshooting in JavaScript, explore different frameworks and libraries, and expand your knowledge in using regular expressions.