Table of Contents

Chapter 1: JavaScript

JavaScript is a client-side scripting language. It is used to improve website or webpage functionality. Also, it is used in conjunction with HTML and CSS to create responsive websites.

JavaScript is one of many client side-scripting languages (e.g., VBScript, PerlScript, Jscript, ActionScript) that exist in the web. However, it is the most popular and widely used. Due to its wide usage, it has spawned multiple frameworks. Frameworks are there to make coding easier. They also improve and enhance the functionality of languages. JavaScript frameworks will be discussed in the later parts of this book.

On the other hand, this scripting language is behind many website features and additional functions on the web. It is safe to say that compared to other languages, JavaScript is the most used in the world. With almost every website employing this programming language in every page it has, without a doubt, its absence will make the Internet boring.

Aside from that, JavaScript is cross platform and it is a useful tool in creating impromptu programs and macros. Due to the mass availability of browsers in every computer and smart devices, you can easily create a decent program with the use of a text-editing program and a web browser.

What’s the Console

The console is a great tool that can allow you to test codes and familiarize yourself with JavaScript. Unlike with creating an HTML page from scratch, saving it on a file, and then testing it on the browser, you can just type the code on the console, press Enter, and it will be integrated or injected to the file immediately.

Some web developers call this method Live Scripting or interactive coding. Lately, it has become apparent that interactive coding is a faster and more efficient way of teaching programming to newbies.

Aside from avoiding the hassle of coding, saving, compiling, and testing, people can just type the line of code they want and it will processed right away. Also, most interactive console or programs provide additional feedback, which provides assistance to developers and learners.

What's Next?

Well, open your Google Chrome's console, and play with it. For starters, use it as a calculator. For example, type:

- 43 + 14
- 32 * 51
- 12 / 3
- 12419 – 4512

Of course, you do not need to use the numbers indicated exactly. Try to type any expression, formula, or equation that you can think of. Whenever you type an equation, press the Enter key. The console will provide you with a result. For example:

> 1 + 1

< 2

> _

Note: In the example codes for the developer's console, the underscore is just there to represent the cursor in the console. Also, the greater than sign signifies user input while the less than sign signifies feedback from the console and/or output.

You might have typed the = sign or wonder why you do not need to add it on the equation. There is a reason for that. The equal (=) sign or symbol plays a different role in JavaScript or in almost every programming language.

Now that you get the hang of fiddling with the developer console, the lessons on how to program or create scripts in JavaScript will begin.

Note:

If ever you encounter an unfamiliar term that was mentioned in a section of this book, try to analyze its meaning through its context. Move forward; it will be surely discussed.

Chapter 2: Variables, Identifiers, and Statements

Programming and scripting are technically alike. Usually, the term programming is used in creating programs that are compiled into an executable binary file, which is unreadable to humans. In layman's term, the lines of code you will write will be converted to machine language that your computer can easily understand.

On the other hand, the term scripting is used in creating programs that are translated into machine code when needed. Unlike programs, scripts are uncompiled code; meaning, you can easily edit or read them whenever you like, even if it is being executed.

In JavaScript, will be scripting. When you write your code and it is executed, it will be left as is. The browser reading your code will be the one who will handle the translation for the computer. Basically, you will just write the script, and then just let it run on the browser as soon as you are finished.

Writing Code

Writing a script is not that different from writing a script for a play. When writing a script for a play, you will put instructions for your actors in paper. In writing a script for a web, you will write instructions for all objects and elements involve in the web page. Those objects are your browser, the page itself, and HTML elements, among others.

And of course, the biggest thing that separates writing a script for a play and writing a script for a website is the language. Most probably, the language you will use for the play is English, a language that you are already familiar with. For the web, you will use JavaScript, a new language you have no idea on how to speak or write with.

When learning a language, you often start with the basic parts of sentences. In English, that would be the noun, pronoun, verb, adjective, etc.. In JavaScript, you will start learning the basic parts of programming and the language itself including variables, assignment operators, expressions, and keywords.

So, do not fret. It will not be as hard as you think. Take a deep breath and proceed to the next topic.

Variables

In the previous chapter, the equal sign was mentioned and used. It seems that you are just playing some Math on the console, right? Now, the next lesson will be about variables. With a decent introduction to scripting and coding, you will surely have an easier time understanding variables.

Variables are storage entities within a program. They store information that will be used later on the program. It makes tracking of useful and crucial values in a script easier. Also, it makes it easier to use a value repeatedly, without remembering and typing the value exactly every time you need it.

In the English language, variables are like pronouns. For example, your noun is “John”. To make it easier to refer to him, you can just use the pronoun he, him, or his in your sentences. That is just how simple variable works. Of course, other uses can be incorporated with variables.

On the other hand, you must be already familiar with term variable. After all, your Mathematics teacher should have mentioned it a lot when you were in high school; you would have encountered it in college as well.

The concept of variables in Math is similar with JavaScript variables or variables in programming — with a few notable differences. First, variables in programming do not only hold numbers. It can also hold or store text, arrays, and objects (more to that later).

You can create variables in JavaScript by using the var keyword or command. Type these lines in your console for example:

```
> var x;
```

Once you press enter, the console will reply with this:

```
< undefined
> _
```

At this point, variable x already exists. However, it is undefined. Google Chrome or your browser does not know yet what it is supposed to be. Will it be a number? Will it be a text? To define it further and store a value in it, you must use the assignment operator (=).

The assignment operator works like this:

> x = 2

< 2

> _

Once you press enter, the console will confirm that the number 2 has been assigned to variable x. To double check if it did, type this:

> x

< 2

> _

Note: To check the value of variables in the developer console, you can just type in the variable's name, and the console will reply with the variable's current value. Writing that on a script you will insert in your page will do nothing; take note of that.

When you call on variable x, the console will respond with its value. What would happen if you call on another variable?

> y

Uncaught ReferenceError: y is not defined (…)

> _

Note: The x in the second line represents a red x icon that signifies an error. It is not related to variable x.

Since y was not declared in the first place, the console returned a reference error. The identifier, text, or string y must be declared first with the var keyword in order to become recognized and "reference-able" by the console.

What will happen if you try to assign a value to an undeclared variable or identifier? Try this out:

> z = 3

What happened? Yes, the console accepted the assignment even if the var keyword was

not used.

< 3

> _

As convenient as it may seem, you should not create and assign variables like this. This would open a can of worms once your code becomes bigger. Primarily, the main problem is that the scope of the variables declared this way would become messed up (more about scopes later).

Also, with the previous revisions of ECMAScript (ECMAScript 5 & 6), “use strict” has become available. This mode will enforce you to always declare variables to prevent errors.

As for now, stick with the usage of var. It is the proper way —unless you know the effects of not using var and you intend it to work that way.

Anyway, by declaring variable x, your web page and JavaScript codes can use it. Of course, declaring a variable is not enough to make it useful. You must assign a value to it as mentioned a while ago. Since it is typical for new (or even seasoned) developer to forget to use var, try to include it together when you assign a value to a variable for the first time. For example:

> var x = 2

< 2

> _

Identifiers

You have seen this term a while ago. An identifier is a combination of letters, numbers, or underscores that provides ‘names’ to variables, functions, methods, properties, objects, etcetera. To put it simply, identifiers are names and they make the management of all the elements in a program easier for programmers and developers.

To be technical, when you input this, var x, you are actually creating a storage allocation in your computer’s memory. That storage allocation is called a variable. Then you will name with the identifier x for you to easily remember that variable.

Without identifiers, you will need to get the address of that storage allocation or variable in your computer’s memory. And remembering memory addresses is not something you would want to do because they are confusing bunch composed of seemingly random letters and numbers.

For example, instead of remembering an address like #25 Hudson Street, Vaughan, Ontario, Canada, it will be much easier for you to remember the exact location by giving that place a name. If that is your friend George’s house address, then it will be easier for you to recall the location as George’s house. The same goes with identifiers and variables (and other program elements).

Case Sensitivity

JavaScript is a case-sensitive programming language (you will see this line a few times after this). What does it mean? It means that a difference in letter casing can result into errors. Also, it means that different letter cased identifiers will be treated as different entities in the program.

For example, variable x will be treated as a different entity from variable X. Due to that, make sure that you always mind how your spell and type your identifiers. Error caused by wrong letter cases can be a pain to find and fix, especially if you have hundreds lines of code.

Statements

At this point, you have already created valid statements. What are statements? Statements are lines of codes that perform a specific task or tasks that do not contain any error on them. However, in some cases, erroneous lines of codes are still referred to as statements in some cases.

In English, it can be considered that statements are synonymous with sentences. Statements in programming are just like sentences in English. To create a proper script for a play or story, you must know how to write comprehensible and proper sentences. The same goes with scripting.

How can you know if your statement is valid? First, if it follows the right syntax rules, it will be considered as a proper or valid statement, which will be executed by the browser or computer without any error. Syntax rules are just like the English grammar rules. More about JavaScript syntax rules will be discussed later.

One of the statements that you have typed on the console is var x. A simple declaration of a variable is already a logical and correct statement. It performs its specific goal, which is to establish variable x.

However, do not that some lines that you create in the developer console are not valid statements in JavaScript. Even if the console does not return any error when you input those lines, some of them will not work in actual code in web pages.

For example, you have played with some mathematical equations before in the console, right? Those equations are not valid statements. They are only available in the console as a learning tool — as mentioned a while ago.

Recap and Additional Information

Generally, you will be scripting with JavaScript. Your script can also be called source code — although, source code is usually referred to the content of your HTML document.

Your script will be composed by computer/browser instructions called statements. To make sure that your browser will do what you want, you must make sure that you create valid statements.

On the other hand, variable is one of many parts of a statement. Variables can contain almost any value that you and your browser can provide by using assignment operators.

To manage and remember your variables easily, you assign them identifiers, which is just a technical term for your variable's name. Identifiers can also name functions, which will be discussed later.

And most importantly, identifiers are case sensitive. For example, the identifier GOOD is considered a different identifier with the identifier good.

Chapter 3: JavaScript Basic Syntax

JavaScript is a programming language. And just like languages, JavaScript has its own grammar rules named syntax. Going with this analogy, statements are JavaScript sentences. And order for it to be understandable, it must follow the language's syntax or grammar.

One of the syntax rules in JavaScript is that every statement must end with a semicolon (;) or a statement separator (some call JavaScript's semicolon as a statement terminator, but it has a slight difference from separators).

The semicolon's job is much like a period or semicolon's function in the English language. It separates statements. It allows the browser to distinguish where a statement ends and where another statement starts. For example:

```
> var x;
< undefined
> var y;
< undefined
> _
```

The semicolon separates statements, right? So, if that is the case, can you put two statements in one line? Try it:

```
> var x; var y;
< undefined
> x;
< undefined
> y;
< undefined
> _
```

It worked. What will happen if you removed the semicolon?

```
> var x var y
    Uncaught SyntaxError: unexpected token var (…)
```

As you can see, you received an error. After all, the console was expecting a semicolon, a line break, or other things, but not a keyword.

Wait. That is confusing, right? How come the earlier examples did not require you to use semicolons, yet they worked anyway? Here is the answer:

Unlike most programming languages, JavaScript is not strict when it comes to statement separators. Just adding a linebreak alone is enough for the parser to understand that your statement is finished, and you are ready to add another statement after it.

Also, when you run a JavaScript code without semicolons, the browser will automatically generate them for you. Despite being okay with the absence of semicolons, they are still needed to produce readable and clean code.

Even if your code will run without semicolons, it is advisable that you use them. It enforces clean and readable codes. Also, omitting these statement separators may lead to unexpected issues.

In the future releases of JavaScript/ECMAScript, it is possible that JavaScript will become strict with separators.

Case Sensitivity

Another thing that you must remember is that JavaScript is a case sensitive programming language. Aside from being case sensitive when it comes to identifiers, JavaScript is also case sensitive to other parts of your script as well, especially in keywords. For example, the keyword var is different from VAR.

```
> var x;
< undefined
> VAR x;
    Uncaught SyntaxError: unexpected identifier (…)
```

In here, since VAR is not the var keyword, the browser thought that VAR was an identifier for a variable or function. And unfortunately, unlike the keyword var, the identifier VAR does not exist since you did not declare or assign any value to it. Due to that, the console did not expect the identifier VAR in the code.

Keywords

The term keyword has been mentioned time and time again in the previous section. Keywords are reserved words or identifier in a program. They can be either pre-existing variables, methods, functions, or objects in the program.

The keyword var is a perfect example. The keyword var is a built-in function or command in JavaScript to allow you to declare and create variables. Just like any keywords or reserve words, you cannot use them as identifiers. For example:

```
> var var;
    Uncaught SyntaxError: unexpected token var (…)
> _
```

When using the keyword var, the parser will expect that the next word or entity after the keyword will be an identifier. Since var is a keyword and not an available identifier, the console returned an error.

Of course, the same thing will happen if you do this:

```
> var = 2;
    Uncaught SyntaxError: unexpected token = (…)
```

Just like before, the console did not expect the assignment operator to be there. Hence the syntax error appeared.

Comments

Before you proceed on placing your script on your webpage, you must know about JavaScript comments or commenting. You might be already familiar with comments, thanks to your HTML background.

JavaScript comments work like the usual HTML comments. However, there are some differences.

The main difference is that the syntax in creating comments in JavaScript starts with two forward slashes (//) or a forward slash and an asterisk (/*) and ends with an asterisk and a forward slash (*/).

For those who are unfamiliar with comments, comments are lines of codes that are ignored by the parser or the browser. Any lines of codes or comments that are included in the comment will not be processed and the error on those lines will not be brought up.

For example:

> // Test

< undefined

> _

In the browser's developer console, you might notice that it replies an undefined. Do not worry; it is still unprocessed. Here is another example:

> // var testVariable = 2;

< undefined

> testVariable

Uncaught ReferenceError: testVariable is not defined (…)

Despite responding to the comment line, the code inside the comment was not processed. Hence, the browser returned an error when the code tried to call or reference testVariable. On the other hand, even if you fill the comment with erroneous code or anything, no error will be returned.

Function of Comments in Scripting or Programming

Comments serve an essential function in script or program development. Primarily, they are used for documentation. In case you are going to share your script to other people, providing them with inline documentation of what your script does, description of the author, and licensing information will be beneficial to you and them.

Here is an example comment block in a popular JavaScript library, jQuery:

```
/*!
* jQuery JavaScript Library v1.11.3
* http://jquery.com/
*
* Includes Sizzle.js
* http://sizzlejs.com/
*
* Copyright 2005, 2014 jQuery Foundation, Inc. and other contributors
* Released under the MIT license
* http://jquery.org/license
*
* Date: 2015-04-28T16:19Z
*/
```

Aside from providing documentation and meta information, comments can be used to disable statements temporarily. Instead of deleting a statement, you can just place two forward slashes to disable it.

It is advantageous because it allows you to “undo” changes that you make in the future in your code. Also, it can be useful in debugging and finding erroneous statements in your script.

In addition, comments can be used as bookmarks within your code. With the help of the search function, you can just move around your code and find the sections that you want to view. For example:

```
<script>
```

… multiple lines of code …

// codeForTextCheck

… insert code here …

</script>

By using the search function in your editor, you can instantly go to the codeForTextCheck section. This is beneficial for people who have thousand lines of codes in their scripts.

Comments can be also used to noting information and putting reminders. If multiple people are editing the script, you will have an easier time if you use comments to give out reminders with the code you create.

Two types of comments

There are two types of comments in JavaScript. The first type is single line; the second type is multi-line.

Single line comments use the two forward slashes. They can be placed almost anywhere in your script. The browser will ignore any text after the two slashes and will resume execution on the next line.

On the other hand, multi-line comments use forward slashes and asterisks. To start a multi-line comment, put a forward slash and an asterisk (/*). All text after the slash and asterisk will be regarded as comments even if they are separated with a line break. To end the multi-line comment, an asterisk and slash must be placed (*/).

Chapter 4: Operators

When you hear of the word operators, it is highly possible that the first things you will think of are the mathematical operators addition (+), subtraction (-), multiplication (*), and division (/). Programming and JavaScript scripting also use these operators.

The biggest difference when it comes to programming operators and mathematical operators is that the former has a lot of different types. Most of the operators in Mathematics are in programming. Technically speaking, math operators are only a subset of programming operators.

So, what are the functions of operators in programming? Primarily, their main function is to manipulate data. In programs or scripts, processing data using these operators are always present. Operators are always in programs whether they were used explicitly or implicitly.

Common Types of Operators

Programming languages have a diverse set of operators.

Arithmetic

The usual mathematical operators are categorized here. And they are:

Addition	+	> var x = 1 + 1 < undefined > x < 2 > _
Subtraction	-	> var x = 1 – 1 < undefined > x < 0 > _
Multiplication	*	> var x = 2 * 3 < undefined > x < 6 > _
Division	/	> var x = 8 / 2 < undefined > x < 4 > _
Modulus	%	> var x = 12 % 5 < undefined > x < 2 > _
Increment	++	> var x = 1

		< undefined > x < 1 > x++ < 2 > _
Decrement	—	> var x = 1 < undefined > x < 1 > x— < 0 > _

You can do other mathematical operations in JavaScript. However, you will need to use the Math object and its methods, where the other math operators are included and are available (sin, round, random, pow, etcetera). You will learn more about the Math object later.

Assignment

Assignment operators are there for you to assign values to variables and other elements in your program. The most used assignment operator is the “=” operator. With it alone, you can create a program.

However, there are other assignment operators that you can use to make your script efficient and your life easier.

Add and Assign	+=	> var x = 1 < undefined > x < 1 > x += 1 < 2 > _
Subtract and Assign	-=	> var x = 1 < undefined > x < 1 > x -= 1 < 0 > _
Multiply and Assign	*=	> var x = 2 < undefined > x < 2 > x *= 3 < 6 > _

Divide and Assign	/=	> var x = 8 < undefined > x < 8 > x /= 4 < 2 > _
Modulus and Assign	%=	> var x = 12 < undefined > x < 12 > x %= 1 < 2 > _

Technically, they are combinations of arithmetic and the assignment operator. When you use them, what happen is that the variable on the left side is treated to be present on the right side of the operator. For example:

```
> var x = 2
< undefined
> x
< 2
> x += 3
< 5
> _
```

In the example, x+= 3 is being treated like x = x + 3. At the beginning, you will not be using too much of these operators. However, in big projects, they can become handy. These operators can shorten the time you need to type, and improves the readability of your code.

On a different note, there are other combinations of these operators. Technically, almost all other operators can be shorthanded this way.

String Concatenation Operators

You can also use some operators in strings. There are two primary string operators in JavaScript. Concatenate and concatenate and assign. Here are some examples on how to use them:

> var stringX = “This is a string”;

< undefined

> var stringY = “, and you can add or concatenate them using the + operator.”

< undefined

> var stringZ = stringX + stringY

< undefined

> stringZ

< “This is a string, and you can add or concatenate them using the + operator.”

> _

You can also do the example using the concatenate and assign. For example:

> var stringX = “This is a string”;

< undefined

> var stringY = “, and you can add or concatenate them using the + operator.”

< undefined

> stringX += stringY

< “This is a string, and you can add or concatenate them using the + operator.”

> _

Of course, the usage of the “+” symbol might result to a confusion and curiosity. Can you add numbers and strings? Well, in a way, you can. However, when you do, the result will be a string, and the number will be only concatenated. For example:

> var stringExample = “This is a string.”

< undefined

> var x = 2

< undefined

> var y

<undefined

> y = stringExample + x

< “This is a string.2”

> _

Logical and Comparison Operators

In later lessons, you will deal with conditional statements. And when conditional statements are involved, you will need to use logical and comparison operators. If you had a logic class, then you will be familiar with them.

Logical and comparison operators deal only with two possible results: true and false. In different terms, using these two types of operators will let you perform Boolean operations. For you to easily understand what this is all about, in case you are confused, then proceed on checking the table below.

Greater Than	>	> 1 > 2 < false > 2 > 1 < true > _
Less Than	<	> 1 < 2 < true > 2 < 1 < false > _
Equal To	==	> 1 == 2 < false > 1 == 1 < true > _
Not Equal To	!=	> 1 != 2 < true > 1 != 1 < false > _
Greater Than or Equal To	>=	> 1 >= 2

		< false > 2 >= 1 < true > 2 >= 2 < true > _
Less Than or Equal To	<=	> 1 <= 2 < true > 2 <= 1 < false > 2 <= 2 < true > _
Equal Value and Type	===	> 1 === 1 < true > 1 === “1” < false > _
Not Equal Value and Type	!==	> 1 !== 1 < false > 1 !== “1” < true > _

Chapter 5: Data Type

In programming, manipulation of data is imminent. What is data anyway? Data is information. It can be in form of numbers or text. Aside from text and numbers, JavaScript can also process other forms of data types such as arrays and objects.

Why is there a classification of data? It is to make the language and functions as clear and clean as possible. Also, it compartmentalizes the usage of operators. For example, there is no way you can multiple texts:

> “addend1” * “addend2”

< NaN

> _

You get a NaN result. NaN stands for Not a Number. In other programming languages, you will receive an error instead. On the other hand, doing operations on different data types can be tricky. For example:

> “big number” – 888

< NaN

> _

Here is another example:

> “one” + 1

< “one1”

> _

Not knowing the data types that you are handling can make your program perform outside your expectations. However, unlike other programming languages, JavaScript is more forgiving.

Dynamic Data Typing

JavaScript Is a Strong Typing Language. It is also considered as a loosely type language. It is also referred to as duck typing language.

It seems that using var is a pain, right? To be honest, the usage of var is convenient. Back then, you must specify a variable's data type. For example, when creating an integer variable in C, you will need to declare it as:

```
int variableExample;
```

If you fail to provide the right type or declare the variable, you will encounter errors in your program. For example:

```
#include <stdio.h>

int main(void) {
        void dd;
    dd = "This is a string";
    printf(dd);
    return 0;
}
```

Error: variable or field 'dd' declared void

That is an example declaration of variable in C, an old and powerful programming language. Since variable dd is a void data type, you cannot assign a different data type on it. If you do that in JavaScript, there will be no problem. For example:

```
> var x = null
< undefined
> x
< null
> x = "This is a string."
< "This is a string."
```

> x

< “This is a string.”

> _

In technical terms, all variables declared in JavaScript can be considered variants. Variant is a data type that can accept different data types. The type of the variable will depend on the data it contains regardless of declarations or first assigned values.

However, the term variant type is not totally applicable to JavaScript. The term is usually used on languages such as Visual Basic and C++.To be precise, it is much better to call it dynamically typed.

On the other hand, do note that this is just the tip of the iceberg. Technically speaking, these variables behave like objects. And these data types can be correctly called as object classes. More about this will be discussed on the chapter about object-oriented programming.

Number

Numbers are one of the data types in JavaScript. Unlike other programming languages, JavaScript only has two types of number literals — Integers and float. Integers are whole numbers that do not have decimal values. Float includes rational and real numbers, with decimal values.

Strings

Strings, in easy to understand term, are text data. In a program or script, you need to use and edit multiple texts. In technical terms, strings are values enclosed in quotation marks. For example, “dog” and “cat” are strings. Even if the quotation marks do not contain any character, it will still be considered as a string.

You can assign strings to variables. For example:

> var sampleString = “This is a sample string.”;

< undefined

> alert(sampleString);

> _

The code will make a message box containing the string, “This is a sample string.”, without the quotation marks appear on the web page.

Adding Quotation Marks on Strings

What will happen if you decide to place quotation marks on your string? Try this example:

> var sampleString = “Quotation marks look like these ””.”

Uncaught SyntaxError: Unexpected string (…)

> _

Unfortunately, adding quotation marks within a string can be problematic for you and the parser. In the example, the parser thought that the string assignment has already ended at the second appearance of the quotation mark. Here is how the parser perceived the example.

var sampleString = “Quotation marks look like these “

“.”

The parser was expecting a statement separator (;) since it thought that the assignment was already finished. And since the “previously thought valid statement” was followed by a quoted dot, the parser returned a SyntaxError due to the unexpected appearance of another string data.

There are multiple ways to work around this issue. First of all, aside from double quotes (“), strings can be contained using single quotes (‘). For example:

> var sampleString = ‘This will still work.’;

< undefined

> alert(sampleString);

> _

How will that help you? Well, if you started a string data with a single quote, the string must end with a single quote, too. The same goes with double quotes. Because of that, if you insert double quotes inside a string that started with a single quote, the double quotes will not terminate or signal the end of the string data. For example:

> var anotherSampleString = ‘The double quotes ”” will not make this line return an

error.’;

< undefined

> alert(anotherSampleString);

> _

If you tried that example, you will notice that, aside from not returning an error, the popup box displayed the double quotations. Alternatively, the single quotations were not displayed since they were used to contain the string.

Smart Quotes or Curly Quotes

Please take note that smart quotes or curly quotes are different from the usual single or double quotation marks. In most word processing programs like Microsoft Word, regular single or double quotes are converted to smart quotes or curly quotes automatically — for aesthetic purposes.

For example, if you press ‘ in your keyboard, Microsoft word will change that to ‘(or ” if a’ is present before the cursor). The same goes with “, which will be changed to “ (or ” if a “ is present before the cursor).

Unfortunately, the character code for these smart quotes is different from the character codes of the regular single and double quotes. Because of that, compilers and parsers will return an error if you use those quotes in place of the regular quotes.

For example:

> var example = “This is a fancy text with fancy quotes.”

Uncaught SyntaxError: Unexpected token ILLEGAL (…)

> _

To fix this, you can just disable the feature in your word processor or perform an undo action to revert the smart quote to a regular quote if you are coding your scripts or programs using word processors or copying code from document files. Another method is to use your word or text processors Search and Replace function.

Primitive Values

If a certain data or value is typed as is, it is usually referred to a primitive value. It is important to know this concept, especially with an object oriented programming language like JavaScript.

As for now, just remember that almost everything in JavaScript is considered as objects. A few of the exceptions are primitive values. Unlike objects, primitive values do not have inherent properties and methods (although strings do have some).

Aside from numbers, other primitive values are data types of their own exist. And they are true, false, undefined, and null. They also are keywords, which have special meanings. At this point, you must already have an idea on what true and false stand for.

undefined and null

As for undefined, it is a value given to a variable that has been declared, but was not assigned a value yet. On the other hand, null is a primitive value that symbolizes the absence of value. Also, null is provided to objects that are not existing in the document. Both undefined and null can be assigned to variables.

Arrays

Array is a concept that aspiring programmers with loose mathematical foundation may find difficult to understand. In a few simple words, an array is a collection of values. In JavaScript, an array is a variable that can contain multiple separate values. For example:

```
var exampleArray = [“Google”, “Yahoo!”, “Bing”];
```

You can access these values from the array using indices. For example:

```
var exampleArray = [“Google”, “Yahoo!”, “Bing”];
alert(exampleArray[0]);
alert(exampleArray[1]);
alert(exampleArray[2]);
```

Array Index

By the way, indices are automatically generated once you add values to the array that you created. By default, indices always start with 0. Also, the indices are assigned according to the order that they were assigned. Google received an index of 0 because it was the first value assigned, and Yahoo! was assigned an index of 1 because it was the second value that was assigned.

However, you can opt to assign a specific index of your own in an element that you will insert in your array. For example:

```
var exampleArray;
exampleArray[0] = “Google”;
exampleArray[1] = “Yahoo!”;
exampleArray[2] = “Bing”;
```

By the way, assigning and accessing values in array can be done like this. The syntax for this is: array[index]. On the other hand, unlike other programming languages, JavaScript do not support using keys or named indices.

Array Values Data Types

Arrays are not restricted in containing strings alone. It can also contain objects and numbers. In addition, you are allowed to place any type of data within it. For example:

var randomArray = [“This is a string”, 1234, document];

Chapter 6: Inserting JavaScript Code

Up to this point, you are now familiar with the basics of JavaScript. Now, you will need to learn how to integrate your script in your web page. Also, starting at this chapter, you will need to use your text editor.

The subsequent chapters will be teaching you on how to take advantage of code blocks such as functions, conditional statements, and loop blocks. Of course, you can still use the developer console.

Anyway, proceed into learning on how to integrate your future scripts in your web pages.

Where to Place JavaScript

Primarily, JavaScript scripts are placed inside web pages' HTML codes. It is common that scripts are inserted inside script tags. There are other ways to insert your JavaScript code within your page, of course.

Within the Page

In case your script is too short, you can just insert it inside the script tag in your document. Usually, the script tag is placed inside the head tag. However, script tags can be placed inside your html's body or even at the end of the document.

Note: The location of the script in the page can change your script's behaviour.

Within Your Server as a Separate File

If your script is too long, you can place your script inside a .js file within your server. If you save it in a separate file, you will not need to put script tags on it, by the way.

To integrate that script into your web page, you can simply link it in your script tag by indicating its location in the src attribute. For example:

```
<!DOCTYPE html>
<html>
<head>
    <title> A Sample Web Page </title>
    <script src = “js/sampleScript.js” > </script>
</head>
<body>
    Insert Page Content Here
</body>
</html>
```

Of course, just like with the previous method, you are free to place the script in any part of your HTML document.

The main advantages of using this method are that your web page document will not be too cluttered with scripts and you can easily reuse or reintegrate your script into multiple pages without doing some nitty-gritty work of copy pasting your code in each web page that needs your script.

Also, you can update the script without worrying that you might mess up your HTML codes in your pages. And if ever you become accustomed in using server-side scripting to generate dynamic content, putting your script apart from your content will be a huge convenience.

Within Content Delivery Networks (CDNs) as a Separate File

If you are worried about your website’s performance and page load issues, then placing your JavaScript files in a Content Delivery Network (also called Content Distribution

Network) can help you reduce latency in your website, especially if you are receiving a lot of traffic in the web.

Content Delivery Networks are servers or computers with a sole purpose of providing commonly accessed content and files for websites. Usually, they are sub-domain sites that often have a domain prefix of cdn.

Aside from that, most JavaScript frameworks are best used with CDNs. Fortunately, big companies like Google cooperate with some of the most popular frameworks in JavaScript (i.e., jQuery).

Sample Web Page File

For now, forget about practicing JavaScript codes within your page. Start with a simple web page file just like the one below.

Code Sample: 1 (HTML TEMPLATE)

```
<!DOCTYPE html>
<html>
<head>
	<title>JavaScript Practice File</title>
</head>
<body>

	<h1>JavaScript Practice File</h1>
	<p>This is an example paragraph in your web page.</p>

</body>
</html>
```

Copy this code in your notepad and save it as an .htm file in your computer. You do not need to upload this web page copy yet on the Internet. You can just open it on Google Chrome and your JavaScript codes will work — just like with regular HTML creation.

By the way, for easier navigation and referencing, all example codes within this book will be accessible through the table of contents.

Sample JavaScript Usage

You’re going to start to learn some practical use of JavaScript. And for starters, you will learn how to insert popup boxes in your webpage. Copy this example:

Code Sample: 2 (WINDOWS.ALERT)

```
<!DOCTYPE html>
<html>
<head>
        <title>JavaScript Practice File</title>
    <script>
            window.alert(“Hello World!”);
    </script>
</head>
<body>

        <h1>JavaScript Practice File</h1>
        <p>This is an example paragraph in your web page.</p>

</body>
</html>
```

When you save this file and open it on your browser, it will load the page and a popup box that says, “Hello World!” will appear. This popup box is usually called as message box, popup box, or alert window by web developers. For clarity in this book, it will be called as alert window.

Chapter 7: Code Blocks, Functions, and Scope

If sentences are like statements, what are paragraphs like? Yes, in programming, you might also have to create some sort of paragraph. They are called code blocks.

Code blocks are multiple statements that are grouped together. They are not necessarily a term most developers use, but for easier understanding of the next topics, groups of statements will be called code blocks. Also, please do not confuse this with the IDE Code Blocks.

In most cases, you will need to separate certain statements and group them according to the goal that you want to achieve. In programming, it is usually done by creating code blocks, and making them functions.

Functions

Functions are grouped lines of codes that do not immediately run when the browser parses them. Unlike regular lines of codes and conditional statements, the statements within a function are executed when the function is invoked.

They are especially useful in decluttering your source code and making it more readable. Also, its primary use is to organize your code and minimize the lines of codes you need to type, especially if you need to execute the code block multiple times. After all, functions can be invoked as many times as you want.

Here is an example of a JavaScript function:

```
> function exampleFunction() {
    alert("This is a function.");
}
< undefined
> _
```

Note: If you are going to type this on the Developer Console, you can press Shift + Enter to create a line break without submitting or entering the code in the console.

Even if you put this block of code in the topmost level of JavaScript script, it will not run. The browser will just record its existence, and it will be executed when an invocation of this function occur.

Note: Unlike variables, you cannot change the code block or value within the function at runtime through your code (although with the use of DOM, it can be overwritten). However, it is possible to change a function's statements through the Developer Console — variables included.

To invoke a function, you must call it by "mentioning its identifier." For example:

```
> exampleFunction();
< undefined
```

```
> _
```

Once invoked, all the statements within the function will be executed by the browser. In the example's case, the alert box will appear.

Also, do note that functions can also invoke other functions within their code blocks. Another thing worth mentioning is that a function can invoke itself; however, you will encounter a maximum call stack size error. That error occurs since the browser will be infinitely looping again and again the invocation process, in simple terms. For example:

```
> function supahLoop() {
    supahLoop();
}
< undefined
> supahLoop()
        Uncaught RangeError: Maximum call stack size exceeded (…)
> _
```

Note: When invoking a function, make sure that you include the parentheses. Without them, some browsers may not invoke the function. In Developer's Console, submitting the function's name and not including the parentheses will only result to the console showing the contents of the function.

Scope

Scope is the accessibility of your variables. Depending on the location where you declared them or how you use them, their scopes changes.

For example, a variable that was declared inside a function will not be available to codes outside functions. Here is what this is all about:

```
> function exampleFunction() {
    var x = 2;
}
< undefined
> exampleFunction()
< undefined
> alert(x)
        Uncaught ReferenceError: x is not defined (…)
> _
```

In this case, it seems that writing a function, putting a variable declaration inside it, and invoking the function will make the variable within it available outside the code block. However, due to scoping, it will not work.

Variables declared inside a function will only be available inside that function. For example:

```
> function exampleFunction() {
    var x = 2;
    alert(x);
}
< undefined
> _
```

When you write the previous sequence of statements, the alert box will be executed and the value of variable x will appear on it.

What will happen if the variable was declared outside the function? Can the function use it? Here is an example:

```
> var x = 23
< undefined
> x
< ‘23’
> function exampleFunction() {
    alert(x);
}
< undefined
> exampleFunction()
< undefined
> _
```

Global and Local Variables

Primarily, there are two main scopes in programming. JavaScript also adheres to that. The first one is global; the second one is local.

A variable has a global scope if it declared on the topmost level of your script; meaning, it was not declared inside a function or code block. Global variables are accessible everywhere in the script.

A variable that is declared inside a function will be considered a local variable. That local variable will be only available on the function that declared it.

However, not all variables used inside functions are considered local variables. If the variable was not declared, but was assigned a value, it will have a global scope. Nevertheless, it is best not to create global variables this way since it can be confusing, especially if you are writing the code together with another individual.

On the other hand, the lifespan of a global variable ends when the page closes. The lifespan of a local variable ends when the function that declares it finishes execution.

There is another way to create a global variable, and that is to use the window object, which will be discussed in the later chapters.

Note: It is preferred by developers to use or create as few global variables as possible. Mainly, usage of too many global variables can affect the performance of your script. Usage of disposable local variables is better to optimize the usage of memory resource of your computer. Use global variables if there is only a need.

Arguments and Parameters

If local variables can only be accessed by the function that declared it, how can other functions access them? Usually, some developers just take advantage of creating a global variable to fix that in a very convenient way. However, if you like to have optimized and well-performing scripts, you will want to use arguments and parameters in your function.

Parameters are local variables in a script that can receive arguments or values when invoked. For example:

```
> function simpleMultiplication(multiplier1, multiplier2) {
    var product = multiplier1 * multiplier2;
    alert(product);
}
< undefined
> simpleMultiplication(25, 5);
< undefined
> _
```

In this example, two parameters were created. Those are multiplier1 and multiplier2. Parameters are like local variables. The differences between them are parameters are declared together with the function's identifier and they can receive arguments or values when the function is invoked.

In the example, the function simpleMultiplication was invoked and the invocation provided two arguments, 25 and 5. So, how can parameters and arguments solve the problem with scopes? Check this example:

```
> function simpleMultiplication(multiplier1, multiplier2) {
    var product = multiplier1 * multiplier2;
    showProduct(product);
}
< undefined
> function showProduct(number) {
    alert(number);
```

```
}
< undefined
> simpleMultiplication(25, 5);
< undefined
> _
```

When this example is executed, the variable product will be provided to the showProduct() function, which effectively passes the value of a local variable to another function.

Some new developers who are not familiar with arguments and parameters do this by using global variables. For example:

```
> var product
< undefined
> var multiplier1
< undefined
> var multiplier2
< undefined
> function simpleMultiplication() {
    product = multiplier1 * multiplier2;
    showProduct();
}
< undefined
> function showProduct() {
    alert(product);
}
< undefined
> multiplier1 = 25
< 25
> multiplier2 = 5
< 5
> simpleMultiplication();
```

< undefined

> _

With a quick glance, you can easily tell that using parameters and arguments over global variables is a much better, cleaner, and simpler way to allow functions to “communicate” and pass through values.

Function return

What would you do if you want the function to pass on a value? In the previous example, what would be a better way to shorten the code? The answer to those questions is the use of the return keyword and statement.

To make it simpler to pass on local variables, you can also take advantage of return statements. Return statements are there to let a statement that invokes a function to receive a value in “return”. For example:

```
> function simpleMultiplication(multiplier1, multiplier2) {
    return multiplier1 * multiplier2;
}
< undefined
> simpleMultiplication(25, 5);
< 125
> alert(simpleMultiplication(25, 5));
< undefined
> _
```

As you might have noticed in the console, when you invoke the function this time, the console provides the answer for the function instead of the usual undefined. Receiving the answer (or the statement or the value of the expression that is indicated in the return statement) means that the function returns a value.

By the way, alert is also considered a function, a built-in function in JavaScript to be precise. However, unlike the new version of the simpleMultiplication example function, it does not return a value.

Anyway, when the alert function is called, it will provide a message box that contains the answer or the return value of the function. Simple, right?

Recap

In JavaScript, you can group statements into code blocks by placing them inside curly braces. Most developers do not do this since they usually take advantage of comments instead to organize their code blocks.

On the other hand, providing your code blocks with identifiers together with the function keyword makes them functions. The main difference between functions and code blocks is that functions will not be executed as long as they are not called or invoked.

When it comes to scope, there are two variable scopes in JavaScript: global and local. Declare a variable outside of functions, and they will have global scope. Declare variables in functions, and they will have local scope.

Passing through variable values in and out of functions can be eased by using parameters, arguments, and return statements. Do remember that it is highly discouraged to use global variables, especially if your script is large due to performance issues.

Chapter 8: Conditionals

To make your program think or grant it with dynamisms, you need to take advantage of conditional operations. For example, if you want to make your script do something else if a certain variable obtain a specific value, then a conditional statement is in order.

Conditional Statement If

The most basic conditional statement in JavaScript, and almost every other programming languages out there, is the if conditional statement. The if conditional statement works in a simple manner. You indicate a condition. When that condition is met, the code block within the conditional statement if will be executed. For example:

```
var x = 2;
x = x + 3;
if(x == 5) {
    alert("Adding x, which has the value of 2, with 3 equates to 5.");
}
```

In the example, variable x was been assigned a value of 2. After that, 3 was added to it. The next statement was a conditional if statement. In the statement, x == 5 is a condition. In human language, it means that x is equal to 5. Below the if statement, an alert statement is placed.

If you read the whole conditional if statement, it will be, "if variable x is equal to 5, then execute alert("Adding x, which has the value of 2, with 3 equates to 5.")". Since x is equal to 5, then the alert box will appear.

By the way, do note that the assignment operator (=) is different from the 'equal to' operator (==). The former is for assigning values while the latter is for comparing values.

So, what will happen if the condition in the conditional statement is not met? For example:

```
var x = 2;
x = x + 3;
if(x == 6) {
    alert("Adding x, which has the value of 2, with 3 equates to 5.");
}
alert("It will not work.");
```

When you run that script, the alert statement will not be executed. Everything inside the curly braces of the if conditional statement will be ignored, and the next statement after

the if block will be executed instead. In this case, the alert(“It will not work.”) statement will be the only thing that will be executed.

What if you want to add a different code block to execute in case the previous if conditional statement did not trigger? You have two ways to do that. The first one is to use another if statement. For example:

```
var x = 2;
x = x + 3;
if(x == 6) {
    alert(“Adding x, which has the value of 2, with 3 does not equate to 6.”);
}
if(x == 5) {
        alert(“Adding x, which has the value of 2, with 3 equates to 5.”);
}
```

The next way will be discussed in the next section.

If Else Statement

Another efficient way to let your script do something if the previous if statement did not trigger is to use the else conditional statement. An else conditional statement is similar to if. The main difference is that the condition of the else statement depends on the previous condition of the if statement that is written before it. For example:

```
var iLoveYou = 0;
if(iLoveYou == 0) {
    alert("Fat chance. I do not love you, too.");
}
else {
        alert("Gosh, I did not know that you love me.");
}
```

In this case, in case that the value of the variable iLoveYou is set to 0, the statements inside if will be triggered, and the else statement will be ignored. In case that the variable takes in a number other than 0, the statements within the if conditional statement will be bypassed, and the statements within the else conditional statement will be executed.

To make it easier for you to understand the else keyword works, then check this alternative version of the code using if statements instead.

```
var iLoveYou = 0;
if(iLoveYou == 0) {
    alert("Fat chance. I do not love you, too.");
}
if (iLoveYou != 0) {
        alert("Gosh, I did not know that you love me.");
}
```

In this version, the else statement is equivalent to the reverse of the condition in the previous if. In the first if statement, the condition read as, "If the variable iLoveYou is equals to 0, the say Fat Chance.". In the second if statement, the condition reads as, "If the variable iLoveYou is not equal to 0, then say Gosh." In other words, the else and second if

statements means that, “run the statements if the value of variable iLoveYou is other than the numerical integer 0.”

Why use else? You must use it because it is easier, and you do not need to think of other conditions. The computer will immediately assume that you want to do something in case that the previous statement is not satisfied. Instead of thinking of the obverse version of the previous if statement’s condition, the computer will provide it instead.

Else If Statement

To get more control of the conditions you created, you can use else if statements. Else if statements are else statements combined with another condition. You can use it in conjunction with other ifs and else statements. For example:

```
var emailDomain = “HotMail”;
if(emailDomain == “GMail”) {
    alert(“You are using a Google account.”);
}
else if(emailDomain == “YMail”) {
    alert(“You are using a Yahoo! account.”);
}
else if(emailDomain == “HotMail”) {
    alert(“You are using a HotMail account.”);
}
else {
    alert(“I don’t know your email provider.”);
}
```

Switch Conditional Statement

In case you will be dealing with multiple possible values, using if, else, and else if can be a bit messy. Fortunately, you can use switch statements instead if you will only base your conditions in one variable or expression, and you will need to provide statements for each possible value. For example:

```
var emailDomain = “HotMail”;
switch(emailDomain) {
    case “GMail”:
            alert(“You are using a Google account.”);
                break;
    case “YMail”:
            alert(“You are using a Yahoo! account.”);
    case “HotMail”:
            alert(“You are using a Hot Mail account.”);
    default:
            alert(“I don’t know your email provider.”);
}
```

Case Keyword

The case keyword is used to denote that value that you want to compare with the expression or variable that you placed on the switch statement. In case that the value together with the case statement is equal to the value in the switch statement, then the code block in it will be executed.

Break Keyword

The break keyword is used to prevent the browser to execute the next statements, and escape the switch statements. If not placed, the browser will execute all the statements within the switch statements until it sees a break keyword or the end of the switch block.

Default Keyword

The default keyword is the “else” statement of a switch statement. In case no case values are found to satisfy the value in the switch condition, then the code in the default keyword will be executed instead.

Chapter 9: Loops

Aside from allowing your script to have statements to be executed depending on the condition of the script, you can also insert loops in JavaScript. Loops are code blocks that allow the browser to repeatedly execute statements. You can specify how many times the loop will reexecute the statements. Or you can just indicate a condition that will tell the loop when to stop looping.

For Loop

The for loop is the most basic and common loop that you can use in JavaScript. With the for loop, you can easily create a loop block that will repeat according to the number that you will indicate. For example:

```
for (i = 1; i < 100; i++) {
    alert(“This message will popup for 100 times.”)
}
```

The for loop condition has three parts. The first is the declaration of variables that you will need in the loop condition or the code block. It will be executed once before the loop begins. In this case, the variable declared is i. And it was assigned a value of 1.

The next part of the condition is the condition itself. As long as the condition returns True, the loop will continue working. When the condition returns False, that is the time the for loop will stop. The condition will be checked before a loop is executed.

In this case, the condition is i < 10. As long as the variable i’s value stays less than 10, the loop will continue. On the other hand, once it becomes equal to 10 or greater than 10, then the loop will stop.

The last part is the step. This statement will be executed after every loop. In the example, the step part is i++. The statement is i incremented using the increment operator. In every loop, 1 is added to the value of i. Because of that, the value of variable i is changed in every loop.

Eternal Loops

Eternal loop is a condition wherein your loop will just keep on looping until the page, your browser, or your computer is closed or crashed. In most cases, eternal loops happen by accident and carelessness. Unless intended, eternal loops can be bad since it can slow down your page, browser, or computer. Depending on the number and complexty of the statements within a loop, your computer may crash or experience a massive slowdown due to wasted computer resources (RAM and CPU) by the eternal loop.

Eternal loop happens if you fail to insert a condition that will make your loop stop. For example:

```
for(i = 1;i > 0; i++) {
    i++;
}
```

This example will loop forever because the condition placed on the for statement will never return false. As the example code goes, the variable I will never have a value equal to 0 or less than 0. Although, if this example was executed in a modern day computer, the impact would be not enough to crash a computer or browser. Nevertheless, slow down can be experienced.

Of course, the best way to fix this eternal loop is to fix the condition that was placed. Another method is to place a conditional statement together with the break keyword. For example:

```
for(i = 1;i > 0; i++) {
    if(i == 100) {break;}
}
```

While Loop

Another loop that you can use in JavaScript is the while loop. The while loop is like an advanced version of the for loop that encourages users to customize their loop. Unlike for loop, the while loop only requires a condition to run. For example:

```
var i = 1;
while(i < 100) {
    alert(“This message will popup for 10 times.”);
    i++
}
```

This is the while loop version of the first for loop example provided earlier. As you can see, the declaration statement for variable i was placed outside the loop itself and the step increment is placed within the loop.

Of course, for new programmers, using the while loop will expose them to higher probability of creating eternal loops.

Do While Loop

The good thing with while loop is its versatility. And to let you have full control on how your loop behaves, you can use the do keyword together with while. To use a do while loop, check out this example:

```
do {
        alert("This message will popup for 10 times.");
        i++;
}
while(i < 100);
```

The main difference of the do while loop in all the previous loop variants is that the condition will be only processed once a loop is finished. Meaning, it will be executed regardless of the condition. It will only check the condition after the first loop, and if the condition returns True, the loop will be activated once more. IT will only stop until the while condition returns False.

Chapter 10: Events

The things that drive responsive webpages are user interactions with the website. Whenever a user clicks on the button, something happens. Whenever a user scrolls the page, something happens. Those things happen due to the JavaScript.

But how can you make those scripts work when the user does something to your page? Well, you need to indicate them in your script. And capture those events.

In this section, you will be taught of how you can take advantage of functions, simple JavaScript code snippets, and incorporating them on HTML events. For example:

<button onClick = “alert(‘This is a popup box.’)” >Click me please!</button>

Try inserting that on your sample web page. Load your sample web page, and then click the button element that you inserted. Of course, as the sample code implies, the page will launch a popup box containing the message, “This is a popup box.”

In layman’s terms, that HTML line makes your browser put a button element that executes a simple alert code whenever you click on it or fire the onClick event. onClick is just one of many events that you can use and integrate on your HTML elements.

By indicating these events in the elements, you will be able to make your pages and elements “respond” to almost all actions that your users will do. A few of events that you can capture are onLoad, onChange, and onMouseOver.

Syntax and Case

Take note: since these events must be coded on your page's body, case sensitivity does not apply to the events. For example:

<button ONCLICK = "alert('Another button clicked.')" >This is another button?</button>

When you click this button, the browser will still "fire" or "invoke" the JavaScript code that you placed on it regardless of the case of the event. However, changing the case of the code within the event can cause errors. Despite being outside of the script tags, inserted codes like these are still considered part of your JavaScript code. So, syntax rules of JavaScript are still applied on them. For example:

<button ONCLICK = "ALERT('THIS WILL NOT WORK.')" >This will not launch a message box. Clicking is futile.</button>

Event Exclusivity

Most of the events can be used on almost all elements. For example, you can place the onClick event on a paragraph element ( ).

<p onClick = ‘alert(“You clicked on me. Yay!”)’ >This is a sample paragraph.

However, certain events are exclusive to some elements. For example, the onChange event cannot be used to button (<button>) and paragraph elements ( ), but it will work on a text area element (<textarea>).

<button onChange = “alert(‘Will this work?’)” >Click me!</button>

<p onChange = “alert(‘Most probably not.’)” >Am I changeable?

<textarea onChange = “alert(‘You have changed my content, you foul beast!’)” />

Since you cannot change the content or value of the paragraph and button elements, the onChange event will not fire. On the other hand, if you do change the content of the text area element, it will launch a popup. Alternatively, if you do just click on it or revert the changes you made on it, the event will not fire.

Curious Cat

What if you try something “smart” and tried to do some “changes” on an element like button? You can actually change the content of those kinds of elements using DOM and the innerHTML property. Will the onChange event will fire if you do that? Try these lines of code:

<button id=‘x’ onChange = “alert(‘Will this work?’)” >Click me!</button>

<button onClick = “document.getElementById(‘x’).innerHTML = ‘xxx’” >Click me! </button>

Unfortunately, you cannot work around that. On the other hand, trying to emulate those events (and hope that it can save you some time) is inefficient. You can just code it instead of experimenting.

Multiple Events

Of course, to make your page more responsive, you can indicate and capture multiple events in one element. For example:

<button onClick = “alert(‘This is a popup box.’)” onMouseover = “alert(‘This is a popup box.’)” >Click me please!</button>

Chapter 11: HTML DOM

JavaScript is an object oriented programming language. In its eyes (figuratively), everything is an object (with a few exceptions like primitive values). In order to use it in conjunction with HTML, it uses HTML DOM or HTML Document Object Model.

What is the Document Object Model anyway? The HTML Document Object Model is a tree (it can be also called as a hierarchy) of all the objects in a HTML web page. It is an organized structure where elements are neatly arranged according to their respective parents and children. On top of a Document Object Model is the document object.

Back then, the document object or the HTML page (including scripts and styling code) is considered the topmost object or parent in the Document Object Model. Nowadays, due to changes in JavaScript and browsers, the document object is now a child of the window object.

Here is a simple sample HTML page.

```
<html>
<head>
    <title>Sample Web Page</title>
</head>
<body>
    <h1>Hello World!</h1>
    <p>This is a sample HTML document.</p>
</body>
</html>
```

If you visualize its Document Object Model in bullet form, it will look like this:

- document
    - head
        - title = “Simple Web Page”
    - body
        - h1 = “Hello World!”
        - p = “This is a sample HTML document.”
        -

Of course, that is just a simple representation of the Document Object Model. The important thing to note is that through the document object, you can access and manipulate all the elements in your page. You can treat the elements in your HTML as "properties" of your document object. For example, try to do this in your console:

> document.title

< "Sample Web Page"

> document.title = "New Title"

< "New Title"

> document.title

< "New Title"

> _

After doing that, you can check the tab of the document you are editing. And yes, the title of the page became "New Title". With Document Object Model, it has become easier to interact and manipulate everything in your HTML document.

It appears so simple, but what about the other elements in your page? This is where it gets interesting. True, it is easy to change and access the title element in your webpage. After all, there is only one title element in every web page. How about the other elements that are used multiple times, like <p> and ?

But before you dive deep to that, know more about the document object and the concept of parents and children first.

The Document Object

The document object is the primary parent node of HTML DOM. Since it is in the topmost part of the Document Object Model, it is often called the root node or object. All other nodes, objects, or elements are contained within or are owned by the document object. By the way, the objects within DOM are often called as nodes due to the structural design of the DOM itself.

Also, despite being the root node, the document object is under the window object. The window object is the window of the browser. Unlike the document object, some browsers do not support the window object. For now, the main focus of this chapter will be the document object since it usually receives the most interaction from you.

Document Object’s Properties and Methods

The document object has a lot of properties, methods, and child nodes that you can access. Of course, you will not need to learn all of them at once at this point. For now, you will know the most used methods and properties you need.

Navigating Through the Document Object Model

Parent and Child Concept

Anyway, the concept of parent and children nodes is simple. If element A is within element B, element A is a child of element B and element B is the parent of element A. For example:

The element <body> is a child node of the element <html> or the document object itself. On the other hand, the element <html> or the document object is the parent node of element <head> and <body>.

Chapter 12: HTML and CSS Editing Using JavaScript and DOM

Now, you are familiar with HTML DOM. You have now the power to change almost anything in your page using scripts. Aside from that, you can create simple functions that can allow you to perform checks on your forms. You can now create email text boxes that automatically checks if the email address format is correct, among other things.

The getElementById method and innerHTML Property

Just like with CSS, JavaScript and HTML DOM also have selectors. However, it is more proper to call them methods of the document object since there are other methods that are referred to selectors (CSS selectors to be precise — i.e., querySelector() and querySelectorAll()).

Primarily, the method that you will be using most of the time is the getElementById() method of the document object. This method allows you to “select” the specific element that you want to access or manipulate in your web document. For example:

```
<html>
<head>
    <title>Another Sample Page</title>
</head>
<body>
    <p id=“sampleParagraph1” >This is an example paragraph</p>
        <p id=“sampleParagraph2” >This is another example paragraph</p>
</body>
</html>
```

If you want to access the text of the paragraph element with the id sampleParagraph1, you will need to code:

```
> document.getElementById(“sampleParagraph1”).innerHTML
< “This is an example paragraph”
> _
```

By the way, to access strings that are placed between an opening tag and closing tag, you can use that object’s innerHTML property. You can also change its value by assigning another string to it. For example:

```
> document.getElementById(“sampleParagraph1”).innerHTML = “Yay. I changed it!”
< ” Yay. I changed it!”
```

> _

Of course, elements that have empty tags, such as input and br, will always return an empty or ”” innerHTML property.

What will happen if there are multiple elements that have the same id? Of course, standard practice dictates you should not create elements with same IDs; however, if you have intended or accidentally created elements like that, the first element that was parsed or read in the document will be selected. For example:

<html>

<head>

<title>Another Sample Page</title>

</head>

<body>

This is an example paragraph

This is another example paragraph

This is another example paragasdasdraph

</body>

</html>

If you try to access the element with the ID “sampleParagraph1”, this will happen:

> document.getElementById(“sampleParagraph1”).innerHTML

< “This is an example paragraph”

> _

innerHTML versus innerText versus textContent

Some of you might have been already researching about JavaScript, and you might have already encountered the innerText property. And you might question why use innerHTML to retrieve strings instead of innerText when it basically does the same thing?

Unfortunately, they do not do the same thing. First of all, innerHTML retrieves the string between the opening and closing tags. It does not perform any alteration and space trimming (removal of trailing or consecutive spaces).

On the other hand, innerText perform trimming and remove any line break, carriage return, tabs, or new line characters in the string. For example:

```
<html>
<head>
   <title>Another Sample Page</title>
</head>
<body>
   <p id="sampleParagraph1" >
         This is     an example paragraph
         </p>
</body>
</html>
```

When you access the text using innerHTML:

```
> document.getElementById("sampleParagraph1").innerHTML
< "
   This is     an example paragraph
   "
> _
```

When you access the text using innerText:

```
> document.getElementById("sampleParagraph1").innerText
< "This is an example paragraph"
> _
```

As you can see, the linebreaks and trailing spaces are removed in the innerText property while it was retained on the innerHTML property.

Another major difference between them is that innerText removes all the tags contained inside the elements — innerHTML returns everything. For example:

```
<html>
<head>
   <title>Another Sample Page</title>
</head>
<body>
   <p id=“sampleParagraph1” >
        This is an <b>example</b> paragraph
        </p>
</body>
</html>
```

When you access the text using innerHTML:

```
> document.getElementById(“sampleParagraph1”).innerHTML
< “
   This is an  <b>example</b> paragraph
   “
> _
```

When you access the text using innerText:

```
> document.getElementById(“sampleParagraph1”).innerText
< “This is an example paragraph”
> _
```

As you can see, the <b> opening and closing tags and the non-breaking space are shown in innerHTML while both were not provided in innerText.

How about textContent? The property is similar to innerHTML. There are performance related technicalities about their difference, but primarily, developer networks, such as MDN or Mozilla Developers’ Network, recommend the use of textContent instead of innerHTML. They recommend the former than the latter because it performs faster and is more secure. Even though it is more recommended, most web developers and scripters use innerHTML.

So, what's the point of learning all of these? Firstly, the innerText property is considered not a W3C standard, which means that some browsers do not support this property. One of those browsers is Mozilla Firefox. Other browsers do not have problems with the innerText property, such as Internet Explorer and Google Chrome, despite its status.

On the other hand, you should know when or where you should use these properties. For example, unlike innerHTML and textContent, innerText cannot retrieve HTML or "hidden text" placed on meta elements like script and style. Alternatively, if you want to alter the HTML content of an element with HTML code, you might want to use innerHTML or textContent instead.

JavaScript and CSS

Aside from manipulating HTML content, you can handle your page's CSS properties using JavaScript. With proper use of JavaScript and HTML DOM, you can change themes on the go, perform animations, and even create cool effects in your page.

Just like with the previous sections, the key here is HTML DOM. The concept of manipulating CSS entries can be done by accessing your elements' properties/attributes. To be frank, in changing and reading CSS properties, you will spend a lot of time tinkering with your elements' style attributes. Nevertheless, there are other ways to do that, too.

This is an example on how you can change an elements CSS properties.

<html>

<head>

<title>Sample Web Page</title>

</head>

<body>

<h1 id="header1" >This is a sample header</h1>

</body>

Example script:

> document.getElementById("header1").style.color = "yellow";

< "yellow"

> document.getElementById("header1").style.backgroundColor = "red";

< "red"

> document.getElementById("header1").style.paddingLeft = "35px";

< "35px"

> _

When manipulating CSS properties, you need to remember a few simple:

- First of all, you can change almost any CSS property of an element through its style property.

- Second, the name of the CSS property is relatively the same when you access it

through JavaScript. The main difference is that, all spaces or dashes in a property's name is removed, and on its stead is an uppercase letter of the next word. For example, changing the left padding on CSS will require you to set a value on padding-left. In JavaScript, the property name is paddingLeft.

- Third, all values that you will place must be in string form. Meaning, you will need to place quotation marks in every value you will assign to a CSS property.

Chapter 13: JavaScript Object Oriented Programming

Congratulations on reaching this part of the book. As of now, you know the basics of JavaScript. With your current knowledge, you can now provide semi-dynamic content. With a little knowledge in server-side scripting language and MySQL, you will be capable of creating fully dynamic websites with rich web content. Sounds good, right? So what is next?

The next part that you need to know is object oriented programming. Learning this programming paradigm will allow you to create advanced, complex, and large scripts with ease and clarity. With it, you will be able to harness more than half of JavaScript's capability.

With the knowledge of object oriented programming, you will be able to create web applications like games, eCommerce sites, and complex data manipulation scripts. Also, you will have familiarity with almost all popular programming languages available. Of course, this is not a promise or a guarantee. This book will only show you the way, and it is up to you if you are going to walk on it.

Programming Paradigms

In programming, goals can be achieved and programs can be created in multiple ways. For example, a calculator program may function all the same but can be coded differently. An addition operation in programming, or even in Mathematics, can be performed in different ways. One can do it by simply using the addition operator (1 + 1), or another can do it by using the subtraction operator and the unary negative operator (1 – (-1)).

Of course, despite achieving the same goal, programs with different source codes may have additional behaviours that may be controlled or uncontrolled. One of those behaviours is the tendency of a code using more resources than the other code, despite providing the same purpose as the other.

In creating larger and more complex program development, how you write your code will matter. Depending on how you write, your program may perform faster or slower. Your development time may become easier or harder. Your code might be easily read or difficult to comprehend.

And due to those concerns, broad programming models and paradigms were developed. Standard practices started to exist. And writing styles were created. On the flip side, due to the advancement in computing, those issues mentioned before have been reduced and might become too unnoticeable, especially for beginners.

These days, choosing the programming model, standard practice, and writing style has become more of a preference issue rather than performance. However, do still note that in bigger projects, as mentioned before, choosing the right model, styles, and practices can still provide a huge impact to you, your team, and your program.

Structured and Unstructured Programming

Since this is a book about JavaScript, a multi paradigm language, you will be provided with an introduction to object oriented programming and a brief explanation to procedural programming.

So what is object oriented programming anyway? As it name implies, OOP is a programming model that focuses on objects rather than just statements or actions. Before you have reached this chapter, you have been performing procedural programming, which heavily relies on functions.

Object oriented programming is considered as a structured paradigm. Structured contrasts with unstructured, a programming model that tends to make programmers create linear programs or codes that are usually run incrementally through line numbers and jump from one point to another using goto commands. The unstructured model is a primitive paradigm that often leads to messy codes, which is called spaghetti code due to its recursive nature.

The main difference of structured and unstructured programming is the usage of functions, procedures, or subroutines. Instead of just relying to goto, a flow control command, developers can take advantage of "grouping" certain statements and use them readily by invoking them. It eliminates the need for the parser to incrementally move through and execute unnecessary statements or make the programmer create conditional jumps to get back to the previous line where the initial goto was invoked. Of course, those can be easily remedied by using loops and other keywords. However, the exclusion of the usage of functions can prevent you to easily allow your program to perform repetition of grouped statements without rewriting and pointing your code to go back in a specific line number again and again.

Of course, unstructured programming was not a preferred choice; rather, it was a limitation of the programming languages back then. When functions were introduced, developers became more accustomed in procedural.

Yes, as its name implies, procedural programming is a paradigm that heavily relies on procedures or functions. Procedural programming is considered as a structural programming paradigm. It is also related to code block programming; however, for simplicity's sake, the book will only cover procedural.

Procedural Programming

As mentioned a while ago, before you have reached this chapter, you were actually doing procedural and functional programming. The main aspect that made you a procedural programmer up to this point is the usage of functions.

Aside from functions, the concept of variable scoping, is also a part of procedural programming, which can be mainly attributed to structured programming.

Do note that most programming languages do offer procedural programming. This paradigm is always considered as an entry point for every aspiring programmer, unless they tackle on OOP head on or decide to explore languages that are heavily reliant on other programming paradigm.

Object Oriented Programming

Now, before anything else, do note that JavaScript is a multi paradigm language. It means that multiple programming models or styles can be used in this client-side scripting language.

JavaScript can be considered as a procedural, event driven (this will be tackled on a later chapter since management of events is crucial in JavaScript), and object oriented programming language. Nevertheless, most people will refer it as an OOP language due to its programming structure and the usage of DOM (Document Object Model), which primarily forces you to interact with objects, hence accustoms you to do OOP.

But what exactly is object oriented programming?

Primarily, OOP introduces the use of objects in programming. Objects are like variables. However, instead of storing a singular data, it can have its own functions, called methods, and variables, called properties.

Together with objects, OOP also introduces namespaces, classes, constructors, inheritance, encapsulation, abstraction, and polymorphism.

Chapter 14: Objects

As you have seen, the code window.alert() allowed you to create a popup box in your web page. This code has four parts: window, . (the dot), alert, and () (the parentheses). The “window” part is an object. The dot is a property accessor. The alert is a method. And the parentheses are enclosures for the method’s arguments.

Those are loads of information that might confusion, but do not fret; take it one step at a time.

An object is an element that contains properties and methods. Methods are like commands or actions that the object can do. Properties are like descriptions or attributes that the object has.

A good analogy is a stove and an object. A stove has one common method, and that is to make fire or induce heat on its stovetop. On the other hand, it has properties like stove type, remaining fuel, and material.

In the previous example, the object that was featured was the window object. The window object has the methods alert(), confirm(), close(), stop(), and so forth. Alternatively, it has the properties screenX, screenY, outerHeight, outerWidth, and so forth.

JavaScript can give you access to these methods and properties. You can use the methods at will whenever you want. And change the properties to whatever value possible. For example, you can use window’s alert() method to make a popup box appear. You can check the location or address of a window or web page by accessing the location property of the window object.

JavaScript has predefined objects. A few of them are String, Array, Object, and Math. When dealing with JavaScript and web pages, HTML DOM will also provide you with predefined objects that were created from the HTML page that was rendered.

On the other hand, almost everything in JavaScripts is treated as an object. For example, string values are treated as objects under String. Because of that, string values or variables containing strings can use methods and properties of String. The same goes with arrays, and the list goes on and on.

Assigning Objects to Variables — By Value and By Reference

Yes, you can assign objects to variables. However, please do note that unlike the usual assignment of literals in a variable, variables that get assigned with an object behave differently.

In programming, assigning to a variable can be by value or by reference. By default, assignment of variables containing regular literals such as numbers and strings to another variable is by value. By value assignment means that the value of the variable being assigned will be only copied to the value of the variable on the left side of the assignment operator. For example:

```
> var x = 2
< undefined
> var y = 3
< undefined
> x = y
< 3
> x
< 3
> _
```

On the other hand, if you assign an object or an array variable to another variable, the assignment will be automatically by reference. By reference assignment means that instead of copying the value of the variable being assigned, the object or array will be referenced. Referencing in programming is like creating another “identifier” that can serve as another identifier holding the methods, properties, or content of another object. For example:

```
> var x = 3
< undefined
> var yObject = new Object()
< undefined
> yObject.sampleProperty = 6
```

< *6*

> x = y

< *Object {sampleProperty:6}*

> x

< *Object {sampleProperty:6}*

> x.sampleProperty = 90

< 90

> y.sampleProperty

< 90

> _

In this example, object y was created. After that, it was given a property named sampleProperty with a value of 6. When y was assigned to x, you can see that its property was included in the assignment. However, when the example tried to change the property that was inherited, the same property in object y reflected the change, too.

Technically, with referencing, the object that was assigned to the variable will receive all the changes that will be applied to the variable that received the assignment. Usually, programmers take advantage of this by referencing an object with a long name to a short name variable. For example:

> var x = document.getElementById('sampleParagraph')

< undefined

> x.textContent

< "This is a sample paragraph"

> x.textContent = "This has been modified through JavaScript."

< "This has been modified through JavaScript."

> _

Instead of repeatedly accessing the sampleParagraph in the page using the getElementById method, which will make you type a lot, you can just reference it to a short named variable such as x. It is much convenient, and can assure you clean code.

Object Creation Using Object Literal

In JavaScript, multiple methods on how to create an object exist. One of the simplest and easiest ways is to create one using an object literal. Doing it this way is almost too similar in creating an array. For example:

```
> var exStove = {fuel:100, type:“single burner”, state:“off”, fuelPerSecond:1}
< undefined
> exStove
< Object {fuel: 100, type: “single burner”, state: “off”, fuelConsumptionPerSecond: 1}
> exStove.fuel
< 100
> _
```

The statement created the object named exStove. It was assigned with the properties fuel, burner, state, and fuelConsumptionPerSecond. To access the properties, you can use the dot accessor. For example:

```
> var exStove = {fuel:100, type:“single burner”, state:“off”, fuelPerSecond:1}
< undefined
> exStove
< Object {fuel: 100, type: “single burner”, state: “off”, fuelConsumptionPerSecond: 1}
> exStove.fuel
< 100
```

When typing the code in a file or within the HTML, you can make your statement much simpler and easier to read by adding line breaks on the object literal as if you are placing a CSS declaration. For example:

```
var exStove = {
        fuel:100,
        type:“single burner”,
        state:“off”,
```

```
        fuelPerSecond:1
}
```

If you want to add another property in your object, you can easily do that by just assigning a value to the property that you want. For example:

```
> var exStove = {fuel:100, type:“single burner”, state:“off”, fuelPerSecond:1}
< undefined
> exStove
< Object {fuel: 100, type: “single burner”, state: “off”, fuelConsumptionPerSecond: 1}
> exStove.fuelType
< undefined
> exStove.fuelType = “LPG”
< “LPG”
> exStove.fuelType
< “LPG”
> _
```

You can also change the value of the property this way, too.

Chapter 15: Classes, Properties, and Methods

Before you proceed on the other methods in creating objects in JavaScript, you must learn about classes. If you have prior experience with other OOP languages, you might get confused a bit in dealing with objects and classes in JavaScript.

In a nutshell, classes in JavaScript are more casually referred to object types or object constructor. But to prevent more confusion, read the next sections carefully.

More often than not, most programmers tend to associate classes with OOP. If there are classes in a language, then it is probably an OOP language. However, in JavaScript, classes or even the keyword class does not exist since it is a prototype-based language.

Note: As of June 2015, ECMAScript 6 or ES6 Harmony has been released. In this version, the keyword class has been included. Due to that, another method of defining "classes" in JavaScript is available.

However, due to it being recently released, it is still better to stick with the standard methods of creating classes. Nevertheless, if the syntax of creating classes in JavaScript is bugging you, you can go ahead with using the class keyword.

But as of now, this is not advisable. After all, not all browsers are compatible with ECMAScript 6 as of yet. Due to that, older browsers will not recognize the existence of the class keyword in your script, and that might make old browser incapable of running your scripts the way you wanted.

Prototype-based, instance-based, prototype oriented, or classless programming is a subclass of object-oriented programming. Instead of relying to user-defined classes to create objects, you can use the predefined classes or objects to create objects of your own. However, it does not mean that you cannot user-defined objects in this type of OOP.

What are classes anyway? Classes are template for objects. To make it easy to create new objects without repeating the code on how the program can construct the objects, programmers often create templates called classes.

For example, in a massive production of a toy car, a template is needed. The template is the class; the toy car that was created from that template is the object.

Of course, to create a custom object of your own, you must create a pseudo class of your own in your code. Unlike in other languages that rely on the usage of the class keyword to instantiate a class in their code, you will need to rely on creating a constructor for your class instead by using the function keyword. Here is an example:

```
> var ToyCar = function () {};
< undefined
> ToyCar
< function () {}
> _
```

To create an object using that class, you will need to use the new keyword and indicate that it will be an object under a certain class. For example:

```
> var RC_Car = new ToyCar
< undefined
> RC_Car
< ToyCar {}
> _
```

The new object will inherit all the methods and properties that were declared in the class. But this discussion will be continued after discussing about methods, properties, and the constructor.

Constructor Function and new Keyword

What is a constructor function? A constructor function can be used to define an object type or class for an object. You can declare the inherent properties and methods of an object in a constructor function. And to create an object based on a constructor, you must declare it together with the new keyword. Here is an example:

> var exObject = new Object();

< undefined

> exObject

< *exObject*

> _

In the example, we created an object by using the keyword new and defining the class or object type, which is Object(). It was mentioned before that JavaScript has predefined objects such as String, Array, and even Object. These same objects can also be used as constructor functions.

However, constructors like those are rarely used. After all, instead of specifying the data type for the variable, it is much faster to just assign the variable and let the browser do the hard work. JavaScript is a dynamically typed language, so it will be much more efficient.

> var exampleString = new String();

< undefined

> exampleString

< *String {length: 0, [[PrimitiveValue]]: ””}*

> exampleString = “Sample String”;

< “Sample String”

> exampleString;

< “Sample String”

> _

This is much longer than this:

> var exampleString = “Sample String”;

< undefined

> exampleString;

< “Sample String”

> _

Object Creation Using Constructor Function

To create an object using a user defined constructor function, you must declare the constructor first. For example:

> function NewClass() {}

< undefined

> NewClass

< *function NewClass() {}*

> var sampleObject = newClass;

< undefined

> sampleObject

< *function newClass() {}*

> _

Another way you can do this is to do this:

> var NewClass = function() {}

< undefined

> var sampleObject = new NewClass;

< undefined

> sampleObject

< *NewClass {}*

> _

Note: This example uses another method to create a function using the var keyword or statement. Basically, you are assigning a function literal to a variable in this method. This might be confusing for new JavaScript developers. You will surely find function declarations like this every now and then while checking out scripts of other websites.

Chapter 16: Properties and Methods

Properties are variables that are associated with an object. Their value can be only accessed through the object. For example, the property window.name is only exclusive in the window object. It works like that to prevent confusion in coding.

For example, the property or attribute height for a paragraph (<p>) object/element will become messed up if the height property is not exclusive since other objects/elements also have the property height.

Adding properties through an object literal has been already demonstrated in the previous section. Is there any other way to add properties to an object? Yes, there is. And that method is to add it on your object's class or constructor. For example:

```
> var Sim = function() {
        this.hunger = 100;
        this.comfort = 100;
        this.hygiene = 100;
        this.bladder = 100;
        this.energy = 100;
        this.fun = 100;
        this.social = 100;
        this.room = 100;
}
< undefined
> var JohnSim = new Sim();
< undefined
> JohnSim
< Sim {hunger: 100, comfort: 100, hygiene: 100, bladder: 100, energy: 100…}
> JohnSim.hygiene
< 100
> _
```

In case you are familiar with the game The Sims, this is how its developers have coded a

Sim object in the game. Of course, the number of properties it has is not complete, but you will get the idea.

In this example, properties of the Sim object were provided immediately. Do note that whenever you create an object, all the statements within the object type, prototype, or class' function will be executed.

this Keyword

You might have noticed something different in this example. Yes, we used the keyword this. The keyword this is a keyword that automatically references the object that owns or contains the statements. In this case, the keyword this is a reference to the newly created object JohnSim. This keyword makes it easier for you to code and of course, shorten the statements that you write.

Methods

As mentioned before, methods are like actions or commands in programming. They can be called functions, too. However, if they are associated to an object, they are called methods instead. Although, there are other terminologies that can be used, for clarity's sake, the term method will be used instead.

To add a method to a class or object type, you will need to write a separate code block from the constructor. Also, you will need to add the function inside the prototype property of your object. The prototype property is always present in your classes. For example:

```
> var Sim = function() {
        this.hunger = 100;
        this.comfort = 100;
        this.hygiene = 100;
        this.bladder = 100;
        this.energy = 100;
        this.fun = 100;
        this.social = 100;
        this.room = 100;
}
< undefined
> Sim.prototype.eat = function() {
alert("Your Sim is eating.")
}
< function() {
alert("Your Sim is eating.")
}
> var JohnSim = new Sim();
< undefined
> JohnSim.eat()
< undefined
> _
```

Do note that the prototype property will only be available in your class. The object that you will create will not have it. Also, all the functions that you placed on the prototype property will become readily available to the objects.

You can also add methods to your object directly. To do that, all you need to is to do a function literal to a method that you named. For example:

```
> var Sim = function() {
        this.hunger = 100;
        this.comfort = 100;
        this.hygiene = 100;
        this.bladder = 100;
        this.energy = 100;
        this.fun = 100;
        this.social = 100;
        this.room = 100;
}
< undefined
> var JohnSim = new Sim();
< undefined
> JohnSim.stand = function() {
alert("Your Sim is standing.")
}
< function() {
alert("Your Sim is standing.")
}
< undefined
> _
```

When assigning a method directly to an object, you will not need to access the prototype property of the class.

The Concept of Get and Set/Let

Whenever you deal with properties, two things happen behind the scene. First, when you access a property from an object, you are actually asking the program to perform a get action. The get action retrieves the value of the property.

On the other hand, whenever you assign a value to an object's property, you are actually asking the program to perform a let or set a value to the property. In creating JavaScript script, you do not need to worry about this. Nevertheless, the understanding of these concepts will help you once you explore programming more.

However, do note that set and let might have different contextual meaning in different programming languages.

Providing Arguments and Parameters to Your Constructor

Would it not be better if you can just create an object and assign values to its properties in one statement? Well, you can do that. You can just take advantage of arguments and parameters. After all, constructors are functions, too. For example:

```
> var Sim = function(firstName, lastName) {
        this.hunger = 100;
        this.comfort = 100;
        this.hygiene = 100;
        this.bladder = 100;
        this.energy = 100;
        this.fun = 100;
        this.social = 100;
        this.room = 100;
        this.firstName = firstName;
        this.lastName = lastName;
        this.name = firstName + " " + lastName;
        alert("Your new Sim's name is " + this.name);
}
< undefined
> var sim1 = new Sim("John", "Sim")
< undefined
> sim1.name
< "John Sim"
> _
```

Chapter 17: Common Methods

String Methods or Commands

In the web, manipulation or processing text is an important task. Just knowing how to process text well in JavaScript can make you become a valuable web developer. So, here are some methods, commands, or keywords that can help you manage string data.

search() method

The search() method allows you to check if a certain string exists within another string. You can also use search() to give you the exact location the string that you want to find in another string. For example:

```
<body>
<p id = “as” >The string isThistheText? exists in this string.”</p>
<script>
var searchQuery = “isThistheText?”;
var article = document.getElementById(‘as’).innerHTML;
if (article.search(searchQuery)) {
    alert(searchQuery + ” is present on the string: ” + article)
}
else {
    alert(searchQuery + ” is not present on the string: ” + article);
}

</script>
</body>
```

If the value of the search() method returns more than 1, it will be considered as true. The search() method returns 0 if the string being searched do not exist. Do remember that any number aside from 0 is considered True. And the number 0 is False.

On the other hand, if you just want to find the location of the string, you can just use search and the number it returns. For example:

```
<body>
<p id = “as” >The string isThistheText? exists in this string.”</p>
<script>
var searchQuery = “isThistheText?”;
var article = document.getElementById(‘as’).innerHTML;
var position = article.search(searchQuery);
alert(“The string ” + searchQuery + ” starts at character ” + position + “.”)

</script>
```

```
</body>
```

Having knowledge about the position of the string you are searching can allow you to perform other useful functions at the string.

Take note: the search() will only return the first instance of the string you are searching. For example:

```
<body>
<p id = “as” >The string isThistheText? exists in this string. isThistheText? “</p>
<script>
var searchQuery = “isThistheText?”;
var article = document.getElementById(‘as’).innerHTML;
var position = article.search(searchQuery);
alert(“The string ” + searchQuery + ” starts at character ” + position + “.”)

</script>
</body>
```

This example code will always return 11 even if you keep adding another instance of the searchQuery at the end or at any position after the first instance of searchQuery,

Array Methods and Properties

Arrays in programming are usually treated as objects — it is true in JavaScript. And in JavaScript, arrays have built-in properties and methods that can help you manage and manipulate arrays better and easier.

Length

One of the properties that you will be using a lot when handling arrays is length. The length property allows you to know how many data or elements are within an array. For example:

```
var exampleArray = [“Google”, “Yahoo!”, “Bing”];
alert(exampleArray.length);
```

In this example, the message box will contain the number 3 because there are three values inside exampleArray.

Dot Accessor Operator

The dot operator is JavaScript’s accessor operator. The accessor operator allows programmers to access the properties and methods of an object. Since arrays are objects, you can use the dot accessor to access its properties and methods such as length. The syntax for using the dot accessor is: object.properties.

Push

Push is an array object method. What it does is that it allows you to add another value in an existing array. For example:

```
var exampleArray = [“Google”, “Yahoo!”, “Bing”];
alert(exampleArray.length);
var exampleArray.push(“DuckDuckGo”);
alert(exampleArray.length);
```

As you will see, the values within the exampleArray will increase by one.

Chapter 18: JavaScript Math

Math is a built-in object in JavaSript, which has methods and properties for math functions and constants, but not a function object. Math is not considered a constructor in JavaScript, unlike other global objects. The methods and properties of JS Math are static. We can consider the constant pi as Math.PI; we can refer to the sine as Math.sin(x). Take note that the X refers to the argument of the method. In JavaScript, constants are prescribed with complete precision, thanks to real numbers (as opposed to integers).

Properties of Math in JS

Refer to the table below for the different math properties used in JavaScript

Properties	Description
Math.E	The Euler's Constant, which is the base of natural logs, about 2.718
Math.LN10	The natural log of 10, about 2.303
Math.LN2	The natural log of 2, about 0.693
Math.LOG10E	Base 10 log of E, about 0.434
Math.LOG2E	Base 2 log of E, about 1.443
Math.SQRT2	Sq. root of 2, about 1.414
Math.SQRT1_2	Sq.root of ½ eq, 1 / sq. 2, about 0.707
Math.PI	Ratio of circle circum to the diam about 3.1415

Methods

Take note that the trigo functions (atan2(), atan(), acos(), asin(), tan(), cos(), sin()) return radians angles. In order to conv radians into degrees, you can divide using Math.PI and multiply it for reverse conversion.

In addition, the precision of math functions in JS are dependent on implementation. Hence, various browsers will yield varying results. Even similar engines on a different architecture or operating system may yield varying results.

Refer to the table below for the different Math Methods in JS

Methods Math.	Description
.abs(x)	Yields the absolute number value
.acosh(x)	Yields the number's hyperbolic arcosine.
.acos(x)	Yields the number's arcosine
.asinh(x)	Yields the number's hyperbolic arcsine
.asin(x)	Yields the number's arcsine
.atanh(x)	Yields the number's hyperbolic arctangent
.atan(x)	Yields the number's arctangent
.atan2(x, y)	Yields the quotient's arctangent arguments
.cbrt(x)	Yields the number's cube root
.clz32(x)	Yields the number prime zeros of a 32-bit int
.ceil(x)	Yields the min integer >= to x
.cos(x)	Yields the number's cosine
.coshh(x)	Yields the number's hyperbolic cosine
.exp(x)	Yields Ex (x=argument, E=Euler's constant (log base)

.floor(x)	Yields the biggest int <= to x
.fround(x)	Yields the number's nearest single precision float rep
.hypot(x)	Yields the sq root of the sum of sq of args
.imul(x,y)	Yields the result of multiplying 32-bit int
.log(x)	Yields the number's natural log
.log2(x)	Yields the number's base 2 log
.log1p(x)	Yields the number's natural log of 1+x
.log10(x)	Yields the number's base 10 log
.max([x[,y[,-]]])	Yields the biggest zero or more numbers
.min([x[,y[,-]]])	Yields the min zero or more numbers
.pow(x. y)	Yields base to exp power
.random	Yields a pseudo-random number between zero and 1
.sign(x)	Yields the sign of the number, which indicates if x is zero, negative, or positive
.round(x)	Yields the number's value rounded to the nearest int
.sin(x)	Yields the number's sine
.sinh(x)	Yields the number's hyperbolic sine
.sqrt(x)	Yields the number's positive sq root
.tan(x)	Yields the number's tangent
.tanh(x)	Yields the number's hyperbolic tangent
.trunc(x)	Yields the number's integral part, which removes any fractional digits

.

Chapter 19: Advanced Data Types – Data Conversion and Constructor

Like other programming languages, JavaScript have built-in data structures. However, these usually differ according to the language. In this chapter, we will try to list all the built-in data structures used in JavaScript and the properties they include. You can also use them to construct other data structures. If possible, other programming languages are drawn.

Dynamic Typing

JavaScript is a dynamic language or a loosely typed. Hence, there is no need to declare the variable type earlier. The type will be automatically determined while you are processing the program. So it will also mean that we can have similar variable as various types:

```
var jvsc = 34; // jvsc is now a Number
var jvsc = "car"; // jvsc is now a String
var jvsc = false; // jvsc is now a Boolean
```

Data Types

Based on the latest ECMA Script, the standard defines seven types of data:

Primitive Data Types

- Null
- Number
- Boolean
- Undefined
- String
- Symbol (added in ECMA Script 6)
- And Object

Primitive Values

All kinds except objects describe values that are fixed. For instance, Strings are immutable, which is not similar to C. These values are primitive values.

Null Type

This type only has one value, which is null.

Boolean Type

Boolean describes a logical entity, and could only have two values: true and false.

Undefined Type

A value is considered as undefined if its variable has not been assigned.

Number Type

There is only one number type based on the ECMAScript standard. This is the range between $–(2^{53} -1)$ and $2^{53} -1)$ or double-precision 64-bit binary format IEEE 754 value. Take note that there's no certain kind of integers. Aside from the ability to signify floating-point number, the number type also has three values that are symbolic: NaN (Not a Number), -Infinity, and +Infinity.

In order to check for smaller of larger values compared to positive/negative Infinity, we can utilize the constants Number.MIN VALUE or the Number.MAX VALUE and beginning with the ECMAScript 6, we can also check if the number is of double precision floating point through Number.isSafeInteger() and the Number.MIN SAFE INTEGER and Number.MAX SAFE INTEGER. Outside this range, the JavaScript numbers are not anymore safe.

The number type with only a single integer has two major representations: 0 can be +0 or -0. Zero is a default for +0. This has almost zero impact in the praxis. For instance, +0===-0 is true. But you will take note of this if you use 0 as a divisor:

```
> 34  / +0
Infinity
>34 / -0
-Infinity
```

Even though the number usually signifies only the value, JavaScript can offer several binary operators. We can use them to signify several Boolean values in one number through bit masking. This is often regarded as a malpractice. But JS provide no other methods to signify a Boolean set such as an array of Booleans or an object with Boolean values defined to name the properties.

Bit masking also has the tendency to make the code hard to read, interpret, and keep. It is important to use these strategies in very limited environments such as when you try to cope with the limitation of storage or in extreme cases if every bit over the network will count. You should only consider this strategy if this is the only option you can use for size optimization.

String Type

The String type in JavaScript is utilized to signify textual type of data. This is a set of values unsigned 16-bit integer. Every element in the String will occupy a position in the String. The first element is within index 0, and the next will be 1, and 2, and so forth. The String length refers to the number of values within.

Not similar to programming languages such as C, the Strings in JS are immutable. Hence, when you create a string, you can’t modify it. But it is still possible to make another string depending on an operation of the original string. This is true of the original substring by choosing individual letters or by calling the String.substr(). This is also true for a concatenation of two individual strings by using the addition operator or by calling String.concat().

Code String Typing

You may be tempted to use strings to signify complicated data. This method has some short-term advantages:

- Debugging is easy with string. Anything inside the string will be printed.
- A lot of APIs such as XMLHttpRequest, local storage values, and input fields are using Strings as common denominator.
- It’s easy to create complicated strings using concatenation

Conventionally, it is possible to signify any data structure using a string. But this is not a good idea. For example, you can emulate a list using a separator even though a JS array is a more ideal solution. But if we use a separator in the list elements, there is the tendency for the list to be broken. You can still choose an escape character, but still this method will require a lot of conventions and will just create unnecessary maintenance problem.

It is ideal to use strings only for textual data. In signifying complicated data, you can parse strings and utilize the suitable abstraction.

Symbol Type

The JS ECMAScript 6 features Symbols, which are unique and immutable primitive value and could be used as the key of an Object property. In some languages, Symbols are known as atoms. While in C, they can be compared to enum.

Objects

In programming, an object refers to a value in memory that is possibly referenced by an identifier.

Properties

In JS, objects could be regarded as a group of properties. Using the object literal syntax, you can initialize a limited set of properties, which you can easily add or remove. The values of property could be of any type, which includes other objects that will enable creating complicated data structures. You can identify properties using key values, which could either be a Symbol or a String value.

We have two kinds of object properties that have specific attributes – The accessor property and data property.

Accessor Property

This property associates a key with one or two accessor functions in order to store or retrieve a value. This property has the following attributes:

ATTRIBUTE	TYPE	DEFAULT VALUE	DESCRIPTION
[[Configurable]]	Boolean	False	If false, you can't delete the property and you can't change it to a data property.
[[Enumerable]]	Boolean	False	If true, you can enumerate the property in for…in loops.
[[Set]]	Undefined or Function Object	Undefined	You can call this function using an argument containing the assigned value This will be executed once you specify property to be changed
[[Get]]	Undefined or Function Object	Undefined	You can call this function using an empty arg list and retrieve the value property if you perform a get access to the value

Take note that attribute is often use in JS engine. Hence, there is no way to directly access it. Hence, the attribute is written within two squared braces rather than one.

Data Property

Data property relates a key with a value. It has the following attributes:

ATTRIBUTE	TYPE	DEFAULT VALUE	DESCRIPTION
[[Configurable]]	Boolean	False	If false, you can't delete the property and you cannot change the attributes other than the [[Writable]] and [[Value]].
[[Enumerable]]	Boolean	False	If true, you can enumerate the property for…in loops.
[[Writable]]	Boolean	False	If false, you can't change the [[Value]] property
[[Value]]	Any Type	Undefined	You can retrieve the value through a property's get access.

The following attributes became obsolete in the ECMA Script3, and renamed in the ECM

Attribute	**Type**	**Description**
DontDelete	Boolean	[[Configurable]] attribute – reverse state of ES5
DontEnum	Boolean	[[Enumerable]] attribute – reverse state of ES5
Read-only	Boolean	[[Writable]] attribute – reverse state of ES5

Accessor Property

The accessor property relates a key with get and set functions in order to store or retr following has its attributes:

Attribute	Type	Default Value	Description
[[Configurable]]	Boolean	False	If false, you can't delete the property and you can't change it to data property.
[[Enumerable]]	Boolean	False	If true, you can enumerate the property for…in loops
[[Set]]	Undefined or Function Object	Undefined	You can call the function using an arg containing the assigned value and executed if you try to change a specified property.
[[Get]]	Undefined or Function Object	Undefined	You can call the function using an empty arg list and restore the value property by performing a get access to the value.

“Normal” Functions and Objects

A JS object is considered as a mapping between values and keys. Keys are Symbols values could be anything. This will make the object a normal fit for hashmaps. Meanwhi normal objects, which can be called.

DATA CONVERSION

Java is a loosely typed language. Hence, as a programmer, you still need to consider th values that you are dealing. A usual error in browser scripting is to read the property control that the user can type a number and will add this value to a different number. The of form controls are strings. Even the characters the series contains still signify a number. string to a value, even if this value is a number, will result to the second value type- string, which is concatenated to the end of the first string value originating from the fo error is caused by the dual nature of the positive operator used for numeric addition a concatenation. Hence, the nature of the operation performed is distinguished by the co operands are numbers to begin with will the positive operator will execute addition. Or el all the operands to strings and performs concatenation.

The discussions that follow is illustrated using tables of values generated using JavaS operations. The headers of the tables show the value as signified in the JS source code their internal representation.

For instance, 321e-2 as a number was the character series encoded in the source co interpreted as a number value of 3.21. The different values here have been selected to sho conversion type. Take note that these aspects may not be true to all the tables presented values are included in the tables for complete comparison, except where no type of con The table bodies list all the results of the different type operations conversion.

Conversion to Boolean

In assessing the expression of an if statement, the JavaScript interpreter will type-convert expression into Boolean so it could arrive into a decision. Meanwhile, different operat their operands to Boolean to identify its action, including the logical operators such as and AND (&&). The operator NOT will convert the operand into Boolean and if its val return false, and if false, it will return true. Because the result of a NOT operation is a which is the inverse of the type-converted veracity of the operand. Two NOT operation yield a Boolean value, which is equal to the result of the type-conversion of the Boolean o

```
var boolValue = !!x;
```

You can use this method to produce the tables below.

Another method of producing a Boolean value, which represents the type-converted vera to pass this value to the function called Boolean constructor.

```
var boolValue = Boolean(x);
```

Numeric Values: Double NOT (!!col)

	NaN	+Infinity	-Infinity	321e-2	16.8	16	8	1.6	1	+0	-0
!!col	false	true	true	true	true	true	true	true	true	false	false

If the numbers are transformed into a Boolean, zero will become false. Other numbers wi of the special numeric value Not a Number (NaN) used if another kind is transformed t this conversion will not result in a sensible number. Take note that NaN is always fals negative and positive infinity, while not finite values, are non-zero numbers and will alw to a true Boolean.

String Values: Double NOT (!!col)

	“xx”	“-0x10”	“-010”	“0xFF” (Hex)	“0x10” (Hex)	“010”	(Octal)	“321e-2”	“16.8”	“16”	“8”	“1.6”	“1”	“0”	“-1.6”	“” (empty string)
!!col	true	true	true	true	true	true	true	true	true	true	true	true	true	true	true	false

The rules are even easier to understand for converting string to Boolean, because all no will be true and empty strings will be false.

Other Values: Double NOT (!!col)

	function() {return;}	new Object()	false	true	null	undefined
!!col	true	true	false	true	false	false

Null and Undefined will be converted to Boolean false values, which are not converted, are always true.

This is the most important aspect of type-converting to Boolean, because it per differentiate between properties in an environment, which could be undefined or could re Considering a null or an undefined value as an object will result to errors. Hence, if y code can prevent producing errors by wrapping the code, which likes to gain access an if. Including the suspect reference to the expression object will be converted into Boolea will be false if the object is not existing while true if it is existing.

```
if(document.documentElement){
        scrollX=document.documentElement.scrollLeft;
}
```

The double NOT operation will allow the setting of Boolean flags, which you can us existence of different objects:

```
var bolCodLe = !!document.documentElement;
...
if(hasCodLe){
        scrollX = document.documentElement.scrollLeft;
}
```

Conversion to String

As discussed earlier, conversion type to a string will usually result from the action of the if one operator is not a number. The simplest way to get the string, which results from the is through concatenation of value to an empty string. This technique is often used to pr below.

Another way is to convert a value into a string to pass it as an arg to the constructor function:

```
var stringValue = String(x);
```

Numeric Values: type-convert to string (“” + col)

	NaN	+Infinity	-Infinity	321e-2	16.8	16	8	1.6	1	+0	-0
“”+col	NaN	Infinity	-Infinity	3.21	16.8	16	8	1.6	1	0	0

Take note that the number produced from the source code 321e-2 has generated the str string signifies the internal number generated from the source code. But the internal num will take the form of double precision IEEE floating point, and which means that we ca numbers with precision. The outcome of the operations could only yield close estimate converted into strings, this string will signify the estimate and could be undesirable or un usually needed to use custom functions to yield string representations of numbers in the f Rare is the case that the type conversion mechanism is suited to generate numeric ou presentation.

Other Values: type-convert to string (“” + col)

	function() {return;}	new Object()	false	true	null	undefined
“”+col	Function() {return; }	[object Object]	false	true	null	undefined

If you type convert the functions or objects to strings, you need to call their toString me default into the Function.prototype.toString and Object.prototype.toString but this coul with a function called to a “toString” property of the function/object. Type conversion of string doesn’t always lead to the source code of the function. The behavior of Function.pr is dependent on the implementation and could vary a lot, like the outcome from the m objects. This includes the methods and objects provided by the environment inclu elements.

Conversion to Numbers

Conversion of values to numbers, especially strings to numbers, is a common requirem use different methods. Any numerical operator aside from the addition and concatenati force type-conversion. Hence, the conversion of a string to a number may involv mathematical operation on the representation of a string, which will not affect the outco multiplying by one or subtracting zero.

```
var valNum = stringValue - 0;
/* or */
var valNum = stringValue * 1;
/* or */
var valNum = stringValue / 1;
```

But the unary + operator will also type-convert the operand to a number, as it will not pe mathematical operations, it will be the easiest method for type-conversion of a string into

Meanwhile, the unary – operator will also include the type-conversion of the operand from negating the value.

```
        var valNum = (+stringValue);

        /* The preceding unary + expression has been confined in the
parentheses.Even though not necessary, this will make your code easier for
better comprehension and clearer operations. This will avoid confusion with
addition as well as the pre and post increment operations.

var x= anyVarNum++ + +stringVar + ++addedNumVar;

- with -

var x= (anyVarNum++) + (stringVar) + (++anotherVarNum);

    (post increment) + (unary plus) + (pre increment)

*/
```

Even though the unary + is the quickest way to convert a string into a number, you can method, which will use the JavaScript type conversion for algorithms. You can call Number as the argument while the return value will be the number, which represents the of the result.

```
var valNum = Number(stringValue);
```

The constructor Number is a slow form of the type-conversion methods. However, if overriding element, it will generate an easier-to-understand source code.

The tables below show the outcomes of type-converting into a number through the unary take note that all the preceding alternative method will generate the same outcome precisely the same algorithm to perform the conversion.

String Values: Type Conversion to Number (+col)

	"xx"	"-0x10"	"-010"	"0xFF" (Hex)	"0x10" (Hex)	"010" (Octal)	"321e-2"	"16.8"	"16"	"8"	"1.6"	"1"	"0"	"-1.6
+col	NaN	NaN	-10	255	16	10	1.23	16.8	16	8	1.6	1	0	-1.6

In converting strings to numbers, it is important to consider the type conversion out strings, which do not signify numbers. The empty string will be converted into the num depends on the application could be damaging to the code. Still, it is crucial to be awar other context strings that follow the JS format for octal number could be disastrou conversion will still treat them as base 10. But the strings, which follow the format for H be interpreted as hexadecimals. Meanwhile, the strings that cannot be interpreted as a num converted to NaN that could be checked for the isNan function. Strings that represen exponential format ("321e-2") are interpreted alongside minus symbols.

Other Values: Type-Convert to Number (+col)

	function() {return;}	new Object()	false	true	null	undefined
+col	NaN	NaN	0	1	0	NaN

Functions and objects are always type-converted into NaN numbers similar to undefined important to take note that null will be type-converted to zero. Possibly because th

converted to Boolean first and then to number. If you take a closer look at the Boole preceding table, null will be converted into Boolean false that will then become numeric need to type-convert these value types to numbers. The conversion is only relevant by accidental outcome of value conversion, which is expected to be a string but could reall arithmetic operations using a value as an operand.

Parsing to Number

Another way of changing a string to number is through a global function designed to p return a number. The function parseFloat will accept a string argument and will return number, which will result from parsing that string. Non-string args will be type-coverted as discussed earlier. The functions for string functions will be interpreted in the string character until they stumble upon a character, which should not be part of the number. A will stop and yield a number based on the characters, which they have interpreted that sh of that number. You can fully exploit this feature, for instance, through a string signify value like parseFloat “34.6eh. The “eh” will be ignored because these characters cannot b preceding set to generate a valid number. The printed number would only be 34.6, whic composition of the CSS and refined of its units.

parseFloat

String Values: parseFloat(col)

	“xx”	“-0x10”	“-010”	“0xFF” (Hex)	“0x10” (Hex)	“010” (Octal)	“321e-2”	“16.8”	“16”	“8”	“1.6”	“1”	“
parseFloat(col	NaN	0	-10	0	0	10	3.21	16.8	16	8	1.6	1	0

Through the parseFloat function, the empty string will generate NaN, along the string subject to numerical interpretation. The exponential format is interpreted and the primary format will not block the reading of the string as a decimal number. Hexadecimal strings the number zero, as the following “x” will not be read as part of a number. Hence, parsin the primary zero.

	function() {return;}	new Object()	false	true	null	undefine
parseFloat+col	NaN	NaN	NaN	NaN	NaN	NaN

The non-string values are first changed into a string, which is used through the parseFl type-conversion to a string will not usually lead in a string that could be read as a numbe be NaN. The functions and objects could have a custom method toString, which could re could be seen as numbers but this will be a special need.

parseInt

The function parseInt will behave like the function parseFloat, aside from its feature of r argument as an integer and as an outcome will recognize fewer characters as potential part of this number. You can also use parseInt to convert a floating point number into in this is not an ideal method, because if the argument is of numeric type, this will be chan and then will be parsed as a number. This method can be very inefficient. This could gen outcomes with numbers like 3e-300, in which the next smaller integer will be zero. But yield 2.

In addition, because of the numerical format employed by JavaScript, the numbers are by near estimates. For instance, 1/6+1/3+1/2 = 0.999999, which will not be interpreted the function parseInt, the equation will result to zero if you call it to act on the operation r

In rounding integers to one, Math.floor and Math.round.Math.ceil are more ideal. For a d which could be expressed as a 32-bit signed integer, the bitwise operation shown belo suitable:

Numeric Values: parseInt(col)

	NaN	+Infinity	-Infinity	321e-2	16.8	16	8	1.6	1	+0	-
parseInt(col)	NaN	NaN	NaN	1	16	16	8	1	1	0	0

If it is acting on number, the result of the argument’s early type-conversion to a string wil results. Take note that the value 3210-2 is the number 3.21 internally, and this type will b the string “32.1”. Hence, the entry in the table above could look odd, but the results are ac

String Values:parseInt(col)

	“xx”	“-0x10”	“-010”	“0xFF” (Hex)	“0x10” (Hex)	“010” (Octal)	“321e-2”	“16.8”	“16”	“8”	“1.6”	“1”	“0
parseInt(col)	NaN	-16	-8	255	16	8	321	16	16	8	1	1	0

Strings in the hexadecimal and octal number formats will signify integers and parseIn according to the rules of JS source code, even if they have leading minus symbols.

Other Values: parseInt(col)

	function() {return;}	new Object()	false	true	null	undefined
+col	NaN	NaN	NaN	NaN	NaN	NaN

As the parseInt performs type-conversion of the non-string args to strings, it will alw same outcomes for function, object, undefined, null, and Boolean arguments like in parse

ToInt32

ToInt32 is an internal function, which you can only use in the JS implementation, and directly from the scripts similar to the method for parseInt. This is a bit off relevant converting Javascript values into numbers, but this could be used in special cases.

The bitwise operators such as AND (&) and OR (|) are operating on numbers, so they are the operands into numbers. But they can still operate on 32-bit signed integers. Hence, wi value, they call the internal ToInt32 function with the number as the argument and use th as the operand. This returned value is a constant 32-bit integer.

The result could be similar to parseInt added with type-conversion of numbers. Even tho is restricted in range to 32 bits, this is always numerical and not +/- Infinity or NaN.

Similar to using numerical operators for operations with no effect on the value of any n doable to do a bitwise operation, which has no effect on the value yielded by calling t following tables were produced through a zero or a bitwise operation.

Numeric Values: ToInt32(col|0)

	NaN	+Infinity	-Infinity	321e-2	16.8	16	8	1.6	1	+0	-0
col|0	0	0	0	3	16	16	8	1	1	0	0

Notice that –Infinity, +Infinity, and NaN are all changed to zero, while the floating p truncated to integer.

String Values: ToInt32(col|0)

	“xx”	“-0x10”	“-010”	“0xFF” (Hex)	“0x10” (Hex)	“010” (Octal)	“321e-2”	“16.8”	“16”	“8”	“1.6”	“1”	“0”	“-1
(col|0)	0	0	-10	255	16	10	1	16	16	8	1	1	0	-1

The string values, which will be type-converted into NaN will be converted into as zero fr

Other Values: ToInt32(col|0)

	function() {return;}	new Object()	false	true	null	undefined
col|0	0	0	0	1	0	0

The undefined functions and objects are transformed to zero through this operation. Boolean true will be changed to the value of 1.

Conversion of User Input

Many of the mechanisms for obtaining user input as well as prompt, generate results in you expect the user to input a number, it is still needed to enter something. If you need to into a number for future operations, you can use any method discussed above, just mak method will be suitable to the input of the nature. Results with some typos or erroneo challenging to detect and manage.

Before changing a string to a number, it may be ideal to employ a Regular Expression i the content of the string to make certain that they are conforming to a format that is acce serve to get rid of the string values, which may otherwise suffer from the nuances of the that converts processes if applied to string values that are unexpected.

Chapter 20: Dates and Time

The Date Object

The data object enables basic storage and recall of dates and times. Below is the syntax for the date object:

```
dateObj = new Date()
dateObj = new Date(dateVal)
dateObj = new Date(year, month, date[, hours[, minutes[, seconds[,ms]]]])
```

Below are the parameters of the date object:

Parameter	Description
dateObi	Mandatory. This is used to assign the Date object.
dateVal	Mandatory. When a numeric value, this parameter signifies the number of milliseconds in UCT between the certain time and Jan 1 1970 midnight. If dateVal is a string, this will be parsed based on the rules in JavaScript. Meanwhile, the dateVal argument could also be a VT_DATE value as generated from certain objects ActiveX.
ms	Optional. Refers to the number between zero to 999, which specifies milliseconds.
seconds	Optional. Should be included if you add milliseconds. This refers to the number from zero to 59, which signifies the seconds.
minutes	Optional. Should be included if you add seconds. This refers to the number from zero to 59, which signifies the minutes.
hours	Optional. Should be included if you add minutes. This refers to the number from zero to 23 (mn to 11:00 pm, which refers to the

	hour)
month	Mandatory. The month is a number from zero to 11 (Jan to Dec)
year	Mandatory. Should be complete year such as 1996 and not 96
date	Mandatory. The date as an integer in between one and 31.

Take note that a data object will contain a number signifying a certain occurrence in time to within a millisecond. When the value of an argument is larger than the range or this is a negative number. Other stored values will be modified. For instance, if you enter 240 seconds, the JavaScript will redefine the number as four minutes.

When the number is NaN, the object will not represent a particular time instant. When you pass no parameters to the Date object, it will be initialized to UTC. Remember, a value should be provided to the object before using this. The date range could be signified in the object Date about 285616 years before and after Jan 1 1970.

The example below shows the use of the Date object.

```
var datString = "Today is: ";

var newDate = new Date();

//Obtain the year, day, and month.
datString += newDate.getFullYear();
datString += newDate.getDate() + "/";
datString += (newDate.getMonth() + 1) + "/";

document.write(datString);

//Result: Today is: <date>
```

Date Object Properties

There are two properties of the Date Object: prototype Property and constructor Property.

Property	Description
Prototype	Yield a reference to the objects class prototype
Constructor	Specifies the function that creates an object

Date Object Functions

Functions	Description
Date.UTC	Yields the number of milliseconds between Jan 1, 1970 UTC and the date added
Date.parse	Parses that string that contains a date, and yields the milliseconds between the date and midnight Jan 1, 1970 UTC
Date.now	Yields the milliseconds between Jan 1, 1970 and the present time and date

Date Object Methods

Below is the list of Date object methods:

Method	Description
getFullYear	Yields the year value in local time.
getHours	Yields the hours value in local time.
getDay	Yields the week day in local time.
getDate	Yields the month day through local time
getMinutes	Yields the minute value in local time
getMonth	Yields the month value in local time
getSeconds	Yields the seconds value in local time
getMilliseconds	Yields the milli seconds in local time
getTime	Yields the time within an object Date as the no. of milli seconds since Jan 1, 1970 midnight
getUTCDate	Yields the month day in UTC
getUTCDay	Yields the week day in UTC
getUTCFullYear	Yields the year in UTC
getUTCHours	Yields the hours in UTC
getUTCMilliseconds	Yields the milliseconds in UTC
getUTCMinutes	Yields the minutes in UTC
getUTCMonth	Yields the month in UTC
getSeconds	Yields the seconds in UTC
getTimezoneOffset	Yields the minute difference between the UTC and the computer

getYear	Yields the value year
getVarDate	Yields the value VT_DATE in a Date object
isPrototype	Yields a Boolean value, which signifies if an object is existing in the prototype chain of an object
hasOwnProperty	Yields the Boolean value, which signifies if an object has a property with the particular name
propertyIsEnumerable	Yields a Boolean value, which signifies if a certain property is a composition of an object and if it is enumerable.
setSeconds	Defines the seconds value in local time
setMonth	Defines the month value in local time
setMinutes	Defines the minute value in local time
setMilliseconds	Defines the milliseconds value in local time
setHours	Defines the hour value in local time
setDate	Defines the numerical month day in local time
setFullYear	Defines the value year in local time
setUTCSeconds	Defines the seconds value in UTC
setUTCMonth	Defines the month value in UTC
setUTCMinutes	Defines the minutes value in UTC
setUTCMilliseconds	Defines the milliseconds value in UTC
setUTCHours	Defines the hours value in UTC
setUTCFullYear	Defines the year in UTC
setUTCDate	Defines the numerical month day in UTC
setTime	Defines the date and time value in object Date
setYear	Defines the year value in loc time
toLocaleTime	Yields a time as a value string suitable to the current local of the host environment

toLocaleDateString	Yields a data in string value suitable to the current local of the host environment
toDateString	Yields a data in value string
toISOString	Yields a date in value string but in ISO format
toGMTString	Yields a date changes to a string in GMT
toJSON	Yields a changed data of type object prior to the JSON serial
valueOf	Yields the prime value of a defined object
toTimeString	Yields the time as a string value
toString	Yields a string representation of the object
toUTCString	Yields a date changed to a string in UTC

Calculating Dates and Time using JavaScript

It is possible to use the object Date in order to perform usual clock and calendar tasks like computing elapsed time and date comparison.

Setting up Date to the Existing Date

If you make an instance of the object Date without defining a date, it will return a value, which will signify the present date and time, which also includes the millisecond, the second, the minute, hour, day, and year. You can then modify or read this date and time.

The example below shows the process of instantiating a date without the use of any type of parameters and showing the result in *mm-dd-yy* format.

```
var dat = new Date();

//Shows the month, day, and year. getMonth() will show a zero-based number.
var month = dt.getMonth()+1;
var day = dt.getDate();
var year = dt.getFullYear();
document.write(month+'-'+ day + '-' + year);

//Result: present month, day, year
```

Setting up a Particular Date

It is possible to set a particular date through passing string date to the constructor.

```
var dt = new Date('1/16/2016');
document.write(dt);

//Result: Sat Jan 16 00:00:00 PDT 2016
```

The time zone shown in the date string is suitable to the time zone defined in the local machine. It is a good thing that JS is dynamic about the string format used as the parameter. For instance, you can use “1-16-2016”, “January 16, 2016”, or “16 Jan 2016”.

Chapter 21: Regular Expressions in JavaScript

Regular Expressions in JavaScript are used to match the character combinations in strings. They are also referenced as objects, and they are used with the test and exec methods of RegExp with the split, search, replace, and match methods of String. In this chapter, we will discuss the JavaScript regular expressions.

Constructing a Regular Expression

You can create a regular expression in one of the two ways:

Using a regular expression that is composed of a pattern within slashes:

```
var re = /ac+b/;
```

RegExp literals offer compilation of the regular expression if you load the script. If the regular expression could stay fixed, it is ideal to use this for better performance. You can also call the function constructor of the object RegExp as you can see below:

```
var re = new RegExp("c+ab")
```

You can easily compile the runtime RegExp through the constructor function. You can use this function if you are aware that the pattern for regular expression could change, or if you are not aware of the pattern, and you are sourcing it from other avenues like user input.

Writing RegExp Pattern

A RegExp pattern is composed of basic characters like /abc/ or a mixture of simple and special symbols such as /cb*a/ or /Subject (\x).\x*/. This example uses parentheses that are used as a memory device. The match made through this composition of the pattern will be stored for future use.

Basic Patterns

Basic patterns are created of symbols for which that you like to look a direct match. For instance, the pattern /cba/ will match the combination of character in strings only if specifically the characters ‘cba’ are matched together and in the right series. This match will be successful in the string that includes ‘cba’.

Special Characters

If the search for a match needs something more than a direct match like finding one or more c's or looking for white space, the pattern will also include special characters. For instance, the pattern /cb*a/ will match any combination of character, in which the c will be followed zero or more b's. The character * means that zero or more instances of the earlier item, which will be followed by c.

The table below shows the complete list as well as the description of the special characters, which you can use in RegExp.

CHARACTER	DESCRIPTION
^	Will match the start of input. Can also match easily after the character line break if the multi-line flag is defined as true. For instance, /^B/ will not match the 'B' in "a B", but will match the 'A' in "An I". Also remember that '^' has a different meaning if it is used as the first character in a set series.
\	The character '\' will match according to these rules: The character '\' preceding a non-special character will signify that the next character is special and should not be interpreted literally. For instance, a 'c' without the character '\' will match the lowercase c's if they happen. However, the '\c' in itself will not match any character, as it will form the special word character boundary. The character '\' that precedes a special character will signify that the next character is not special and must be literally interpreted. For instance, the sequence /c*/ depends on the asterisk (*) to match zero or several c's. Meanwhile, the series /a*/ will ignore the * as a special character so the strings will match.
*	Will match the preceding expression 0 or several instances. This character is equal to {0,}. For instance, /co*/ will match the 'cooooo' in "The baby cooooooed" and 'c' in "The cat meowed", but not in "The teen smirked."

$	Will match the end of the input. If the multi-line flag is set to true, this will also immediately match before the character line break. For instance, /g$/ will not match the ‘g’ in “digger”, but will match in “dog”.
?	Will match the preceding expression zero or one time. This is equal to {0,1}. For instance, /e?re?/ will match the ‘er’ in “ranger” as well as the ‘re’ in “ogre” and also the ‘r’ in “torso”. If you use this immediately after the quantifiers such as {}, ?, +, or *, this will make the quantifier non-greedy, which matches the fewest possible characters, in comparison to the default that could match characters as much as possible. For instance, if you use /\g+/ to “321def” will match 321. But adding /\g+?/ to the same string will only match 3.
+	Will match the preceding expression once or several times. This is equal to {1,}. For instance, /r+/ will match the ‘r’ in “brandy” and all the r’s in “grrrrrrrrowl”, but not in “bndy”
.	The decimal point will match any single character aside from the newline character. For instance, /.s/ will match ‘as’ and ‘is’ “Say yes if you think he is smart” but not ‘say’.
(x)	Will match ‘x’ and will store the match, as shown below. The parentheses in this character are known as capturing parentheses. The ‘(cute)’ and ‘(bear)’ in the series /(cute) (bear) \1\2/ match and the series will store the first two words in the string “cute bear cute bear”. The \1 \2 in this pattern will match the last two words of the string. Notcie that the \1, \2, \n are used in the matching composition of the regex.
x(?=y)	Will match ‘x’ if ‘x’ is followed by ‘y’. For instance, /Jessica(?=Alba)/ will only match ‘Jessica’ if it is followed by ‘Alba’. /Jessica(?=Alba|Simpson)/ will match ‘Jessica’ if this is followed by ‘Alba’ or ‘Simpson’. But neither ‘Alba’ nor ‘Simpson’ is part of the matching results.
(?:x)	Will match ‘x’ but will not store it in memory. The parentheses used in this character are known as non-capturing parentheses. These will allow you to define subexpression for regexp operators to perform their function. Take a look at this example: /(?:bee){1,2}/ If this expression

	was /bee{1,2}/, the characters {1,2} will only apply to the last 'e' in 'bee'. Because we have used non-capturing parentheses, the {1,2} function will apply for the whole word 'bee'.
x(?!y)	This is known as a negated lookahead. Will match x if it is not followed by y. For instance, /\d+(?!.)/ will match a number if it is only followed by a decimal point. The regexp /\d+ (?!.)/.exec("2.564") will match 564, but not 2.564.
x|y	Will match x or y. For instance, /yellow|black/ will match 'yellow' in "yellow flag" and 'black' in "black shirt".
{n,m}	In this character, n and m are positive integers and n is lower than or equal to m. This will match at least n and at most instances of the preceding expression. If you omit m, it will be treated as infinity. For instance, /a{1,3}will match nothing in "brndy", the 'a' in "brandy." The first two a's in "brandy," and the first three a's in "braaaaaaandy". Take note than in matching "braaaaaaandy", the match will be "aaa", although the primary string carries more than a's.
{n}	Wil match precisely n instances of the preceding expression. N should be a positive integer. For instance, /e{2} will not match the 'e' in "rent", but it will match all the e's in "reent" as well as the first two e's in "reeent".
[^xyz]	This is a complemented or a negated character set. It will match anything, which is not confined in the braces. You can define a range of characters through the use of a hyphen. Everything that will work in the regular character set will also function here. For instance, [^bod] is similar to [^b-d].
[xyz]	This character set will match any character confined in the braces, which includes the escape series. Special characters such as the asterisk (*) and the dot (.) will lose their specialness when confined within the character set, so there is no need to escape them. You can define a range of characters through the

	use of a hyphen. For example, the pattern [d-g] functions similar to the match of [defg]. Hence, it will match the ‘e’ in “elephant” and ‘f’ in “fowl”.
\b	Will match a word boundary, which matches the position of the word character is not preceded or followed by another character-word. Remember, a matching word boundary will not be included in the match. To put it simply, the length of the word boundary match is 0. You should not confuse this with [\b].) For example: /\bs/ will match the ‘s’ in “soon”; /oo\b/ will not match the ‘oo’ in “soon”, as ‘oo’ is followed by ‘n’ that is a word character /oon\b/ will match the ‘oon’ in “soon” as the ‘oon’ is the string end, and not followed by the character word. /\w\b\w/ will not match anything, as the word character may never be followed by both word and non-word character. Take note that the RegExp in JavaScript will define the certain set of characters to be the character words. Characters that are not specified in the set will be regarded as break words. The character sets are fairly limited, and it solely composed of the Roman alphabet in lower and upper case, underscore symbol, and decimal units. Characters that are accented are considered as word breaks.
\cX	In this character, X refers to a range from A to Z. This will match a character control in a string. For instance, /\cM/ will match the string in control-M.
\B	This character will match a boundary non-word. This will match a position that the preceding and following character are the same, which are both non-words or words. The start and end string are regarded as non-words.
\D	Will match any character that is non-digit, which is equal to [^0-9]
\d	Will match any digit character, which is equal to [0-9]

Use of Parentheses

Adding parentheses in any RegExp pattern will result to that section to be stored in memory. Through this, you can retrieve the substring for other purposes.

For instance, the series /Section (\d+).\d*/ shows added escaped and special characters and will indicate this section of the pattern should be stored in memory. This will precisely match the characters ‘Section’ followed by a single or several numeral character (\d refers to any numeral character while + refers once or more instances). This is followed by a decimal point that is a special character. This precedes the decimal point with \means the series should find the character ‘.’ literally.

If followed by any numeral character zero or more times, (\d refers to numeral character, while * refers to 0 or several times. Also, the parentheses can be used to store the memory for the first matched of numeral character. This series is found in “Read Section 3.4, line 5” while ‘3’ is stored in memory. The series is not found in Section 3 and 4, since these strings are not followed by the decimal point.

In order to match a substring without the storing it to the memory, inside the parentheses, you can preface the series with ?. For instance, (?:\d+) will match one or several numeral characters. However, this will not remember the characters that match.

Using Methods with RegExp

You can use regular expressions with RegExp methods exec and test alongside the String methods: split, search, replace, and match. Below is a short description of these methods:

Method	Description
Split	This String method employs a fixed string or a RegExp in order to break a string into substring arrays
Replace	This String method can look up for a match in a string, and will replace this matched substring with another specified substring
Search	This String method can check for a string match. It will return the match index, or negative 1 if there is a failure in search
Match	This String method can look up for a string match and will return mismatch null or information array.
Test	This RegExp will check for a string match, which returns information array.
Exec	This RegExp method will launch a lookup for a string match, and will return information array.

If you like to determine if a pattern is present in a string, you can use the search or test method. For more info you can use the match or exec methods. If you are using the match or exec and the former is successful, these methods will yield an array and will update the properties of the relevant RegExp object as well as the predefined regular expression object. If the match is a failure, the method exec will yield null that will coerce false.

Chapter 22: Errors and Debugging

Thanks to unavoidable human errors, bugs can instantly conquer your scripts. And since modern browsers do not display error messages on the screen, errors that can render your scripts acting weird or malfunction can become unnoticeable.

Old browsers like Internet Explorer 6 and lower do provide error messages in a page's JavaScript code. However, due to numerous scripts out there that are so prone with errors, error messages became a nuisance rather than a helpful tool for web users.

And because of that and the increasing lack of knowledge about JavaScript by common internet users, error reporting in browsers have become "reserved" to developers only. Nowadays, browsers are equipped with debugging and developer console. And two of those browsers that have great consoles are Chromium (Chrome, Opera, etcetera) and Mozilla Firefox (Pale Moon, Waterfox, etcetera).

Of course, having a debugging console is not enough. After all, the most common behavior that browsers have when they encounter an error is to halt the execution of the script. Unfortunately, you might want to let the browser continue on executing the other statements despite encountering an error. If you need that to happen, you will need to use try and catch.

Try, Catch, and Finally

Try and catch are two keywords that can allow you to specify a certain block of code and test it if it will generate an error. On the other hand, the finally keyword will allow your program to execute statements within it regardless if the browser encountered an error. Basically, this is how they work:

try {alert(This is a message);}

catch{alert(“The script encountered an error”);}

finally{alert(“I don’t care if there is an error or none. I will still appear!”);}

What will happen is that, when the statement inside the try block gets an error, the statements on the catch block will be executed. If there are no errors, the catch block will be ignored. Alternatively, the statements within finally will still execute regardless of error state.

Console.log

What if you want to see everything in the console? Every value or variable that are getting change, you would want that especially during debugging. To be able to see a lot of information in your console, you can take advantage of logging functionality of console browsers

To print a message on the console, all you need is to provide a value to the Console.log method. For example:

> var x = 1

< undefined

> var y = 2

< undefined

> var z = 3

< undefined

> var a = x + y

< undefined

> var b = y + z

< undefined

> console.log(a)

2015-01-01 00:00:01.001 **3**

< undefined

> console.log(b)

2015-01-01 00:00:02.002 **5**

< undefined

> _

Notification Errors

Notification errors, which show up through IE dialog boxes or through console are the outcomes of runtime errors and syntax. These notification errors include the number line at which the error happens.

If you’re using FireFox, then you can just click the error console to proceed to the precise line in the script with error.

Debugging a Script

Below are the different ways to debug a JavaScript:

Add Debugging Code to the Programs

The methods **document.write()** or **alert()** in the program can debug the code. For instance, you can use the following:

```
var debugging = true;
var whatImage = "gadget";

if( debugging )
alert( "Voice swapCalls() with argument: " + whatImage );

var statusSwap = swapCalls( whatImage );

if( debugging )
alert ( "Voice swapCalls() with argument: " whatImage );
var statusSwap = swapCalls( whatImage );

if( debugging )
    alert( "Exits swapCalls() with swapCalls=" + swapCalls );
```

By studying the order and content of the alert() as they are positioned, you can easily study the status of your program.

Add JavaScript Debugger

A debugger refers to the application, which places all factors of execution script under the command of the programmer. Debuggers offer better control over the script status through an interface, which permits you to study and establish values as well as execution flow.

Once you load a script into the debugger, you can run a single line at a time or commanded to stop at specific breakpoints. When the execution is stopped, you can study the state of the script as well as the variables to know if something is missing. It is also possible to take a closer look at variables for value changes.

JavaScript Validator

Another way to test the JavaScript code for erroneous bugs is to run it through a program, which tests it to ensure that this is valid, and that it follows the right syntax of JavaScript. These are known as validators or validating parsers, and usually added with JS editors and HTML.

The JavaScript Lint, developed by D. Crockford, is regarded as the most convenient validator for JavaScript. This is available for free, and you can just paste the code into the specified text area. Once you click the jslint button, the program will parse through the JS code, which ensures that the function and variable definitions will follow the right syntax. This will also test JS statements like while and if, to make certain that they too follow the right format.

Helpful Tips for Developers

The following tips could help you to reduce the errors in your script as well as to simplify the process of debugging.

- Use indentation to create codes that are easier to read. Also indent the statements to make this easier for you to match up the curvy brackets, starting and ending tags, as well as other HTML and script elements.

- Use a lot of comments, because these will enable you to discuss the reason why you wrote the script and you can also explain the tricky sections of code.

- Keep the consistency in naming your functions and variables. You can try using names, which are long enough and more meaningful. These names should also describe the variable contents as well as the use of the function.

- Use constant syntax in naming functions and variables. To put it simply, keep them all uppercase or lowercase. If you like to use the Camel Back notation, use it all throughout.

- Write codes in modular form. If possible, categorize your statements into functions, which will allow you to group statements that are related to each other, and check as well as reuse specific portions of the code with less effort.

- Check long scripts in modular form. To put this simply, don’t write the whole script before you check any portion. You can try to write a piece and obtain it to work before including the next portion of the code.

- Be mindful of your quotation marks. Take mote that the quotation marks will be used in pairs surrounding the strings and that both marks should be of the same style. Choose single or double and use the same style all

throughout.

- Use function names and descriptive variable and lessen the use of names that are single character.
- Be mindful of your equals signs. Avoid using a single equals signs for comparing other elements.

- Explicitly declare variables through the keyword var.

Chapter 23: AJAX

What is AJAX? AJAX stands for Asynchronous JavaScript and XML. What does it do? Well, one of the things you can do with AJAX is to access other pages, scripts, and/or data without making your browser load another page.

Primarily, AJAX is an important component of creating dynamic content web pages. With it, you can deliver dynamic content and create rich web applications such as chat. You can perform updates on any part of your page, without forcing the whole page to reload. And those are just a few of the things you can do with AJAX.

By the way, if it XML is included in AJAX, it does not mean that you need to learn XML. With HTML knowledge and the things you have learned so far, you can take full advantage of AJAX.

How does it work? Basically, you must create an XMLHttpRequest. That request will be sent to the server. The server will process the request, and a message will be sent back to you. The reply of the server depends on the client and server scripts you created. Also, it will be up to your script to process the reply you receive.

However, despite sounding too simple, achieving rich interactions with your page and your server depends on your ability to create server scripts. Knowledge on PHP, ASP, or other server scripting language is required. Also, if you will be playing with data, knowledge in database management and SQL language will be needed.

Of course, it is possible that you are unfamiliar with how they work as of now since you are only in the beginning of your journey as a web developer (you are still learning client side scripting).

Another reason that learning AJAX will not be fruitful as of now is that you might not have proper server access. Meaning, if you do not have a web hosting account or do not have a server application in your computer or network, you can only do little with AJAX.

Nevertheless, it is still possible for you to use it, and add more functionality and flair to your web documents.

Steps in AJAX Operations

1. Client event occurrence
2. Creation of XMLHttpRequest
3. Configuration of XMLHttpRequest
4. Asynchronous Request of the XMLHttpRequest to the Webserver.
5. The Webserver will return the result, which contains the XML doc.
6. The object XMLHttpRequest will call the function callback() and the result will be processed.
7. Update of HTML DOM.

Chapter 24: JSON

Originally, JavaScript was developed as a page-script language for Netscape Navigator. It is still a common misconception that it is still a subset of Java, but this is not true. This is a scheme-like language with soft objects and C-like syntax.

Meanwhile, the JSON is a subset of the literal notation object of JS. Because JSON is a subset of JS, this could be used in the language easily.

```
var myJSONObject = {"bindings": [
        {"ircEvent": "PRVTMSG", "method": "newURI", "regex": "^http://.*"},
        {"ircEvent": "PRVTMSG", "method": "deleteURI", "regex": "^delete.*"},
        {"ircEvent": "PRVTMSG", "method": "randomURI", "regex": "^random.*"}
    ]
};
```

In the example above, the object is constructed that contains a one member bindings. This also contains array confining three objects such as regex, method, and ircEvent members.

The members could also be restored using the subscript or dot operators.

```
myJSONObject.bindings[0].method    // "newURI"
```

In order to change a JSON text into the object, we can use the function eval(), which calls the JS compiler. Because JSON is a formal subset of JavaScript, the compiler will parse the text and will yield the structure of the object. Meanwhile, the text should be confined in parentheses to prevent tripping on the ambiguity of JS syntax.

```
var myObject = eval('(' + myJSONtext + ')');
```

The function eval is fast to use. But this could compile and run any JS program, so there could be security concerns. Using eval is signified if the source is competent and trusted. It is a lot more secure to use a JSON parser. In online applications over XMLHttpRequest, communication is allowed only to the same origin, which provides that page, so this will be trusted. But this might not be efficient. When the server is not thorough in encoding JSON, or if this doesn't validate all the inputs, then it may yield invalid JSON text that can also carry harmful script. The function eval will also run the script, which will unleash the

bad script.

To improve your defense, you can use a parser, which will interpret not only JSON text, but will also reject other scripts. In web browsers that offer support, parsers are a lot faster compared to eval. Of course, JSON is also included in the ECMAS standard.

```
var myObject = JSON.parse(myJSONtext, reviver);
```

The alternative parameter reviver is a function, which you can call for each key and value at each level of the final outcome. The result of the function reviver will replace every value. You can use this to change generic objects into occurrences of pseudo-classes or to change the date strings into objects Date.

```
myData = JSON.parse(text, function (key, value) {
    var type;
    if (value && typeof value === 'object') {
        type = value.type;
        if (typeof type === 'string' && typeof window[type] === 'function') {
            return new (window[type])(value);
        }
    }
    return value;
});
```

The JSON stringifier will go in the opposing direction, changing the JS data structures into text. JSON will not provide support for cyclic data structures. Hence, be sure that you are careful in not providing cyclical structures to the stringifier.

```
var myJSONText = JSON.stringify(myObject, replacer);
```

If the method stringify sees an object, which contains the method toJSON, it will call that method, and will stringify the returned value. This will allow the object to identify its own JSON representation.

The method stringifier may take another string array, which are used to choose the

properties that will be added in the JSON text.

The method stringifier could take an optional function replacer. This will be called after toJSON method (if there's one) on every of the values in the structure. This could be passed every key and value as parameters, and this could be confined to object that holds the key.

The values that are not represented in JSON such as undefined and functions are not incuded.

Numbers that are non-finite are replaced using null. In order to substitute other values, you can use a function replacer such as this code:

```
function replacer(key, value) {
    if (typeof value === 'number' && !isFinite(value)) {
        return String(value);
    }
    return value;
}
```

Chapter 25: jQuery

Before the development of jQuery, web developers had to make their own JS frameworks. This permitted them to work around certain bugs without spending too much time to debug the usual features. This resulted to developers making JS libraries, which are free to use.

Basically, JQuery is a particular library of JS code. There are other JS code libraries like MooTools. However, jQuery became widely used because it is simple to use and very effective.

Even though many developers are still confused of jQuery and JavaScript as two independent languages, it is crucial for you to know that these are both JS. The main difference is that the jQuery could be optimized to perform other typical scripting functions and it could still do while using less code lines.

Many web developers have been debating if jQuery or JavaScript should be used in a certain situation. But there is actually no correct answer. You can use either language to create the same results, but usually jQuery could do this with less code lines.

In general, jQuery is enough for majority of web development projects. There will be some projects, which require conventional JavaScript. But they are becoming quite rare. Even though jQuery could be the better option in most settings, as a starting developer, you must still learn both jQuery and JavaScript.

Even though using JavaScript alone could slow down the completion of your project, it is crucial to know how this language works and how it could affect the DOM (Document Object Model).

Bear in mind that the main difference between JavaScript and jQuery is that the latter has been optimized to work with different browsers easily. Unluckily, JS still has some concerns with cross-browser compatibility because of poor JS implementation strategies

of developers.

Chapter 26: JavaScript in Bootstrap

For JavaScript in Bootstrap, plugins could be individually included, although some have called for dependencies, or all together. Both bootstrap.min.js and bootstrap.js contain all plugins in one file.

Data Bootstrap Attributes

It’s okay to use Bootstrap plugins completely through the markup API without encoding one line of JavaScript. This is the first class API Bootstrap and should be your main consideration in using a plugin.

With this, in certain situations, it could be ideal to shut off this functionality. Hence, we can provide the capacity to disable the attribute of the data API by unbinding the events on the namespace of the body using the data API.

```
$('body').off('.data-api')
```

On the other hand, to target a certain plugin, you can add the name of the plugin as a namespace alongside the data-api such as this:

```
$('body').off('.alert.data-api')
```

You can also use Bootstrap plugins completely using the JavaScript API. Take note that all public APIs are chainable, single methods that could yield the acted collection.

```
$(".btn.danger").button("toggle").addClass("thin")
```

The methods must be receptive of the objects that are optional – a string that targets a certain method or nothing that initiates a plugin using a default behavior:

```
$("#firstModal").modal()                    // initialized with defaults
$("#firstModal").modal({ keyboard: false }) // initialized with no keyboard
$("#firstModal").modal('show')              // initializes and invokes show immediately
```

Every plugin will also show the raw constructor in the property ‘Constructor’

```
$.fn.popover.Constructor
```

If you want to obtain a certain plugin occurrence, you can directly retrieve it from an element.

```
$('[rel=popover]').data('popover')
```

There are times that it is crucial to use Bootstrap plugins using other UI frameworks. In these instances, the collisions namespace could occur occasionally. When this happens, you can call the noconflict in the plugin that you want to revert the value.

Events

Bootstrap offers customized events for many of the special actions of most plugins. In general, these could come in past participle form and infinitive, where the infinitive will be triggered at the beginning of the event. Meanwhile, its past participle form will be triggered when the action has been completed.

All the infinitive events offer the functionality preventDefault, which offers the ability to cease the implementation of the action before it begins.