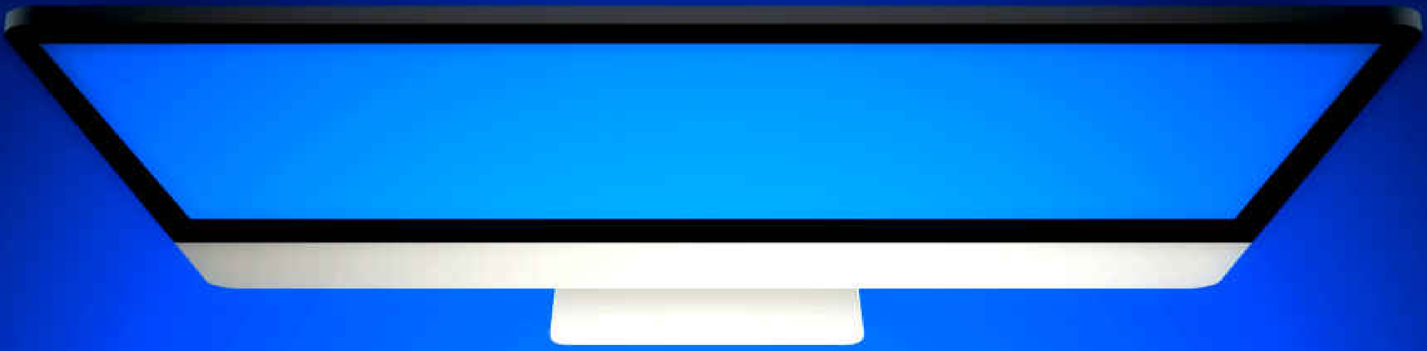


JAVASCRIPT CRASH COURSE

The Ultimate Beginner's Course to Learning Javascript Programming in Under 12 Hours

EPROGRAMY

JAVASCRIPT CRASH COURSE

The Ultimate Beginner's Course to Learning JavaScript Programming in Under 12 Hours

By Eprogramy

Table of contents

Introduction

Welcome to Your New Programming Language

So, you've decided to learn JavaScript Programming? Well, congratulations and welcome to your new Programming Language! You're going to love it!

In Eprogramy we believe that the foundation of a good education is to provide the tools able to open the doors of the future. It is indisputable that the world has entered an era in which we all have everything at the fingertips. Social needs have changed. Thus, today communication has changed. Communication opens doors, so it is our challenge to learn how to communicate. In Eprogramy we want you to learn how to communicate with a new language: The programming language. This language allows us to use our PC or notebook to create.

Maybe the road is difficult and hard, but we will give the reader the tools to make learning enjoyable and fruitful. We'll show all aspects necessary to learn how to program. From the ABC´s to the solution of the most common programming problems and much, much more. Always with the help of numerous examples that will contribute to a better understanding

We know what we do and believe that you are fully capable of incorporating our teachings.

The doors of the future are here. Let's go step by step together.

Let´s get started!

Eprogramy Team

Chapter 1

JavaScript Programing Language

History of JavaScript

Since the inception of the World Wide Web, websites contained static pages. Upon entering different web pages, what was being viewed was in the most basic form allowed in its time (early 1990s). The interaction of a page was not possible, but now it finally is, with the programming language you will be learning in this guide, Javascript.

Through the use of Javascript, you will be able to interact with page components like buttons, text fields, lists, tables and much more by having an infrastructure set up to listen for interaction of user mouse clicks or keyboard strokes and to respond accordingly depending on what is programmed for it to do. The whole idea of Javascript is to be able to give instructions to your web page to react to different things or do things in certain orders depending on time, actions, and various other elements that can be used to trigger javascript events.

The two popular browsers, the program you use to access web pages on the World Wide Web, were the Netscape Navigator and Internet Explorer. The Netscape browser was the first to come out with a programming language to give the ability for users to make their web pages interactive. This was named “Livescript” in its time but soon changed to “Javascript” when another programming language with the name of “Java” came out and they found a good opportunity to use its name as a way to make their language more reputable.

Although, in 1996, the Javascript programming language was handed to an international standards body with the name of EMCA who handled the language from then on.

What is JavaScript?

Let’s clear a few misconceptions about Javascript - it is not Java. Many people tend to get the two mixed up and don’t entirely understand the difference. The two languages are different not only in concept, but design and should be viewed as two completely different programming languages.

Javascript is a programming language most commonly used in website interaction. It can be used to make web pages dynamic and interactable, to the point where even games can be made from Javascript and HTML5! Javascript code is either placed inside an HTML document (HTML is the format all web pages are written in, or at least the format web pages are translated as by your browser) or it is referenced from an HTML document. If you want more understanding of this, go find an HTML tutorial.

The way Javascript is executed is not through any special third party program, like through a compiler, if you’ve ever studied an application level based programming language - but rather using the Javascript built in support from major web browsers like Chrome, Firefox, Internet Explorer, and more. Therefore, you simply have to embed Javascript code within the HTML language, which is a standard language for browsers to interpret web-page content like text and appearance (with the help of CSS).

The tools needed to be able to write Javascript code can be as simple as a notepad to getting started. For this guide, we will be using Sublime Text, which is a great text-editor tool for web-development languages like HTML/CSS and Javascript. Another alternative text-editor if you don’t like Sublime Text, is Notepad++ which is also another great tool for writing code pertaining to web-development.

Chapter 2

Important Things to Understand Basics of HTML and CSS

HTML

In order to understand how we can implement the Javascript language and work with it, we must understand the fundamentals of how a web page in its most basic form works. HTML stands for HyperText Markup Language. It is the standard mark up language used to create static web pages. That is why, with the help of Javascript, we can make these web pages more dynamic and responsive.

Tags

In HTML, there is a standard convention to starting a statement, and ending one. In English, we must start our sentences with a capital letter and end them with a period to maintain a standard flow of writing which is easy to interpret. Well, HTML uses the idea of "Tags" to be able to understand what exactly you are writing. There are a few vital tags you must remember in HTML, and understand what they are used for.

In any HTML file, there is a standard structure that must be followed. Here is sample code showing an example of it:

```
<html>

        <head>
                <title> This the webpage title </title>
        </head>

        <body>
                This is the content of the webpage.
        </body>

</html>
```

When you are making a tag, you must start it with a “<” symbol and then indicate the name of the tag, and then end it off with “>” symbol. Then, the “inner” portion of the tag, or within the tag, goes the rest of the code. To end the tag, you do the same convention as starting the tag, except end it with a “/”. This is why you see the top of document as: <html> and the end of the document as </html>.

Visual demonstration:

<tagname> content within tag </tagname>

Now let’s analyze the sample code down:

<html>

This is the tag that indicates when the HTML portion of the site will be, so it must be presented in every HTML document. The tag is then closed at the end of the document. Therefore, the <html> tag is the parent of all the tags within it, the children.

<head>

This is the tag that would hold all the header information when loading a web-page. Everything within this tag will not be shown to the user, and is mostly used to initialize scripts to ready them for use. An example would be to set a script tag in the head section to prepare for use for the rest of the HTML document. We will have to do this when learning about the jQuery library.

<title>

The contents of this tag will hold the title of the webpage.

<body>

This is the whole content part of the website. Just line in any essay, you have your thesis, body and conclusion. This is the same idea in the form of tags.

There are many other tags in HTML that you could look up online. Let’s look at a variety of examples and identify some helpful tags.

```
<html>

        <head>
                <title> Example of “<p>” tag </title>
        </head>

        <body>
                <p> This is a paragraph. </p>
        </body>

</html>
```

The bolded tag in the above example essentially describes a paragraph and formats it in such a way where it has proper padding/spacing without you having to format it yourself through the use of CSS (ability to style HTML elements).

```
<html>

        <head>
                <title> Example of “<p>” tag </title>
        </head>

        <body>
                <h1> This is a heading. </h1>
        </body>

</html>
```

This <h1> tag used in the above example simply bolds the text in between it and enlarges it. All properties of tags that you will be using are editable through CSS which we will be going through soon.

Here is an example using both tags that we have learned and showing the output. Note that once you have created a file with HTML in it, you must save the file with a .html extension. You can then open the file and test it out by opening with a browser of your choice.

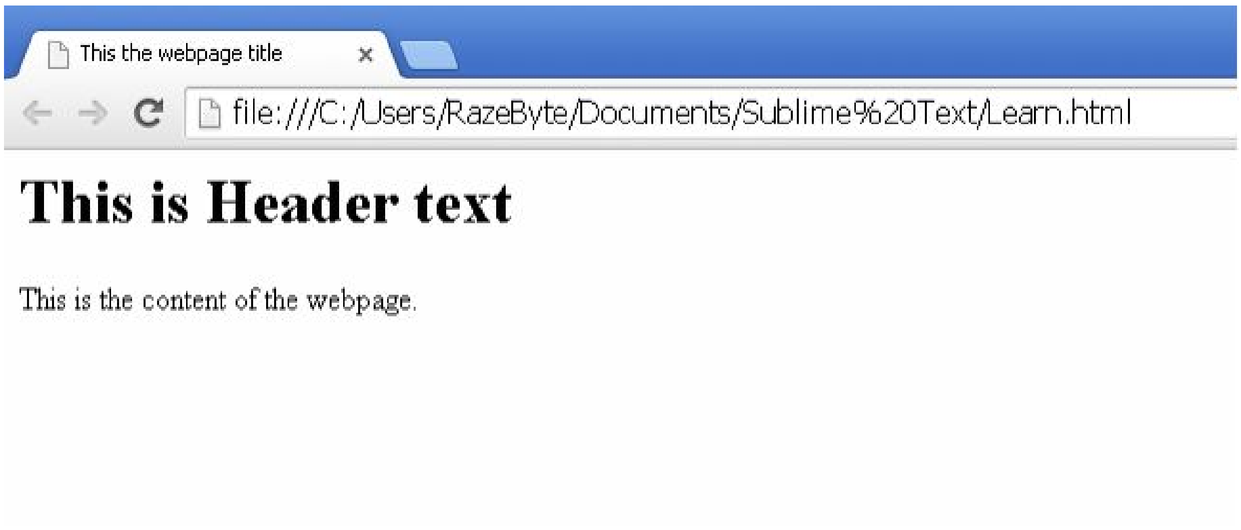
```
<html>

        <head>
                <title> Example of "<p>" tag </title>
        </head>

        <body>
                <h1> This is a heading. </h1>
        </body>

</html>
```

Output


This the webpage title
file:///C:/Users/RazeByte/Documents/Sublime%20Text/Learn.html

This is Header text

This is the content of the webpage.

Finally, let's go over the idea of commenting in HTML. Sometimes you might want to code something but not exactly print it to the screen, but just want to explain a statement that you are doing or adding a reminder in the code without having it translate visually on the web browser so that it won't be able to see it.

You can comment by starting your comment with <!-- and ending it with →

Therefore, if I wanted to comment a certain line, I would do the following with a functioning paragraph:

```
<p> This is paragraph text </p>
<!-- This line prints out a paragraph that says “This is a paragraph text” -->
```

There are tags, and then there are attributes, which contain values which are then used to determine how the tag's contents are displayed or are interpreted by CSS or Javascript. Two important attributes to note when going in to Javascript and CSS are: id and class.

You can use these on any appropriate tag like the <p> tag within the <body> tag.

The format of how an attribute works is like the following:

```
<p id = "content"> This is the content of the webpage. </p>
```

As shown in the example, in order to declare an attribute, you must make space within the first tag which is <p> and then name the known attribute, so id for example. Then to give it a value, you must have an equal sign and then two quotation marks and the value within the quotation marks. In this case, this paragraph, contains the attribute id which is the value of “content” - this can be used as a reference for languages like CSS and Javascript through the use of selectors which you will be learning about.

There are still many tags that you can learn about in HTML but these are the basic tags that you need to know to start writing javascript. Also, you should now understand the basic structure of an HTML document.

Let's review:

The tag **<html>** and **</html>** defines the HTML document, and the tag itself simply defines the content within it as HTML type so the browser is able to interpret it.

The tag **<head>** and **</head>** provide information about the document and initialize any external scripts or styling sheets.

The tag **<title>** and **</title>** simply give information to the browser to display a title of the HTML document.

The tag **<body>** and **</body>** holds all the content of the document which includes the paragraphs, navigation bars, tables, images, etc.

The tag **<h1>** and **</h1>** simply formats text in between it as a header or a title to a paragraph.

The tag **<p>** and **</p>** formats text in between it as a paragraph accordingly.

CSS

CSS stands for "Cascading Style Sheets" and essentially styles how HTML elements are displayed on the web browser.

Now, before we move on, let's quickly go over how commenting works in CSS. Just like in HTML, commenting is useful, and can also be done in CSS, but it has different syntax for it. In order to comment in CSS, you must use the following notation:

/* This is a comment */

The text in between the slash and symbol is not processed and is simply there for human reading. So to begin a comment you must do /* and to end a comment you must do */.

Let's scan through the idea of how to write CSS code.

Example:

```
p {
        color: red;
        font-size: 14px;
}
```

In the above example, there are a few critical things to note of the format in the above CSS code.

The element we must specify when writing CSS code is the tag name or the selector.

```
p {
        color: red;
        font-size: 14px;
}
```

The bolded text indicates the selector, which is the <p> tag, which handles the paragraphs of the document. If you place any text in between <p> in the body, you will now see the text being red, and the font size being 14 pixels. Let’s move on to analyzing this code further.

```
p {
        color : red;
        font-size : 14px;
}
```

The bolded text indicates the property in the above example. In every selected tag, you must indicate properties of that tag, like the color of the text, the size of the text, the style of the text, the font-weight and much more. You can always look for any property that you’re wanting on the internet. There are many properties and we won’t be going over them in this guide. There are different properties that are applied to different tags. For the “p” tag in this instance, we can reference the color property and the font-size property. Upon referencing them within curly braces (which simply indicates the block of properties that applies to the selector which is p) and then assigning it to a certain value.

```
p {
        color: red ;
        font-size: 14px ;
}
```

To assign a certain value to the property of color, you must make a colon symbol which simply separates the property and the value, and then type out a value accordingly. Once you have written out the property and the value, use the semi-colon as a way to let the browser know that you have ended that statement so it can read the next statement.

When working with colors, instead of using the standard red, blue, white, yellow, etc - You can reference more specific colors depending on the RGB values (Red, Green, Blue) which can be obtained using an HTML color selector. This is a link of a site that can give you HTML color codes for use in your CSS.

Link: http://html-color-codes.info/

To review CSS syntax let's look at it in the following approach:

1. p
2. {
3. color:
4. red;
5. }

The code above takes the color of the paragraph and changes the color to red. Each line has been numbered, but wouldn't be included in the code. It is numbered so we can scan through each line and go over how it works.

1. Selector/Tag

2. Indicating the start of the selector properties

3. Property/Attribute name with a colon to prepare to indicate the value

4. Value of the Property/Attribute and then a semi-colon to indicate the end of statement

5. Indicating the end of the selector properties for the certain selector which was p

Remember, you can do this as many times, just remember to separate the selector blocks so the browser doesn't get confused. Well, let's first go over how you can actually start coding in your HTML document with CSS. To begin, you must go in between your <head> </head> tags and input the following:

```
<html>

        <head>
                <title> This the webpage title </title>

                <style type = "text/css">
                        p {
                                color: red;
                                font-size: 14px;
                        }
                </style>

        </head>

        <body>
                <h1> This is Header text </h1>
                <p id = "content"> This is the content of the webpage. </p>
        </body>

</html>
```

In order to start coding CSS in an HTML document, you must use the "style" tag with the help of the "type" attribute which references to the standard convention "text/css" which you must use every time you want to write CSS in your HTML document. Within this style tag, is your CSS code which references successfully to all p tags in the HTML document. Well, what if you'd like to only reference a specific paragraph?

Now the reason I like to call p as a selector instead of a tag is because the term selector is a general term and what can be put in the position of p is also general. Instead of putting an actual tag that is being referenced in the HTML document, you are able to reference an id by using a hashtag and then the id name accordingly:

```
#idname {
        color: red;
        font-size: 12px;
}
```

You can also use the selector to reference a tag with a "class" attribute with a certain name. For example:

```
.classname {
        color: red;
        font-size: 12px;
}
```

Now you must be wondering why id or class could ever become helpful. Let me enlighten you with the following sample code:

```
<html>

        <head>
                <title> This the webpage title </title>

                <style type = "text/css">
                        #content {
                                color: red;
                                font-size: 14px;
                        }
                </style>
```

```
    </head>

    <body>
        <h1> This is Header text </h1>
         <p id = "content"> This is the content of the webpage.
</p>
    </body>

</html>
```

In the above example, you can see that we listed a specific paragraph to be affected by the change we listed in the CSS within the head tag. This is one way that you can use the id selector. To use the class selector (not really a big difference, just two preferences) is to do the following:

```
<html>

    <head>
        <title> This the webpage title </title>

        <style type = "text/css">
            .content {
                color: red;
                font-size: 14px;
            }
        </style>

    </head>

    <body>
        <h1> This is Header text </h1>
            <p class = "content"> This is the content of the
webpage. </p>
    </body>

</html>
```

In the code above, instead of using a hashtag symbol to select a specific attribute with a specified id name, but rather a class name this time. Using this knowledge of selectors, we can reference specific HTML elements using Javascript and use that to manipulate the tags, by adding attributes or changing the content within the tags and making them dynamic to user interaction.

In CSS, there are different ways to actually implement the styling language into the HTML. I have already showed you one way, which is to have it in the head tag with a style tag. Well, there are three main different types of ways to implement CSS, one way is inline, other way is internal, which is the way we did it and the other way is to do it externally. The way we did it directly in the head, internally. To make it external, instead of having the CSS code within the tags, we have the style tag reference a file that contains all the CSS code.

Let's simply look at examples of how each way works. Remember, if you reference a certain file, like a CSS one in this instance, if you don't list the direct directory path (e.g. C:\Program Files (x86)\Crazybump) and just the name, then it will resort to looking at the root directory your HTML file which is referencing is in.

External CSS

Filename: *Learn.html*

```
<html>

        <head>
                <title> This the webpage title </title>

                                        <link rel="stylesheet" type="text/css"
href="pageStyle.css">
<!-- This is the conventional line used in the head tags to reference an
external CSS file. The href attribute is used to reference the specific file
and is the only one you will change from this line for your CSS file. -->
```

```
        </head>

        <body>
                <h1> This is Header text </h1>
                <p id = "content"> This is the content of the webpage. </p>
        </body>

</html>
```

Filename: pageStyle.css

```
#content {
        color: red;
        font-size: 14px;
}
```

Now, since we’ve already seen how internal CSS and external CSS referencing works, we can look at the cheapest way to do CSS: Inline. This way is not recommended just because it isn’t neat or organized to look at. Although it is useful when testing a few things with a specific tag. In order to do inline, for any tag you want to style, you must add the attribute “style” with an equal sign and then quotation marks for all the properties. Then for each property, you must have the value separated by a colon and when you are done writing down all your properties, you must end it off with a semi colon. Here is an example:

```
<p style = "color: red; font-size: 14px" id = "content"> This is the content of the webpage. <p>
```

This above is an example of inline coding. Here is the full source code:

```
<html>

        <head>
                <title> This the webpage title </title>
        </head>
        <body>
                <h1> This is Header text </h1>
                 <p style = "color: red; font-size: 14px" id = "content">
This is the content of the webpage. </p>
        </body>

</html>
```

We have finally went over the fundamentals of HTML and CSS and can finally get started on understanding how to implement Javascript!

Chapter 3

JavaScript Language Structure

The way Javascript works is that it can be placed in the <body> and the <head> sections of an HTML document. This can be done either externally or internally depending on your preference in the situation. In order to insert javascript code internally into the HTML document, you must use the <script> tag.

Example:

```
<script>
        alert("Javascript test");
</script>
```

Another way to do this is to specify the script type:

```
<script type = "text/javascript">
        alert("Javascript test");
<script>
```

To go over what exactly is happening, we simply start with a script tag, and within it, code the Javascript code. In this case, we are using the build in javascript function called "alert" and it simply prints out what is written within the quotation marks. One thing to note is that in programming in general, whenever you're working with text (or strings), you must use quotation marks to indicate that it is indeed text and not anything else or else the interpreter would get confused when processing the JS code.

The way the script tag can be used is in the following ways:

```
<html>

        <head>
                <title> This the webpage title </title>
```

```
                    <script type = "text/javascript">
                            alert("Javascript test");
                    </script>
        </head>

        <body>
                <h1> This is Header text </h1>
                    <p style = "color: red; font-size: 14px" id = "content">
This is the content of the webpage. </p>
        </body>

</html>

        OR

<html>

        <head>
                <title> This the webpage title </title>
        </head>

        <body>
                <h1> This is Header text </h1>
                <script type = "text/javascript">
                        alert("Javascript test");
                </script>
                    <p style = "color: red; font-size: 14px" id = "content">
This is the content of the webpage. </p>
        </body>

</html>
```

As seen in both cases, you can either choose to use it in the head tag or the body tag. Another thing to note is the fact that you can place any number of scripts in an HTML document. Now as we know, there is an internal way of adding javascript and there is also an external way of doing it.

In order to do it externally, you must use the script tag except in this case, the script tag is fed the “src” attribute which simply would hold the value of the directory path to the js file that you are referring to.

For example:

```
            <script src = “test.js”></script>
```

In this certain case, the script will refer to a js file of name “test” within the root folder since no directory was supplied. In order to actually create test.js - all you would do is write the raw Javascript code and then save the file with a js extension.

Here is a full example with the HTML document:

Declaration within <head></head>:

```
<html>


            <head>
                        <title> This the webpage title </title>
                  <script src = “test.js”></script>
            </head>

            <body>
                        <h1> This is Header text </h1>
                              <p style = "color: red; font-size: 14px" id = "content">
This is the content of the webpage. </p>
            </body>

</html>
```

OR

Declaration within <body></body>:

```
<html>
```

```
	<head>
		<title> This the webpage title </title>
	</head>

	<body>
		<h1> This is Header text </h1>
	   <script src = “test.js”></script>
			<p style = "color: red; font-size: 14px" id = "content">
This is the content of the webpage. </p>
	</body>

</html>
```

The advantages to having external javascript code is that cached Javascript files are able to load up faster and thus increase page-loading speeds. Another great thing about using external Javascript files is that it is easier to maintain, especially for large projects.

Now let’s go over a few conventional things to understand about programming in general. The spacing in code doesn’t matter at all in terms of it working or not, but it is conventional and efficient to have properly spaced code so other programmers can read your code and so that you can read your code yourself when you’re coding. As you continue to program and practice, you’ll start to get the hang of understanding the standard spacing conventions and when you need to do it. The best way to do it is to simply continue reading this guide and notice when there is spacing.

In programming, you might see a name and then curly braces like “{“ or “}” - all these do is communicate with one another. Think of a tag in an HTML document like <tag> and then when it ends, it would do </tag> and everything in between the tag would be the content that would be processed according to the function of the tag. Well the same applies to functions in programming (we’ll learn more about this later) - The first curly brace “{“ would be used to introduce the beginning of the block of code and the ending curly brace “}” would be used to enclose the block of code. This might not make sense right now but you will soon realize how this all makes sense when you're learning about functions and classes.

Another fundamental thing to understand in programming the is the use parentheses - which are used to intake arguments or variables, which would then be used within the function to work with. For example, if you remember functions in high school, you will remember what this would mean:

$f(x) = 5x + 10$

When the x value is replaced with an input like 2, it would then become:

$f(2) = 5(2) + 10$

$f(2) = 20$

Now if we applied the same idea with functions in programming, we can see that all it is doing is providing the function the capability to intake values to work with. Again, this may not make sense, but I'm just preparing you for when we go over classes and functions and how their structure will work.

We have went over commenting in HTML and CSS, and surprise - it's also in Javascript! Remember, comments are not executed and the browser would ignore those lines. Writing anything after double slashes would simply be a one line comment.

For example:

```
// This is a comment
```

Now the next way to comment is to use /* and */ in which the comment would be placed in between those two symbols, the /* being the first and */ being the last. This would be considered a multi-line comment.

For example:

```
/* This is a multi-line
        comment. */
```

Just like writing a sentence in English, understand that when ending a statement, you use a period - well the same convention applies in programming. Remember to place a semi-colon after any statement you create in Javascript. There are a few exceptions, like when you are creating a function or a class, but this will be discussed later in the guide.

The final topic to cover in the Javascript language structure is case sensitivity. Everything in Javascript is case sensitive meaning if you create a variable (you will be learning about this in the next section) or a function, the names will have to be addressed in the exact format they were written in, not a capital letter or lowercase letter different.

Chapter 4

JavaScript Variables

A variable is essentially a storage unit that holds a certain type of data. It is named by the programmer, and is used to identify the data it stores. A variable can usually accessed or changed at any time. You can write information to it, take information from it, and even copy the information and add it to another variable. In every programming language, variables hold a specific data type, whether that be a string (text) or a whole number (integer) or a decimal number (float/double), these are things to consider when you are programming. There are many different data types like numbers, strings, arrays, objects, and more.

In order to declare a variable, you must use the keyword “var” and then the name of the variable afterward (you can name this anything you want). Remember that everything in Javascript is case sensitive. The use of variables is to hold a value so it can be used for later use or manipulation in the code. Let’s go over a few examples of variable declarations:

```
        var numberOfDogs = 16;                                   //
number
        var nameOfDog = “Copper”;                          // string
        var hasDog = true;                                       //
boolean
        var cars = [“First”, “Second”];          // array
        var anObject = {myName:”Something”};          // object
```

You might not totally understand what is going on in these statements like the array or object, but it is easy to grasp the number and string variable declaration. For the numberOfDogs variable, we are declaring a variable by using the “var” keyword and then typing out the name of the variable. In this case, the variable name is “numberOfDogs” which is then assigned to the value of 16. This is done by the use of an equal sign which essentially signifies what it’s meant to signify - it is the assignment operator which assigns values to the array.

The next variable is “nameOfDog” which is equal to Copper - although the dog’s name is surrounded by quotation marks - this is used to tell the browser that it is a piece of text and not anything else. If Javascript detects a number based on the fact that it doesn’t have any quotation marks around it, then writing text with a number in it, would make sense for quotation marks to be added to implicitly tell the interpreter that it is indeed text that we’re trying to assign and not a name and a number like “Copper9” - this is why strings are surrounded with quotation marks to signify that it is indeed text that should not be viewed as anything else.

The next variable is important because you will be seeing this used a lot in programming. A boolean is simple a data type for variables that will only hold a value of true and false. This is used a lot in programming as you can have your program compute algorithms that decide on things, and then declare a certain variable true or false depending on the current situation and adapt accordingly. Long story short, booleans are a big deal in programming and there is a whole logical backend that must be explained for it, which will be the case later on in this guide.

The next two variables look confusing and will be explained on a later part of this guide which covers arrays and objects, but were listed just to note that they are two of the four main data types that are used in Javascript. Do note that there are more than just these four data types, but these are the ones you will encounter most of the time or work with. Now let’s go over important variable operators so we can see how working with variables works in terms of manipulation in arithmetic before we continue on with variables.

Chapter 5

JavaScript Operators

Just like in Math class, you learned about addition, subtraction, multiplication, division and more. These are all arithmetic operators, that are used within programming to intake numbers, process them, and calculate them accordingly. Let’s go over these operators in programming, as they are one of the most important things to understand and also one of the easiest to grasp.

When opening the HTML file in your browser, you should see a black page. Most browsers should have a console of some sort, usually activated by the F12 key. This will be useful while we learn to program. So let's start that! Firstly, always put your code between the script tags. Next, the console.log() function allows you to see the value of any expression or object between the parenthesis. For now, you simply need to know how to use that, but later I will explain what that actually means. So here's some example code:

```
console.log(7);
```

This will print "7" to the console. Notice the semicolon. It is important. Do not forget it.

I've divided up programming into 4 main groups: expressions, statements, keywords and blocks. I’ll begin with expressions. Expressions are mathematical steps that computers do to create a single value. Essentially, they work exactly like in math class, although you need to know the characters for certain things. In Javascript, there are 5 operations. Addition(+), subtraction(-), multiplication(* that is an asterisk by the way), division(/ that is a slash), and modulus(% that is a percent sign). Most of these should be familiar, although modulus is not too well known. Modulus means to get the remainder of a division. Here are some examples:

```
6 % 3 = 0          7 % 3 = 1
8 % 3 = 2          9 % 3 = 0
```

This can be useful to find out if a number is even. Simply modulus by 2 and if it is 0, the number is even, otherwise it is odd. This can also be useful when trying to find the number of minutes, like on a clock. Simply modulus the time in minutes by 60 (60 minutes in an hour) and the result will be the minutes counter of a clock. Notice that when using modulo, you can only get numbers from 0 to any number lower than your operator (which can also be a decimal).

Other mathematical operators like exponents, roots, rounding, etc, are not default operations and are actually functions which we will discuss later. Be aware, that expressions follow BEDMAS (or PEMDAS, whichever you were taught). Either way, here are some expressions and their values.

```
7 * (3 % 2) =  7
(8 - 1 + 3) * 6 - 8 / (0 - 4) = 62
```

If you would like to see the computer do this, you can print it out using console.log, like this:

```
console.log(7 * (3 % 2));
```

The number 7 should appear in the console. That is expressions. There are also, object expressions, but that will be discussed later.

Keywords is the next division of programming. There are many, which we will get around to as we learn different parts of programming. Some common ones are “var”, “for”, “while”, “return”, “function”, and “break”. The computer looks for these special words to perform specific actions within your code. That is a very generic description, but they have a very broad range. Be wary of writing them down when you do not mean to use it. If used improperly, it can not only cause mistakes in code execution, but can simply make your code not run at all due to an error in syntax.

Statements are the next division of programming. Statements always end in a semicolon (except in very specific circumstances). Notice that console.log ends in a semicolon, and therefore is a statement. This is indicative of the fact that console.log is a function, but we'll get to that.

Let's begin with a very common and important statement. This has to do with variables. Variables are values that are stored inside memory for reuse later in your code, often several times. Variables let us give names to stored values. Imagine variables like a box, specifically a moving box. When you get the box out, you write the name of it on the side of the box so you can know what should be inside the box. This is called variable declaration. You are pulling out memory (the box), giving it a variable name (writing on the side of the box), and also telling it where the box should be, or rather what kind of "scope" the variable has, but we'll come back to that.

The Arithmetic Operators

Addition

```
5+5 = 10
var x = 5;
var y = 5;
var sum = 0;
sum = x + y;
```

In the example above, a variable of sum is taking the addition of two variables (x and y) and adding them together to produce the value of 10.

Subtraction

```
10-5 = 5
var x = 10;
var y = 5;
var total = 0;
total= x - y;
```

In the example above, a variable of total is taking the subtraction of two variables (x and y) and subtracting them together to produce the value of 5.

Multiplication

```
5*4 = 20
var x = 5;
var y = 4;
var total = 0;
total= x * y;
```

In the example above, a variable of total is taking the multiplication of two variables (x and y) and multiplying them together to produce the value of 20.

Division

```
20/5 = 4
var x = 20;
var y = 5;
var total = 0;
total= x / y;
```

In the example above, a variable of total is taking the division of two variables (x and y) and dividing them together to produce the value of 20.

Modules

```
7 % 2 = 1
var x = 7;
var y = 2;
var total = 0;
total= x % y;
```

In the example above, a variable of total is taking the remainder of two variables (x and y) and by finding how many times 2 multiplies in to 7 evenly before it can't, and the remainder is the output. For example:

How many times does 2 get in to 7?

3 times.

```
2 * 3 = 6
7 - 6 = 1
```

Therefore, 7 modules 2 is equal to 1 because that is the remainder.

Post-Incrementation

```
5 + 1 = 6
var x = 5;
x++;
var y = 0;
y = x++;
```

In the example above, a variable of x is being incremented by the value of 1. Therefore, the value of x must now be 6 because x + 1 = 6 as x = 5. If you were to print out the value of x, the output would be 6. You can do x++ again and the value would then become 7. The value of y on the other hand is still 6 because it is post incrementing meaning that the value of x has changed by an increment of 1 (to 7), but the value of y has yet to be changed until it is called again (y = x++) which in the case would be y = 7.

Pre-Incrementation

```
5 + 1 = 6
var x = 5;
++x;
var y = 0;
y = ++x;
```

In the example above, a variable of x is being incremented just like it was when it was post-incrementing as a variable by itself. The value of y on the other hand changes directly to 6 because it is pre-incrementing meaning that the value of x has changed by an increment of 1, and the value of y changes, meaning that y = 7.

Post-Decrementing

```
5 - 1 = 4
var x = 5;
x--;
var y = 0;
y = x--;
```

In the example above, a variable of x is being decremented by the value of 1. Therefore, the value of x must now be 4 because x - 1 = 4 as before x = 5. If you were to print out the value of x, the output would be 4. You can do x-- again and the value would then become 3. The value of y on the other hand is still 4 because it is post decrementing meaning that the value of x has changed by a decrement of 1, but the value of y has yet to be changed until it is called again (y = x--) which in the case would be y = 3.

Pre-Decrementing

```
5 - 1 = 4
var x = 5;
--x;
var y = 0;
y = --x;
```

In the example above, a variable of x is being decremented by the value of 1. Therefore, the value of x must now be 4 because x - 1 = 4 as before x = 5. If you were to print out the value of x, the output would be 4. You can do --x again and the value would then become 3. The value of y on the other hand is 3 because it is pre decrementing meaning that the value of x has changed by a decrement of 1, and the value of y has changed.

The Assignment Operators

The assignment operators are operators that are used when assigning variables values, the most commonly used operator being (=). Here are a list of examples:

Equal Sign: =

```
  var x = 5;
```

In this example, if you were to print out x, the value would be 5 because you assigned the variable x equal to 5.

Add-Equal Sign: +=

```
var x = 5;
  x += 5;
```

In this example, if you were to print out x, the value would be 10 because you are adding the value of 5 on to the value of x. This statement is the same thing as saying x = x + 5 → x += 5.

Subtract-Equal Sign: - =

```
var x = 5;
  x -= 5;
```

In this example, if you were to print out x, the value would be 0 because you are subtracting the value of 5 from the value of x. This statement is the same thing as saying x = x - 5 → x -= 5.

Multiplication-Equal Sign: - =

```
var x = 5;
  x *= 5;
```

In this example, if you were to print out x, the value would be 25 because you are multiplying the value of 5 on to the value of x. This statement is the same thing as saying x = x * 5 → x *= 5.

Division-Equal Sign: - =

```
var x = 5;
  x /= 5;
```

In this example, if you were to print out x, the value would be 1 because you are dividing the value of 5 on to the value of x. This statement is the same thing as saying x = x / 5 → x /= 5.

Modules-Equal Sign: - =

```
var x = 5;
  x %= 5;
```

In this example, if you were to print out x, the value would be 0 because you are finding the remainder in (5/5). This statement is the same thing as saying x = x % 5 → x %= 5.

Another important thing to understand when working with numbers in Javascript is that it matters how you declare your number. If you declare it with a decimal value, when you divide it, it will output as a decimal number if it doesn’t divide by two evenly, while if you declare it with an integer value (whole number) it will divide by two and round down if it divides unevenly.

For example:

```
var numberWithDecimal = 5.5;
var wholeNumber = 5;
```

Now that we have went over all the operators that we can use to work with variables, let’s go over some special cases with variables. First of all, if you would like to join two strings together, the way to do that is to concatenate them together (join them together) and to do that is to simply use the + operator.

For example:

```
var x = “Hello”;
var y = “World”;
var line = x + y;
```

The value of line in the case would be “Hello World” without the quotation marks.

The next special case is when trying to add a string and a number together. So for example:

```
var line = 15 + “test”;
```

The variable of line will now hold the value of “15test” without quotation marks, as it won’t understand why you’re trying to add a number to a string, so it will automatically treat it as the following in the backend:

```
var line = “15” + “test”;
```

Now another thing to consider is that Javascript evaluates an expression from left to right meaning that the order of your variables matter. If you were to do the following:

```
var line = 15 + 10 + “test”;
```

This line would produce: “25test” without quotation marks as it would evaluate the expression from left to right. One thing to consider is that since the two variables in this example are numbers, it will add them, but since the third variable is a string, it will treat the number that was evaluated as a string. Remember, that the rules of operation will follow accordingly with BEDMAS or PEMDAS so you are able to use brackets.

The next special case is if you were to do the following:

```
var line = “test” + 15 + 10;
```

In this certain example, the output would be “test1510” without the quotation marks, because it would evaluate the line as a string type as soon as it sees the first variable being of string data type and then it would evaluate everything after it as string accordingly.

Remember that in Javascript, you can update variables dynamically, so for example:

```
var x; // undefined
var x = 10; // x is a number
var x = “This is text”; // x is now a string
```

Throughout this guide, you’ve been seeing me use double quotation marks for strings - it actually doesn’t matter - you’re able to use single quotes to declare string variables as well. The catch with single quotes and double quotes usage is that if you’re using double quotation marks outside your string variable, and want to use it inside, Javascript will get confused, so in order to make it unconfused, you are to place an escape character: “\” right before the double quotation mark that is confusing the Javascript string variable which will simply bypass that double quotation mark and ignore the “\” when outputting the string. Another way is to simply surround your string variable with single quotes and be able to use double quotation marks within your string variable without confusing Javascript - or vice versa.

Examples:

```
var text = "It’s great to learn!" ;
// Single quote inside double quotes

var text = "Their name is ‘Microsoft’" ;
// Single quotes inside double quotes

var text = 'Their name is “Microsoft”' ;
  // Double quotes inside single quotes
```

Chapter 6

Boolean Logic

A boolean is a variable that holds the value of either true or false - and these variables can be manipulated to logically breakdown conditions to come up with a decision.

If Statements

An **if** statement, simply put, checks to see **IF** something is true, or false. To create an **if** statement, you simply write the word **if** followed by two rounded brackets. Within an if statement is the block of code that will execute if the condition of the if statement is met. **if** statements can also include **else if** statements, and **else** statements. An **else if** statement will run if the **if** statement returns false. However, it also contains its own parameter that must return true. An **else** statement also is called if the **if** statement returns false, the difference between an else statement, and an **else if** statement, is that no parameters need to be true for an else statement to run.

Example:

```
 var a = 10;
 var b = 20;

 if (a == b) {
        alert("a is equal to b ");
 }
else if (a > b) {
        alert("a is greater than b ");
}
else {
        alert("a is not equal to b or greater than b ");
}
```

The operands used in this example ("==" and ">") are explained below. In the following code, we first check to see if "a" is equal to "b". If that is true, we simply print "a is equal to b" and ignore the rest of the code. If "a" does not equal "b", then it goes down and calls the else if. The **else** is due to the first **if** returning false. The **if** part of **else if** is checking its own parameters, which happen to be whether "a" is greater than "b". If "a" is greater than b, then we would have printed "a is greater than b". However, since "a" is less than "b", we go further down and simply call **else** . The **else** runs if all the above parameters return false. It does not require its own parameter to be true. **Note** That each **if** statement can have an infinite amount of **else if** statements which follow it, but only one **else** statement. Also **Note** that you cannot have code between an **if** statement and an **else** statement, because the code would not be able to find the **else** . As well, an **else** statement requires there to be an **if** statement before it, since for an **else** statement to be called, an **if** statement HAS to equal false immediately before it. It is also important to know that it is possible to have an **if** statement INSIDE of another **if** statement. This is called a **NESTED if** statement

Example:

```
        var a = 20;
    var b = 30;

   if (a == 20) {
       if (a < b) {
            alert("a is less than b and it is equal to 20");
       }
   }
```

In this example, we first check to see if the value of "a" is equal to the value of "b". If it is, we then check to see if the value of "a" is less than the value of "b". If this statement also returns true, we can print to the console "a is less than b and it is equal to 20". This can work with as many **if** statements as you would like. This can be useful if completely necessary to check the parameters separately, or if you want to add additional logic between each check.

You will understand fully the way an **if** statement works by the end of this section.

The Relational Operators

Javascript supports 6 different types of relational operators, which can be used to compare the value of variables. They are the following:

== This operator checks to see if the value of two integers are equal to each other. If they are the equal, the operator returns true. If they are not, it returns false.

Example:

```
    var a = 10;
    var b = 10;

    if (a == b) {
            alert("they are the same");
   }
```

This code checks to see **IF** the value of "a" is equal to the value of "b". Since the value of "a" is equal to the value of "b", the value within the brackets will be true. This ends up causing the above code to print out the statement "they are the same".

!= This operator checks to see if the value of two things DO NOT equal the same thing. If two things DO NOT equal the same thing, then it will return true. If they DO equal the same thing, it will return false.

Example:

```
   var a = 10;
   var b = 20;

   if (a != b) {
           alert("they are not the same");
```

```
}
```

This code will check if “a” does not equal “b”. Since “a” does not equal to “b”, the program will print “they are not the same” in the console.

> This operator Checks to see if something is **Greater than** something else. If the value of the variable in front of it is greater than the value of the variable after it, it will return true. If the value of the variable in front of it is less than the value of the variable after it, it will return false.

Example:

```
var a = 10;
var b = 20;

if (a > b) {
        alert("a is larger than b");
}
else {
        alert("a is not larger than b");
}
```

This example checks to see if “a” is larger than “b”. Since “a” is not larger than “b”, this if statement will return false. Instead, it will print “a is not larger than b”, because the code passes the failed **if** statement and calls the **else** statement.

< This operator checks to see if something is less than something else. If the value of the variable before the operator is less than the value of the operator after the variable, then the code will return true; else, it will return false.

Example:

```
var a = 10;
var b = 20;
```

```
  if (a < b) {

  alert("a is larger than b");
  }
  else {
alert("a is not larger than b");
}
```

This example is just like the one before it, however, the operator changed from greater than, to less than. Therefore the **if** statement will return true this time. Since the value of “a” is less than the value of “b”, and the program will print to the console “a is larger than b”.

>= This operator checks to see if something is greater than **Or Equal** to something else. If the value of the variable before it is greater than **Or Equal** to the variable after it, then it will return true. If the variable after it is greater but not equal to the variable before it, it will return false.

Example:

```
var a = 20;
var b = 20;

if (a >= b) {
        alert("a is larger or equal to b");
}
  else {
  alert("a is not larger or equal to b");
  }
```

In this example, “a” and “b” both have a value of 20. Although “a” is not greater than “b”, it is EQUAL to “b”, therefore the statement returns true, and the code ends up printing “a is larger or equal to b”.

<= This operator checks to see if something is less than **Or Equal** to something else. If the value of the variable before it is less than **Or**

Equal to the variable after it, then it will return true. If the variable after it is less but not equal to the variable before it, it will return false.

Example:

```
  var a = 20;
  var b = 20;

if (a <= b) {
        alert("a is larger than b");
}
else
        alert("a is not larger than b");
}
```

This example is identical to the one before it except the operator was changed from >= to <=. Although the operator has changed, the result is the same. Since “a” is equal to “b”, this code will return true, printing “a is larger than b” to the console.

The Logical Operators:

Javascript supports 3 Logical Operators that can be used in your logic, which will often be used in conjunction with Relational Operators.

&& This is known as the logical AND operator. If the operands before and after this operator both return true, then the condition returns true. If both of the operands are false, or one operand is false, the condition will return false. **BOTH** operands MUST be true for the condition to return true.

Example:

```
      var a = 20;
  var b = 20;

if (a == 20 && a == b) {
        alert("a is equal to 20 and b ");
}
else {
        alert("a is not equal to 20 or a is not equal to b");
}
```

In this Example, we check to see if “a” is equal to 20, AND if “a” is equal to “b”. These two conditions will be referred to as operands. Since “a” is equal to 20, and equal to “b”, the statement returns true, and we print to the console “a is larger than b”.

```
        Example 2:
        var a = 20;
  var b = 30;

if (a == 20 && a == b) {
        alert("a is equal to 20 and b");
}
else {
        alert("a is not equal to 20 or a is not equal to b");
}
```

In this next example, we check the exact same thing as the first example. The difference this time, however, is that we changed the value of “b” from 20 to 30. This means that when we run this code, the **if** statement will return false because although “a” is equal to 20, it is not equal to “b”. Therefore we print out “a is not equal to 20 or a is not equal to b”. Take note that the order does not matter. If the first operand was false instead of the second, we would have the same result.

|| This Operator is known as the logical OR operator. If the first operand **OR** the second operand is true, the statement will return true. This means that if the first operand returns false and the second is true, the statement will still return true, and vice versa. The statement also will return true if both the operands are true. Essentially, when using an OR operator, at least one operand must return true.

Example:

```
        var a = 20;
  var b = 30;

if (a == 20 || a == b) {
        alert("a is equal to 20 or b");
}
else {
        alert("a is not larger than b");
}
```

In this example. We check to see if “a” is equal to 20, OR if “a” is equal to “b”. In this case, “a” is equal to 20 and is not equal to “b”. Since only one of these operands need to be true, the statement as a whole will return as true. Therefore the program will print “a is equal to 20 or b” in the console.

Example 2:

```
  var a = 20;
  var b = 20;

if (a == 20 || a == b) {
        alert("a is equal to 20 and b ");
}
else {
        alert("a is not equal to 20 or b ");
}
```

In this example, I have changed the value of “b” from the previous example from 30 to 20. This means that “a” is equal to 20 and “a” is equal to “b”. Both operands return true, therefore the statement returns true. This is due to the fact that an OR operator requires 1 or more of the operators to be true. This program will output “a is equal to 20 and b” in to the console.

! This operator is known as the logical NOT operator. It is used to reverse the logical state of its operand. This operand can be used with any other operand to reverse its output. If an operand was returning true, after applying the logical NOT operator, the operand will return false.

Example:

```
  var a = 21;
  var b = 20;

if ( !(a == 20) && a > b) {
        alert("a is not equal to 20, and a is greater than b");

}
else {
        alert("a is equal to 20, and a is not greater than b ");
}
```

In this example, we check to see if the value of “a” does not equal to 20, and we check if “a” is greater than “b”. Since “a” is both not equal to 20, and greater than “b”, we can print out to the console “a is not equal to 20, and a is greater than b”.

Example 2:

```
  var a = 21;
  var b = 20;

if ( !(a == 20 && a > b) ) {
        alert("a is not equal to 20, and a is greater than b ");

}
else {
        alert("a is equal to 20, and a is not greater than b ");
}
```

In this example, I changed the position of the bracket from the NOT operator to extend to both operands. Now for the statement to be true, “a” has to NOT equal to 20, AND “a” must NOT be greater than “b”. In this situation the statement will return false, because “a” is greater than “b”. Due to the operand it needs to be less than “b” for the statement to return true.

Therefore this program will return “a is equal to 20, and a is not greater than b” to the console.

Combining Operators

In Javascript, it is possible to use as many operators as you require per statement.

```
          Example:
  var a = 21;
  var b = 20;

 if ( (a == 20 && a > b) ||  (a != 20 && a < b)) {
          alert("correct");
 }
 else {
          alert("wrong");
 }
```

With this example, we can see how we can achieve much more complicated statements. Here, we apply an OR operator with two operands, however, each operand also incorporates an AND operator. Therefore, for this statement to return true, “a” must be equal to 20 and be greater than “b” OR “a” must not be equal to 20 AND “a” must be less than “b”. Note that we use brackets to make the code more easy to understand, and to prevent the code from misreading our logic.

```
          Example 2:
          var a = 21;
  var b = 20;

 if ( a == 20 && (a > b ||  a != 20) && a < b) {
          alert("correct");
 }
 else {
          alert("wrong");
 }
```

In the above example, we have significantly changed the logic of the program, by only switching the position of a bracket. Notice how now the brackets surround the two inner operands, instead of two pairs of brackets surrounding each pair of outer operands. For this statement to return true now, 3 different conditions must return true. Variable “a” must be equal to 20, AND, either “a” must be greater than “b” OR it must not be equal to 20; AND “a” must be less than “b”. With this new logic, the first operand must be true, on top of EITHER the second or third having to return true. Also on top of that, “a” must be less than “b”. This statement will return false. It will fail the first condition, because “a” does not equal 20. It will pass the second condition, because it is greater than “b”, and it will fail the last condition because “a” is not less than “b”. Since this would have required three correct conditions, and only had one, the statement returns false and prints “wrong”.

Assignment

We are now going to go through another assignment to make sure we understand everything about operators. We have 3 friends who want to know how much money they have compared to the others. We will make a program that will output who is richer than who. We will have three integers named bob, john, and tom. We will give each one of them a different value, and the program must output who is richer. It must also tell us if someone has the same amount of money as someone else.

For example:
If bob = 20, tom = 10, and john = 5, the output must be:
“bob is richer than tom, who is richer than john”

But if bob = 20, tom = 20, and john = 5, the output must be:
“bob is just as rich as tom, both are richer than john”

The program needs to work for each possible outcome. I highly encourage you to try this program by yourself and struggle through it as much as you can before you look at the answer. A big part of programming is problem solving and understanding code, so I recommend you try to figure it out by yourself first.

Answer and explanation:

Note that the program will be explained through comments in the code.

```
// feel free to change these values around
var tom = 10; //amount of money tom has
var bob = 20; // amount of money bob has
var john = 10; // amount of money john has

/* We first check if everyone has the same amount of money
*  note how I never check if tom is equal to john, because if
*  tom is equal to bob and bob is equal to john, then obviously
*  tom is equal to john
*/
if (tom == bob && bob == john) {

        alert("Everyone has the same amount of money");

}

/*
*  If the first statement returns false
      *  I next check the unique case where tom and bob have the
same amount
*  but john does not
*/
else if (tom == bob && bob != john) {

        /*
        * I now check if bob is greater or less than john
        * and then output the correct answer
        */

        if (bob > john) {
```

```
                     alert("tom is just as rich as bob, both are richer
than john");
            }
            else if (bob < john) {
                     alert("tom is just as rich as bob, both are more
poor than john");
            }
        }
        /*
          * I repeat the same process as the previous statement but with
different people
        */
        else if (tom == john && john != bob) {

            if (john > bob) {
                    alert("tom is just as rich as john, both are richer
than bob");
            }
            else if (john < bob) {
                     alert("tom is just as rich as john, both are more
poor than bob");
            }

        }
        /*
        * The last possible combination of names,
        * same check as previous statement
        */
        else if (john == bob && bob != tom) {

            if (bob > tom) {
                    alert("bob is just as rich as john, both are richer
than tom");
            }
            else if (bob < tom) {
                     alert("bob is just as rich as john, both are more
poor than tom");
```

```
                }
            }

            /*
            * Now I check the last possible combinations
            * where each person has different amounts of money
            * the next 6 statements cover each possible outcome
            */
            else if (tom > bob && bob > john) {

                alert("tom is richest, followed by bob, followed by john");
            }
            else if (tom > john && john > bob) {

                alert("tom is richest, followed by john, followed by bob");
            }
            else if (bob > tom && tom > john) {

                alert("bob is richest, followed by tom, followed by john");
            }
            else if (bob > john && john > tom) {

                alert("bob is richest, followed by john, followed by tom");
            }
            else if (john > bob && bob > tom) {

                alert("john is richest, followed by bob, followed by tom");
            }
            else if (john > tom && tom > bob) {

                alert("john is richest, followed by tom, followed by bob");
            }
        }
```

When writing code it is important to comment all of your logic so that someone who has not written it can easily understand and edit it. I recommend that when you write code you comment it as much as you feel is required. I also encourage you to try to find a better way to solve this problem. Maybe there is a way you can write this in a quarter of the length; with programming, there is never only one answer.

Chapter 7

Loops and Arrays

This section of the guide is important because it covers one of the most important concepts in programming. Loops are what it says they are - they are simply functions that allow you to loop a certain block of code for a certain amount of iterations depending on what you ask of it. Although this may not seem useful right now, when you start to apply your knowledge to real programming applications, you will find use of them. Before we cover arrays, let's go over how loops work.

Loops

Loops in Javascript are relatively the same as many other programming languages. The syntax for it is also pretty straight forward. In every loop, there is a condition that is set forth for the loop to check whether or not it is true and until the case is the opposite, is when the loop itself would break out itself. For example, if you were to create a loop and say that I want it to run a certain block of code 10 times before stopping, there are a few ways to do this because there are a different type of loops. A general idea of a loop is the following:

```
loop (condition) {
        // block of code
}
```

In order to create a loop in Javascript - we must understand the different type of loops and when they are useful. Let's go over these types of loops and then I'll explain how each of them work as we go through and by the end, you should have gained a good understanding of how loops work.

For Loops

```
for (x = 0; x < 10; x++) {
```

```
                alert(“Hi “ + x);
        }
```

There are quite a few things that are going on in the above example so let’s go over it line by line. The first line says:

```
        for (x = 0; x < 10; x++)
```

In the above statement, it is a conventional standard for the for loop. We must first declare the variable, so let’s start by declaring the x variable and make it an int starting at 0. Next, let’s add a semicolon to separate that specific component of the for loop by adding a condition. That is why I did x < 10 which simply says that for however long that x is less than 10, run the block of code that is under my curly braces (“{“ and “}”). This statement is now separated by another semi-colon to let the final element of the for loop to be entered. This is the part where you tell the specific variable that you declared, to do a certain arithmetic operation. In this case, we are telling x to increment each time the loop passes.

Let’s go over this in a pseudocode approach (english-readable):

```
          for ( declare variable; create condition for variable; affect variable)
{
                // Run this block of code
        }
```

Once it runs the block of code, it will go back to the first statement and check whether or not the condition is still met. It will skip declaring the variable since it will notice that it has already been declared. It will then skip to affecting the variable before checking to see if the condition is met.

Here’s a blueprint to follow for the for loop or any loop for that matter:

Declares Variable → Checks **Condition** → **Runs** block of code → **Affects** Variable → Checks **Condition** → **Runs** block of code → **Affects** Variable → Checks **Condition** → **Runs** block of code → Repeat starting from “Affect Variable”

Now if we look back at the code we created with a little modification on alert:

```
for (x = 0; x < 10; x++) {
        alert("Current Number: " + x);
}
```

We can see that this loop is simply incrementing x from 0 to 9, so technically 10 iterations since 0 is inclusive. Therefore, the loop alerts the following as the output:

```
Current Number: 0
Current Number: 1
Current Number: 2
Current Number: 3
Current Number: 4
Current Number: 5
Current Number: 6
Current Number: 7
Current Number: 8
Current Number: 9
```

Remember how I said that a for loop's last statement is the arithmetic operation for the variable that has been declared - well the variable can be affected in any way, you can increment by 2's if you wanted to as well.

Now that we know how a for loop works, it will be easier to understand how a while loop and a do-while loop work.

While Loop

The only difference with a while loop compared to a for loop is that it only carries the condition of the loop, and not the variable. Therefore, you must actually declare the variable outside of the scope of the loop before affecting it in the loop to carry out a certain condition.

For example:

```
   var x = 0;
while (x < 10) {
        alert(“Current number: “ + x);
        x++;
}
```

In the example above, as seen, a while loop simply checks for a condition and when it has been met, runs the block of code, and then affects the variable x so it goes back to checking the condition when the block of code is done executing, and simply continues to do it until x is not less than 10. The output for this is the same as the for loop output:

```
Current Number: 0
Current Number: 1
Current Number: 2
Current Number: 3
Current Number: 4
Current Number: 5
Current Number: 6
Current Number: 7
Current Number: 8
Current Number: 9
```

There are things that can go wrong with this while loop like the accidental creation of an infinite loop. For example:

```
  var x = 0;
while (x < 10) {
        alert(“Current number: “ + x);
}
```

In the above example, it is doing exactly everything the previous loop was doing except the output (which is to alert a certain piece of text) would be infinitely displaying because the variable x is never being affected.

The final type of loop we will be covering is the do-while loop. This loop does everything the while loop does, except it does something at least once regardless of the condition before resorting to depending on the condition.

For example:

```
var x = 0;
do {
        alert("Current number: " + x);
    x++;
} while (x < 10)
```

The above example simply runs the iteration of the block of code at least once regardless of the condition that is given at the end before actually testing the condition. This can be useful if you ever wanted a loop to do something, except at least do it once before checking the condition.

Continue keyword

The continue keyword is used when you want to skip an iteration of a loop. For example:

```
 var x = 0;
 while (x < 10) {
        if (x == 5) {
                continue;
    }
        alert(x);
}
```

In the above example, we are checking if x is equal to 5, if it is then what we do is skip the iteration of the loop, meaning the output of this loop then becomes:

```
Current Number: 0
Current Number: 1
Current Number: 2
Current Number: 3
Current Number: 4
Current Number: 6
Current Number: 7
Current Number: 8
Current Number: 9
```

As seen in the output above, the loop skips 5 because it was told to skip that iteration of the loop before it even had the chance to alert the number out.

Break keyword

The continue keyword is used when you want to break out of the loop. For example:

```
var x = 0;
while (x < 10) {
        if (x == 5) {
                break;
    }
        alert(x);
}
```

In the above example, we are checking if x is equal to 5, if it is then what we do is break out of the loop, meaning the output of this loop then becomes:

```
Current Number: 0
Current Number: 1
Current Number: 2
Current Number: 3
Current Number: 4
```

As seen in the output above, the loop breaks out once it reaches the fifth iteration meaning, to essentially stop the loop from continuing on any further.

If we combine the conceptual understanding of loops and arrays, we can come up with complex code - that is why we will be learning arrays, before we look at a real-world application where this may be useful to use so you can gain a better understanding of why loops and arrays are helpful. You will be using loops and arrays a lot when programming even if it doesn't seem like it right now.

Arrays

Arrays in programming are helpful because it allows you to create lists of strings/numbers/objects or whatever data type you can think of. Again, this might not be apparent at first but will be when you start to program and realize that you may need to use arrays in certain instances. So an array is basically a single variable that holds multiple values that can be accessed individually. Let's go over the syntax of how an array declaration would work.

In order to declare a variable, you must do the following:

```
var names = [“John”, “Bob”, “Tom”];
```

In the example above, what we are doing is declaring a variable with the name of names, and then we are assigning it a list of values, which are of string type. So in order to assign values to the array, we added square brackets which simply set the boundaries for the array. Then, we noted that the array type is string by surrounding each element by quotation marks and then adding a comma in between each element to indicate that it is a different element in the array.

In order to access specific element values from the array, we use the following syntax:

```
alert(names[0]);
```

In the above example, what we are doing is alerting the value of the array's index of 0. In arrays, we refer to the values of the array through the index value - and the index value starts at 0. Therefore, “John” is index of 0, “Bob” is index of 1 and “Tom” is the index of 2. The length of the array itself is 3. In order to get the length of the array we use the following property:

```
alert(names.length);
```

Since arrays in general are objects (you will learn what this means later) they have variables and functions, and length is a property of arrays, and therefore can be accessed by the names variable since it is a variable of array type. The output of this piece of code would be 3 since it would count the first element of the array as 1 and then 2 and so forth.

Now let’s say that we want to declare an array but not assign it any values until later in the code. Then you would do the following:

```
var names = []
names[0] = “John”;
names[1] = “ Bob”;
names[2] = “Tom”;
```

In the above example, we are declaring an empty array that we preparing to add elements to, and we do so by referring to an index that doesn’t yet exist and give it a certain value. Now we can refer to the names array’s index and retrieve the appropriate value that was assigned to it.

```
alert(names[0];
```

The above example now returns “John” as it was assigned the string John. One thing to note is that you’re able to overwrite index values by simply using the assignment operator for the certain index that you would like to overwrite and then assign it something else.

The other way of declaring a Javascript array is by doing the following:

```
var anArray = new Array(30);
```

The above code simply does the same thing as the former way of declaring a variable, and is actually preferred over this way, but this way is being informed to you incase you ever wanted to use it for whatever reason. In this example, there are 30 undefined elements in the anArray variable and can be assigned like they were before.

```
var anArray = new Array(10, 20);
```

In the above example, the anArray variable is declaring two elements in its array of 10 and 20 - the best way to actually create a variable and the conventional and standard way is the way it was taught before we went over new Array() but was taught regardless just so you know that there is a way to do it a second way.

Now this is great and all, having arrays, and being able to hold big databases of names, or social security numbers, etc - but imagine trying to print all these values out manually by pointing out the index of the array and retrieving the value. Well, let's think about it, if I want to print out multiple values from an array from multiple different index values, then I must alert all the different indexes of the array, therefore I can conclude that I could use a loop to affect the variable of the index to alert and retrieve the values of the array variable.

For example:

```
var names = [“John”, “Bob”, “Tom”]
for (i = 0; i < names.length; i++) {
        alert(names[i]);
}
```

In the above example, a variable of name “names” is being declared with three values of “John”, “Bob” and “Tom” - once the array variable has been set. We are looping the i variable until it reaches right before the length of the array (number of elements held in the array) in order to print them out. Then, we alert the array's index values out by placing the i in the brackets next to the array's name in order to refer to the specific index, which then prints out the names accordingly. So the output of this code would be:

```
John
```

```
Bob
Tom
```

Using our understanding from arrays and loops, we have managed to combine their use to output all the elements from the array. The idea of this concept of combining loops and arrays to work together is just one way you can use loops or arrays to do what you want. The whole idea of programming comes from thinking outside of the box by deciding how you are able to manipulate numbers and strings and functionality of the language in order to compute the targeted algorithm that you're aiming to accomplish (algorithm: set of steps to complete to accomplish a certain goal).

Multidimensional Arrays

Multi-dimensional arrays are an important part of programming. They allow us to define a large series of points in a single variable. This can be used to define images by their pixels, height map data, and whatever you can think of. Creating multi-dimensional array is not too easy, but it's definitely possible, and extremely useful. To create a multidimensional array, simply create an array with a given size, and then set each of its elements to an array of a given size, usually with a loop, and so on for each dimension.

```
var dim1 = 10;
var dim2 = 20;
var dim3 = 30;

var arr = new Array(dim1);

for(var x = 0; x < dim1; x++)
{
		arr[x] = new Array(dim2);
		for(var y = 0; y < dim2; y++)
		{
				arr[x][y] = new Array(dim3);
				for(var z = 0; z < dim3; z++)
				{
					arr[x][y][z] = Math.floor(Math.random()*10);
```

```
                }
            }
        }
```

Chapter 8

Useful Functions in JavaScript

Some functions in Javascript should be committed to memory. Rather than always searching up these common functions, just memorizing them makes it easy to code what you need.

Let’s start with a basic one. The getElementById function. This lets us get HTML elements from the document, access and modify them. To use this function, you must access the document object, followed by a dot, followed by “getElementById”. This function returns a DOM object, which we will discuss in more detail later. This function depends on there being an HTML element with the id attribute set to the value of the first parameter.

```
var x = document.getElementById(‘text1’);
x.innerHTML = “Hello World!”;
```

This call looks for an element with the id of “text1”. We then set this element’s innerHTML to “Hello World!”. This will make it write that to the screen.

The createElement function is also rather useful. It creates elements from a string that defines what type of element to create. It then returns this element for use in your code. This function is also part of the document object.

```
var x = document.createElement(“button”);
```

Another useful function is the appendChild function. It can be called on any DOM object. This function will set the value of the first parameter, which should be a DOM object, to be placed as a child of the DOM object that the function is being called on. This function must be used in order to actually put elements onto the screen. The createElement function creates the element, but does not attach it to the document. This function can. We can easily access the body of the document from the document object.

```
var x = document.body.appendChild(document.createElement(“button”));
```

The Math object is full of useful functions. This includes things like the pow function, which calculates powers; the sqrt function, which calculates the square root of a number; the abs function, which gets the absolute value of a number; the ceil function, which rounds a decimal number up to a whole number; the floor function, which rounds a decimal number down to a whole number; and the round function, which rounds a decimal number the way math class has taught you: .5 and above rounds up, lower rounds down.

```
var x = Math.pow(5, 4); // x = 625
var y = Math.sqrt(x); // y = 25
var z = Math.abs(-7); // z = 7
var i = Math.ceil(3.2); // i = 4
var j = Math.floor(3.2); // j = 3
var k = Math.round(3.2); // k = 3
```

Another useful function is the setInterval function. This function allows you to make a function be called every x milliseconds. This function takes in 2 parameters: a function to be called, and a number of milliseconds to wait between calls. It also returns a number which is the interval id, which can be used to stop the interval.

```
var x = setInterval(function(){console.log(“Groundhog Day!”);}, 1000);
```

This example will print Groundhog Day! every second. Another useful function is the clearInterval function. It stops an interval by its id. It takes one parameter, which is the interval’s id.

```
clearInterval(x);
console.log(“Groundhog day is over!”);
```

Chapter 9

DOM Elements

DOM elements are what get shown on your HTML document. DOM stands for Document Object Model. DOM elements contain a type, which are defined by the tags that create it; attributes, which make the element more unique; and innerHTML, which defines what text or elements are contained in it. Let’s see an example element. This would go inside your HTML file.

```
<div id=’x’ style=’border: black 1px solid’>
        This is part of the innerHTML.
        <button>Click Me!</button>
</div>
```

We can see that this element has the type of “div” as defined by tags. This “div” element has two attributes set as well. First, the id attribute, which let’s us find this element using code. Second, it’s style attribute, which changes how the element is displayed on the screen. We can also see that this element has it’s innerHTML filled in, with some text and even another element, which is a button. Using these elements, we can do a variety of things like get user input and output messages for the user to read.

There are several different types of elements. There are “div” elements, which divide the page into sections; “input” elements, which allow users to give input to your code; “a” elements, which are links to websites; “li” elements, which displays a list of items; “canvas” elements, which allow us as developers to draw graphics to the screen through code; and many more.

Chapter 10

AJAX

AJAX stands for Asynchronous Javascript And XML. This is the method of asynchronously retrieving data from a server, usually some sort of XML file, like an HTML file. AJAX can be used to dynamically load parts of your web page, specifically things that need to be calculated server-side and sent to the client, for example, on YouTube, you could open your "My Subscriptions" page and the server would calculate which videos you need to see based on your subscriptions and load the data for these videos, including name, link, image, description and so on. AJAX is rather simple to accomplish. The first step is to figure out which type of request you can make. This is because older browsers use the "ActiveXObject", while modern browsers use the "XMLHttpRequest". So here is the code.

```
var requestType;
if (window.XMLHttpRequest)
{
        requestType = XMLHttpRequest;
}
else
{
        requestType = ActiveXObject;
}
```

This now lets you use the proper request throughout your code without needing to check each time which request to use. We are essentially setting requestType to have the value of either of these functions. Now let's configure our request. First, we make our object.

```
var request = new requestType("Microsoft.XMLHTTP");
```

Here we create our request using the requestType we defined before. We also use the parameter of "Microsoft.XMLHTTP". This is kind of tricky. The XMLHttpRequest object has no parameters by default. This means that if requestType is an XMLHttpRequest, the parameter will be ignored. The ActiveXObject, on the other hand, does have a single parameter, which in this case will always be “Microsoft.XMLHTTP”. This means that if requestType is an ActiveXObject, it will use this parameter. So that is how we create our request. The next step is to define what should be done when our request is done. To do this, we set the request’s onreadystatechange variable and set it to a function.

```
request.onreadystatechange = function()
{
        if(request.readyState == 4 && request.status == 200)
        {
                document.write(request.responseText);
        }
}
```

In this function, we first check two things: readyState and status. ReadyState checks which state the request is in, where 4 is finished. Status checks what code the server returned, where 200 is “good”. Status can also be other values, like 404, which means “file not found”. If the request is finished and has the status code of 200, we run whatever code we want, which in this case is simply to write out the text to the document. The final step in AJAX is to actually send the request. Here is the code for that.

```
request.open(“GET”, “local url of target file”, true);
request.send();
```

So first we call the open method, with 3 parameters. First, the method, which can either be GET or POST. I will not go into too much detail, but essentially, GET sends extra data through the url, while POST sends extra data in a more hidden and secure manner. Second, the file. This is local to the website address, so if your address is “ www.123random.com ”, and for your file parameter you put “myfile.php”, AJAX will go to “ www.123random.com/myfile.php ”. Third, asynchronous or not. Generally, you will want to make this asynchronous, because if you don’t the whole website will freeze until the request is finished, which if the file is not able to be sent, could take a while. So unless you know what you’re doing, keep the third parameter “true”. Finally, we call the send method, which does exactly what is says—sends the request. Be aware, that if you are using the POST method, to place a string as a parameter for the send method, but before using POST learn how to properly use it and if you actually need it or not.

Our example code is very simple in that we do nothing special with the response. One thing you can do is, rather than using responseText, you can use responseXML. Using this, you can call “getElementsByTagName” functions and access all sorts of different data very easily.

Chapter 11

NodeJS

NodeJS is a replacement for regular Apache servers. NodeJS offers real-time server management (as opposed to Apache servers, which runs PHP code only when it is requested by a client) and allows server-side programming in Javascript. With NodeJS, you can completely customize the manner in which your server responds to server requests. This can be useful for example if you want to respond with a random file / set of data every time a client sends a certain request to the server. On top of this, you can also have your server do a number of real-time tasks, like responding to sockets and interacting between sockets even. This can be used to do a number of things, but we will be using it in terms of games. The first step is to download NodeJS and install it. Now let's setup our project. You may want to make a new folder to keep your area clean. In this folder, create a new file called "package.json". You may need to open your text editor of choice and "save as" package.json. In this file you will write the following:

```
{
  "name": "write the name of your project here without spaces, but surrounded by double quotes",
 "version": "version number of your project",
 "description": "A description of your project",
 "dependencies": {
  "mime": "^1.2.11",
  "socket.io": "^1.1.0"
 }
}
```

The dependencies tell us which NodeJS modules we will be using. There are 2: Mime, and SocketIO. Mime is for returning the proper type of files being sent. SocketIO allows us to connect to users and be able to dynamically send data to the user, which could modify the user's session and show them new data. This is kind of similar to AJAX, except with the difference that sockets work in real-time. With AJAX, you have to wait to send a request, for the server to respond and then finally you can get your data. With sockets, you do not even need to send a request to get data. You may need to connect to the server, but after that, the server can send you data, even when you haven't asked for it. This is very important in games, because let's say a user on another computer moves. There is no way your computer could tell that that user moved. The server has to send that data to you and it is much easier to simply have the server send data when it needs to update you, rather than having the client keep sending requests, which can waste more bandwidth.

Once you have the package.json configured, simply open your command-line or terminal and navigate to the directory of your project (where package.json resides). Then type "npm install". This will install Mime and SocketIO to your project. The next step is to create a new file, call it whatever you like, with an extension of .js. This will be your main Node server code, so you may want to name it something along those lines. So let's see some server code. Here is a simple example.

```
var http = require('http')
var server = http.createServer(handler);

server.listen(80, "127.0.0.1");

function handler(req, res)
{
	res.writeHead(200, {'Content-Type': 'text/plain'});
	res.write('Hello World\n');
	res.end();
}
```

This code is simple. First, we load the "http" module, using the require method. The require method can be used to load modules, including custom ones. Then we create a server, and supply it with a handler function. This handler function has two parameters: req—requests; and res—response. Request tells us all sorts of data about the user's request, which can include the url, POST data or whatever else. Response allows us as the programmers to respond to the user's request with whatever data we want, whether that be related to the user's request or not. In this example, we ignore whatever request they made and simply respond with a status code of 200 (meaning “good”) and a content type of “text/plain” (meaning that we are sending plain text to the user). This is the writeHead function—we are writing the header of our response. Our body on the other hand, we need to use the write function. This simply writes text to our response, in this case we simply respond with “Hello World”. Finally we call the end function, which ends our response, and sends it off to the user. So that’s our handler function. Finally, once we create our server, we tell it to listen on port 80 at the 127.0.0.1 ip. This is our very simple server.

Let’s make our handler more complex now. First, let’s try and duplicate an Apache server. Here’s a Node server that does just that.

```
var http = require('http');
var fs = require('fs');
var mime = require('mime');
var server = http.createServer(handler);

server.listen(80, "127.0.0.1");

function handler(req, res)
{
        fs.lstat("public" + req.url, function(err, stats)
        {
                if(err)
                {
                        res.writeHead(400, {'Content-Type': 'text/plain'});
                        res.write("An error occured attempting to retrieve
the requested file.");
                        res.end();
                        return;
                }

                if(stats.isDirectory())
                {
                                fs.readFile("public" + req.url + "/index.html",
"utf8", function(err, data)
```

```
                                {
                                        if(err)
                                        {
                                                res.writeHead(400, {'Content-
Type' : 'text/html'});
                                                res.write("An error occured
attempting to retrieve the requested file.");
                                                res.end();
                                                return;
                                        }
                                        res.writeHead(200, {'Content-Type' :
'text/html'});
                                        res.write(data);
                                        res.end();
                                });
                        }
                        else
                        {
                                fs.readFile("public" + req.url, "utf8", function(err,
data)
                                {
                                        if(err)
                                        {
                                                res.writeHead(400, {'Content-
Type' : 'text/html'});
                                                res.write("An error occured
attempting to retrieve the requested file.");
                                                res.end();
                                                return;
                                        }
                                        res.writeHead(200, {'Content-Type' :
mime.lookup(req.url)});
                                        res.write(data);
                                        res.end();
                                });
                        }
                });
```

```
}
```

So what in the world is going on here? First, we have loaded two new modules: fs (file system) and mime (detects the type of a file). In our handler, we do a variety of things. First we get the stats on our url, but with “public” appended to it. This means that no matter what, the handler will only look through files in a “public” folder that must be in your project (right next to your server’s .js file). This is to prevent people from seeing anything other than what is public. It would not be very good if people could see your server’s code, because then they may be able to find exploits easier and use them. We also give this lstat method a callback. This is what makes the method asynchronous. It is very important this stuff is asynchronous, because otherwise, your entire server would freeze when it is checking for file data. This is bad because this would mean your server would stop responding to users until the file stuff is finished doing it’s thing.

In the callback, the first thing we do is check if an error exists, and if it does, respond to the user with a very generic error message (if you wanted to make this better, you could change this error message based on what error you are getting, but this is supposed to be simple) and do not proceed to the rest of the method. If there is no error, we check if the file we are looking at is a directory. If it is, then we try and get the default file, the index.html file. We do this by simply appending ‘/index.html’ and trying to read this file. The readFile also has a callback, meaning it is also asynchronous. In this callback, we again check if we have an error, and return a generic error message if so. Otherwise, we use the data and send it to the client.

Notice that we are assuming that the index file can only be an html file. This may not always be the case, but most of the time, you could have an html file that simply loads other dynamic data using AJAX. Now, if the original request was not a directory, meaning it is a file, we simply try and load that file, check if there is an error, and respond if we can. Notice that for the “Content-Type”, we use “mime.lookup(req.url)”. This method finds the appropriate content type for the file we are requesting. This is simply to avoid confusing the browser, or whatever else is requesting from the server. Now, this is nowhere close to the most complex handler you could make. You could have it load and run php files potentially, and you could make the error handling more robust, but as a simple Apache server, this will do just fine.

The next thing we will learn is SocketIO. As stated above, sockets allow for real-time data to be transmitted between the clients and the server. SocketIO is rather easy to use, and we will not go into some of the complexities, but you could do quite fancy stuff with what you will learn. So let’s learn it! The first step is to load the module in your server.

```
var socketIOModule = require(‘socket.io’);
```

The next step is to create a socketIO object. Simply use socketIOModule as a function and use the server from our previous example as the first parameter.

```
var socketIO = socketIOModule(server);
```

Finally, we will create a handler for the SocketIO ‘connection’ event. This means that whenever a user connects through SocketIO, this handler will fire.

```
socketIO.on(‘connection’, socketConnect)

function socketConnect(socket)
{

}
```

The socketIO object means that globally, all of SocketIO will call the socketConnect callback, whenever a connection event is raised. Our callback, has the parameter of socket, which can be set to respond to events in the same way the socketIO object is set to respond to events—using the “on” method. The first parameter in this method is the event name. You can make custom event names, you just type in the event name in this first parameter, and whenever the server gets this event (from this socket), it will call the second parameter—the callback. The callback’s parameters are dependent on the type of event, but all custom events will only have a “data” parameter, which is any object. Socket.emit is the other side of the exchange. This actually sends an event to the client or server on the other side of the socket. The first parameter is the event to send in the form of a string. The second parameter is the object to send.

```
socketIO.on('connection', socketConnect)

function socketConnect(socket)
{
        socket.emit('greetClient', {});
        socket.on('greetServer', function(data){
                if(data.playerName == "Spykill")
                {
                        socket.emit('changeRank', {rank: 'admin'});
              }
       });
}
```

In this system, when a user connects, the server automatically sends a “greetClient” event. On the client-side, we would handle this event, and probably send a “greetServer” event, where we might send some data like the playerName. We check the playerName, and if it is someone important, we send a “changeRank” event to the client, with some data telling which rank the player should be. Now let’s look at some client-side code.

```
<!DOCTYPE html>
<html>
<head>
        <title></title>
        <script src="https://cdn.socket.io/socket.io-1.2.0.js"></script>
        <script type="text/javascript">
        var socket = io();
        socket.on('greetClient', function(data)
        {
                socket.emit('greetServer', {playerName: "Spykill"});
        });
        socket.on('changeRank', function(data)
	{
		if(data.rank == ‘admin’)
		{
			document.write(“You are an admin!”);
		}
	});
	 </script>
</head>
<body>

</body>
</html>
```

The first thing we need to do that is unique, is to load the socket.io.js file from the socket.io website. You could also make it load socket.io from your server, but it doesn’t matter much. Next, we write a script. We connect to our SocketIO server by calling io(). We then tell this connection to run a callback method, whenever it gets the ‘greetClient’ event. In this callback, we simply emit—meaning send—a return event called ‘greetServer’ with the playerName set to “Spykill”, which if you recall will be responded by the server to emit a ‘changeRank’ event. We also tell our socket to call another callback when we receive the ‘changeRank’ event, that will simply write out whether we are an admin or not. This is a very simple client server relationship. In reality, you would want to do things like check if the user has valid login credentials, and you would send users more data.

NodeJS allows a ton of customization. It has a large collection of modules which you can find and implement into your project. This makes it easy to develop custom applications and servers for your project, whatever it may be.

Chapter 12

Canvas

The canvas is a new element in HTML5. It allows programmers to draw to the screen using JavaScript, and can handle both 2d and 3d applications. We will avoid the 3d side, as this goes into WebGL territory, and that is very difficult to accomplish and understand, and there is a lot of code involved. So, let’s talk about the canvas in 2d. The first step in using a canvas is creating the element. In the body of your html file, wherever you wish the canvas to appear, simply use the <canvas> tag.

```
<body>
        <canvas></canvas>
</body>
```

Next we need to give our canvas some dimensions. In the first canvas tag, add the ‘width’ and ‘height’ attribute, and set them to the desired value. You should also set the ‘id’ attribute to an identifier to use to find this element.

```
<canvas id=’ourCanvas’ width=’800’ height=’600’></canvas>
```

You may also want to add a border of some sort to distinguish the background from the canvas. This is getting into CSS territory, so I will keep this short. Just set the style attribute to “border:black 1px solid”. This will make a black border with a width of 1 pixel around your canvas.

```
<canvas id=’ourCanvas’ width=’800’ height=’600’ style=’border:black 1px
solid’></canvas>
```

Now that we have our canvas set up, we have to use it. We need to make a function that will run after the canvas has loaded. The first thing it must do is find the canvas, using its id. Then it must get a 2d context for the canvas, which will be used to draw to the canvas.

```
function doCanvasStuff()
```

```
{
        var canvas = document.getElementById('ourCanvas');
        var context = canvas.getContext('2d');
}
```

Before we continue, there is one final thing we must do to our canvas element before continuing. We must set its onload attribute to execute our doCanvasStuff function, so that once the canvas has loaded, it will run our canvas code.

```
<canvas id='ourCanvas' width='800' height='600' style='border:black 1px
solid' onload='doCanvasStuff()'></canvas>
```

Next, we need to use our context to draw some stuff. First let's talk about the styles. Styles describe how the computer will draw to the canvas. This can mean that the style is a solid colour, or maybe it's a gradient. There are a variety of things you can do with styles, but we will focus on solid colours for most of this. There are two styles that you can set: the fill style and the stroke style. The fill style does exactly what it sounds like—it fills shapes. The stroke style basically follows the line of the shape. You can use the stroke to outline your shape. To set a style to a solid colour, simply set the context's strokeStyle or fillStyle to an html colour value. Whether this colour value is hexadecimal or it is the value of an rgb function is not important. Here is an example of both.

```
context.strokeStyle = '#888888';
context.fillStyle = 'rgb(200, 45, 130)';
```

Here we set the stroke style to a hexadecimal value (notice that we have a pound sign, #, in front of our hex value), which evaluates to grey, and we set our fill style to the value of the rgb function (notice that this function is in quotes, as this is not a JavaScript function, but either an HTML function or a CSS function) with the red value set to 200, the green to 45, and the blue to 130, which ends up looking like a pink colour. Now whenever we call a fill function, or a stroke function, these colours will be used. So how do we actually draw things? Let's start with a rectangle. There are multiple methods to make a rectangle. The first is to use the fillRect function (or the strokeRect function, as this will allow you to just outline a rectangle). It takes four parameters, which you could probably guess: x position, y position, width, and height. Be aware that the x and y positions start at an origin at the top-left. You can have negative x and y values, just part of the rectangle will be cut off. All of these values are measured in pixels.

```
context.fillRect(100, 100, 200, 200);
```

This rectangle will go from a position of (100, 100) to a position of (300, 300). Another way to make a rectangle (and is actually more flexible, so you could modify it to make other shapes) is to use paths. Paths simply define a shape, and have a variety of uses. To create a path, simply call the beginPath function. It takes no parameters. Next, we need to start our shape. We do this by calling the moveTo function with the parameters of an x and y position.

```
context.beginPath();
context.moveTo(100, 100);
```

Next we define the lines of our rectangle using the lineTo function, with the parameters of an x and y position. This will make a line from our current position, which we last set using the moveTo function, to the position defined by the parameters. Be aware that after using the lineTo function, the current position is set to the position defined by the parameters.

```
context.beginPath();
context.moveTo(100, 100);
context.lineTo(300, 100);
context.lineTo(300, 300);
context.lineTo(100, 300);
context.lineTo(100, 100);
```

The last lineTo call depends on whether you wish to stroke your rectangle or not. If you do not add this last lineTo, there will not actually be a line on that side of the rectangle, meaning the stroke will not draw a line there. Fill on the other hand, will fill that area no matter what, because to make a solid shape, it must connect the last point with the first if a line does not exist there.

The final thing we need to do is tell the computer to actually draw the shape in some way. Here we will do both a stroke and a fill. Be careful with the order. You generally want to fill before you stroke, because you want the outline to be over top the filled in shape. To do either of these functions, use the fill function and the stroke function, each empty of parameters.

```
context.beginPath();
context.moveTo(100, 100);
context.lineTo(300, 100);
context.lineTo(300, 300);
context.lineTo(100, 300);
context.lineTo(100, 100);

context.fill();
context.stroke();
```

With this second method, we can do more fancy things than a rectangle. For example, if we change our second lineTo's x parameter to something like 500, our shape will appear to be half a trapezoid. You could do other interesting things as well. For example, if you make a for loop, that runs for a random number of loops, where in each iteration, you make it lineTo a random coordinate in the range of the canvas, you could make a jumble of lines. Not too useful, but when you combine loops and lineTo's you can do very interesting things. Here is the code for that jumble of lines.

```
context.beginPath();
context.moveTo(Math.random() * canvas.width, Math.random() *
canvas.height);
for(var i = 0; i < 300; i++)
{
        context.lineTo(Math.random() * canvas.width, Math.random() *
canvas.height);
}

context.stroke();
```

Chapter 13

JavaScript Objects

In Javascript, objects are rather fluid. They are not defined like other languages. They store values given a variable name, and that is all. These variables can be anything you want, from numbers, to other objects, to functions. There are two ways to make objects. Here is the first.

```
var x = { y : 17, z : “Yo wat up?”, i : function(){ console.log(“Hello
friend!”); }, j : { j : “This j can be inside of the other j because they are
within each other” } };
```

We start with curly brackets to define our object. Inside, we put a variable name, followed by a colon, followed by a value. Notice that in our j object in our x object, we have a j variable. This is because the j variable in the j object is contained within the j object, and ignores the x object. That was kind of confusing. Look at it this way. When we are setting our j object, we are creating a new object. In this object we have a variable called j, with some value. The x object just knows that it has a variable called j, while we know that this j is an object. Now how do we access these values? Simply type the object, followed by a dot, followed by the variable name. Then treat them like whatever value you set them as.

```
console.log(x.y);
console.log(x.z);
x.i();
console.log(x.j.j);
```

We treat x.i as a function, as seen by using the parameter brackets. We also treat x.j as an object, by using the dot to access it’s j variable. Changing these object values is also possible. To do this, simply treat the value as a normal variable. Just write out its location (within the object, for example, x.j.j), followed by an equals sign, followed by a value. Be careful which value you are placing inside a variable, as there are no restrictions when setting variables in objects. You may create x.y to be a string, but then accidentally set it to a number. This may be desirable or not depending on your situation.

```
x.y = 42;
x.j.j = “I made a new string!”;
```

You can also add new variables to an object by simply setting it. You do not need to write the var keyword.

```
x.j.k = “new variable!”;
```

The other way of making an object is to use a function. To do this simply take any function, and call it with the “new” written in front. This function is called the constructor.

```
function Person()
{

}

var david = new Person();
```

Inside the constructor, we can use the this keyword. The this keyword refers to the object containing the function being run. In the case of the constructor, this refers to the object it is creating. Using this, we can create values when the object is created. Remember that the constructor is still a function, so it can have parameters, and remember that the object is still an object, so all the features of an object, including being able to create variables at any time, are still valid.

```
function Person(personName)
{
	this.name = personName;
}

var david = new Person(“David”);
david.age = 23;
```

```
console.log(david.name + “, ” + david.age);
```

We can also create functions in this object, and we can use the this keyword in them as well.

```
function Person(personName, personAge)
{
            this.name = personName;
            this.age = personAge;
            this.birthDay = function()
            {
                        this.age++;
          };
          this.talkAbout = function()
          {
                              console.log(this.name + “ is currently ” + this.age + “
year(s) old”);
          };
}

var david = new Person(“David”, 23);
david.talkAbout(); // David is currently 23
david.birthDay();
david.talkAbout(); // David is currently 24
```

Here we change the value of age from within the object itself. Objects are incredibly useful for transferring related data. Be careful when attempting to access variables, though, as if the variable has not been defined, it will have a “value” of “undefined”, which may break your code.
A neat feature of objects is that when they are being passed in as a parameter for a function, they are passed by reference, not by value. This means that if you change the values in an object, the original object will be changed as well.

```
function modify(obj, val)
{
            obj.y = val;
}

var x = { y:9 };
modify(x, 4);
console.log(x.y); // 4
```

Chapter 14

JavaScript Functions

A function is a block of code that can be run at any point in your program with a single line of code, and it can be used as many times as you'd like. This allows you to write a block of code once, and then be able to use it as many times as you'd like throughout your code. To create a function, simply write "function", a variable name, and then brackets, followed by squigly brackets, which hold your code.

```
function printSomething()
{
        console.log("Something I guess? I don't know...");
}
```

This is a very simple function. Between the normal brackets, you can place variable names. These are called parameters. These parameters will allow you to give values to the code, which could give you a different output.

```
function printSomething(times)
{
        for(var i = 0; i < times; i++)
        {
                console.log("Mum!");
        }
}
```

This function uses a variable called times, runs a loop for that number of times, and in each loop prints "Mum!". So if you run this code with a value of 7, "Mum!" will print out 7 times. You can have as many parameters as you want (although the fewer the better). Just separate each variable name with a comma.

```
function printSomething(times, whatToPrint)
{
	for(var i = 0; i < times; i++)
	{
		console.log(whatToPrint);
	}
}
```

This function will print out whatever whatToPrint is set to for however many times is set. So how do we use this code? To make this code run, simply type the function name, followed by brackets, which contain values, separated by commas, which should relate to what the function is using it for. For example, for our function, our first parameter is the times variable, so we should only be putting a number as the value for our first parameter, specifically a whole number (although in this case, whether the number is whole or not is not important). As well, for the second parameter, we should put what we want to print. This is called "calling" a function.

```
printSomething(7, "I like to move it, move it!");
```

We can call our function as many times as we want, whenever we want. Something interesting about functions is that you can use functions as values when setting variables, and then call them from these variables. The difference between calling a function, and using it as a value, is that when calling a function, you use brackets with parameters in them, while when using a function as a value, you do not place brackets.

```
var aFunction = printSomething;
aFunction(3, "Strike!");
console.log("You're out!");
```

An important feature of functions is the fact that you can “return” values. This means that the function will act as the returned value when called. To return a value, simply use the return keyword, followed by a value. Here is an example. To use the returned value, simply imagine the function call as being the returned value.

```
function calculateHalfCircumference(radius)
{
```

```
            return Math.PI*radius;
}

var circumference = 2*calculateHalfCircumference(7);
console.log(circumference);
```

In this example, we create a function, and return half of the circumference (why you would want to do this, I do not know, but for the sake of an example, that is what we will do). Outside the function, we create a variable, and set it to twice the returned value of our function, which we know will be half the circumference of our circle. If we are taking half of the circumference and multiplying it by 2, that means we are getting the total circumference, which we print out. Returning also has a secondary function. It stops execution of a function. This means that say you have a loop in your function that runs for a long time, but you want it to stop after an if-statement runs. Just use the return keyword and it will stop the rest of the loop from running. The return keyword does not need a value to be returned either. You can use the return keyword on its own to end execution of your function where it may be.

```
function kindaUseless(number)
{
            var i = 0;
            while(true)
            {
                        i++;
                        if(number == i)
                        {
                                    console.log(i);
                                            console.log(“That wasn’t the best way to do
that”);
                                    return;
                        }
            }
}
```

This is not a very useful example, but it shows the syntax of it. This loop will run as long as true...is true. This loops will run forever. We also have a variable called i that we simply make count up. In each iteration, we check if the number we are looking for is equal to our i variable. If it is, we print out i and also just print something about this function in general. Finally, we return the function. Since the function stops execution after it’s returned, the infinite loop is escaped, and our program can continue. This is extremely useful, especially in situations where you want to limit the number of iterations. For example, if you are searching for something in an array, and you find it, you may want to just return right there, to avoid wasting more iterations on looking for a duplicate or something.

Functions are incredibly useful, especially when your code becomes exceedingly lengthy. Rather than have similar code all throughout your code, you could use a function to only have the code written out once, and called where it needs to run. Functions also allow us to make code clearer just as comments do, but in the form of the function name. For example, say you have some hard-to-read or confusing code that prints the output of your code. You could make a function called "printOutput". This makes it easier to read code, because instead a block of code, you have a single line that just tells you what it's doing: printing the output.

Chapter 15

Debugging

Sometimes when programming, your code may give wrong outputs. This is called a logic error. Your browser will not tell you anything is wrong, because technically your code follows the rules of Javascript. It's like writing in any language. You may follow the rules of the language, like grammar, structure, etc, but your paragraphs don't necessarily have to make sense. The most common method of debugging is simply using console.log and printing out variables to find where values stop making sense. Then you can find the lines that are causing the error and debug that.

One thing that may or may not be considered debugging is optimization. Once you've written your code, it may be tempting to just leave it and continue on with other tasks, but this may not be what you want to do. If your program gets the desired results eventually, but it takes a long amount of time, then it is not very good. That said, there is no single strategy to make code more optimized. Sometimes your logic could simply be a long method of doing something, and sometimes it could be how you implement that logic. Be wary of creating unused variables, as they can take up processing time and memory. This is because of a system called "garbage collection". It goes through all the variables in your code and removes any ones that are no longer valid. For example, if you make a for loop, after the for loop, that variable is not valid, so it should be deleted to save on memory. This takes up (small) amounts of processing time.

Here is an optimization example. Say you want to find the distance from one point to another. Math class tells us to use Pythagorean theorem. But this is rather slow. First the computer must square both sides (which is quite expensive—performance-wise), then add them, then square root them. This calculates the accurate distance between two points. Another solution though, could be to use "Manhattan distance". This will not get you accurate distances, but it could be good in situations where the exact value is not important, and speed is more important. To do Manhattan distance, simply absolute (which means to make a value positive, so -7 becomes 7 and 8 remains 8) both the x and y components and then add them. This gives you an inaccurate distance, but it is much faster than its more accurate counterpart. This is particularly good when you are guessing the fastest route. Rather than constantly calculating the distance accurately, Manhattan distance is a cheap alternative and will get you a near enough distance.

Chapter 16

Exception Handling

An exception is an error that comes from actually running your code. This is different from a syntax error. A syntax error will stop your code from running because your code doesn't make sense. Exceptions are different. Exceptions will stop your code from running because certain values are breaking your code.

Another example would be when you're working with file handling, and you're trying to read a file in a directory that doesn't exist. The compiler would then again return an exception such as "FileNotFoundException" - there are different exceptions but this type of exception is regularly found in File Handling in many programming languages.

In order to regulate exceptions in your code, you can use the try-catch statements. What these statements do is check if a block of code returns an exception, and if it does, instead of halting the whole operation of the program, it will catch the exception and try to either fix it or do something alternatively depending on what the programmer codes. Here is a use of a try-catch statement:

```
try {
        adddlert("Welcome guest!");
} catch (err) {
        alert("Error!")
}
```

The try statement does not require parameters but the catch exception requires the parameter of an Exception object, and when an exception does occur, the variable "err" of type Exception is given data, and the "err" variable can be manipulated to either print out the error or do various other things. In the above example, adddlert isn't a real function and therefore will return an exception. Now there is a finally statement that is executed regardless of the situation in try and catch.

For example:

```
try {
        adddlert("Welcome guest!");
} catch (err) {
        alert(“Error!”)
} finally {
        alert(“Finished!”)
}
```

Chapter 17

Examples

Here we will be showing off an A* pathfinding algorithm in the form of a Pathfinding object. This object has a method called findPath, which takes in three objects. The first object defines the start of the path, and contains an x and y variable. The second object defines the end of the path, and contains an x and y variable. The third object defines the map that we will be pathfinding around, as a 2d array where a 1 is solid and a 0 is free.

```
var Pathfinding =
{
		findPath: function(start, end, map)
		{
				var time = new Date().getTime();
				var openList = new Array();
				var closedList = new Array();
						openList.push({x: start.x, y: start.y, g: 0, h:
Math.abs(end.x-start.x) + Math.abs(end.y-start.y)});
						closedList.push({x: start.x, y: start.y, g: 0, h:
Math.abs(end.x-start.x) + Math.abs(end.y-start.y)});
				var reachedTarget = { value: false };
				 var pathTile = {x: -1, y: -1, g: 0, h: Math.abs(end.x-start.x)
+ Math.abs(end.y-start.y)};

				var width = map.length;
				var height = map[0].length;

				while(!reachedTarget.value)
				{
						openList.sort(function(a, b){ return (a.g+a.h)-
(b.g+b.h); });
					Pathfinding.addAdjTilesToOpenList(openList[0],
map, openList, closedList, end, reachedTarget, pathTile, width, height);
					closedList.push(openList[0]);
```

```
                              openList.splice(0, 1);
                              if(openList.length == 0)
                              {
                                        closedList.sort(function(a, b){ return a.h -
b.h; });
                                                                                                return
Pathfinding.getPathFromLastTile(closedList[0]);
                              }
                    }

                    var path = Pathfinding.getPathFromLastTile(pathTile);
                    return path;
          }

          , addAdjTilesToOpenList: function(tile, map, openList, closedList,
end, reachedTarget, pathTile, width, height)
          {
                    var i = tile.x;
                    var j = tile.y;
                      if(i-1 >= 0 && (map[i-1][j] != 1 || (i-1 == end.x && j ==
end.y))) { Pathfinding.tryPush(Pathfinding.calcTile(i-1, j, tile, false, end,
reachedTarget, pathTile, openList, closedList), openList); }
                     if(i+1 < width && (map[i+1][j] != 1 || (i+1 == end.x && j
== end.y))) { Pathfinding.tryPush(Pathfinding.calcTile(i+1, j, tile, false, end,
reachedTarget, pathTile, openList, closedList), openList); }
                      if(j-1 >= 0 && (map[i][j-1] != 1 || (i == end.x && j-1 ==
end.y))) { Pathfinding.tryPush(Pathfinding.calcTile(i, j-1, tile, false, end,
reachedTarget, pathTile, openList, closedList), openList); }
                     if(j+1 < height && (map[i][j+1] != 1 || (i == end.x && j+1
== end.y))) { Pathfinding.tryPush(Pathfinding.calcTile(i, j+1, tile, false, end,
reachedTarget, pathTile, openList, closedList), openList); }

                      if(i-1 >= 0 && j-1 >= 0 && (map[i-1][j-1] != 1 || (i-1 ==
end.x && j-1 == end.y))) { Pathfinding.tryPush(Pathfinding.calcTile(i-1, j-1,
tile, true, end, reachedTarget, pathTile, openList, closedList), openList); }
                     if(i+1 < width && j-1 >= 0 && (map[i+1][j-1] != 1 || (i+1
== end.x && j-1 == end.y))) { Pathfinding.tryPush(Pathfinding.calcTile(i+1,
j-1, tile, true, end, reachedTarget, pathTile, openList, closedList), openList);
}
                              if(i-1 >= 0 && j+1 < width && (map[i-1][j+1] != 1 || (i-1
== end.x && j+1 == end.y))) { Pathfinding.tryPush(Pathfinding.calcTile(i-1,
j+1, tile, true, end, reachedTarget, pathTile, openList, closedList), openList);
}
                              if(i+1 < width && j+1 < width && (map[i+1][j+1] != 1 ||
(i+1        ==        end.x        &&        j+1        ==        end.y)))        {
Pathfinding.tryPush(Pathfinding.calcTile(i+1,    j+1,    tile,    true,    end,
reachedTarget, pathTile, openList, closedList), openList); }
              }

              , tryPush: function (obj, openList)
              {
                            if(obj != undefined)
                            {
                                          openList.push(obj);
                            }
              }

                     , calcTile: function (x, y, parent, diagonal, end, reachedTarget,
pathTile, openList, closedList)
              {
                                          if(Pathfinding.findTileInList({x:x, y:y}, openList,
closedList) == -1)
                            {
                                          var i = end.x-x;
                                          var j = end.y-y;
                                          if(i == 0 && j == 0)
                                          {
                                                        reachedTarget.value = true;
                                                         //pathTile = { x: x, y: y, parent: parent, g:
parent.g+(diagonal ? 1.414 : 1), h: Math.abs(i) + Math.abs(j) };
                                                        pathTile.x = x;
                                                        pathTile.y = y;
                                                        pathTile.parent = parent;
```

```
                                        pathTile.g = parent.g + (diagonal ? 1.414
: 1);
                                        pathTile.h = Math.abs(i) + Math.abs(j);
                                }
                                    return { x: x, y: y, parent: parent, g: parent.g+
(diagonal ? 1.414 : 1), h: Math.abs(i) + Math.abs(j) };
                        }
                        else
                        {
                                return undefined;
                        }
                }

                , findTileInList: function (tile, openList, closedList)
                {
                        for(var i = 0; i < openList.length; i++)
                        {
                                    if(openList[i].x == tile.x && openList[i].y ==
tile.y)
                                {
                                        return i;
                                }
                        }
                        for(var i = 0; i < closedList.length; i++)
                        {
                                  if(closedList[i].x == tile.x && closedList[i].y ==
tile.y)
                                {
                                        return i;
                                }
                        }
                        return -1;
                }

                , getPathFromLastTile: function(tile)
                {
                        var path = new Array();
                        path.push(tile);
                        while(path[path.length-1].parent != undefined)
                        {
                                path.push(path[path.length-1].parent);
                        }

                        return path.reverse();
                }
}
```

So what just happened? The first thing we do is we get the time. This is simply for debugging purposes. It will tell us how long it took to find a path to the end. The next thing we do is we create two lists, one as our open list and the other as our closed list. Our open list will hold all the tiles that are in queue to be tested, while our closed list will hold all the tiles that have already been tested. To both of these arrays, we add the starting tile. A tile is defined by having an x and y position; a “g” value, which tells how costly (time wise), it is to take the route to that tile; an “h” value, which is a ‘heuristic’ value, meaning that it is guessing the distance to the end point; and finally a parent, which will be another tile object that is the tile that allowed this path to exist. This starting tile does not have a parent defined. This is because the starting tile does not need a parent. This will make more sense later.

After that, we define a reachedTarget object with a single variable, a true or false value. This is so that we can change the value of this object later from inside functions, due to the fact that objects are passed in by reference. We also create a pathTile variable. We will set this to the last tile in our completed path. We also get the width and height of our 2d map. Finally, we get into the meat of this function. We start a while loop that continues to run until reachedTarget is set to true, so it will continue to run as long as we haven’t reached the target. In this loop, we first sort the tiles in our open list based on their perceived value, in ascending order. Their perceived value is calculated by the current cost of their route plus the estimated distance to the target. Obviously this sorting will not get us the actual valuable tiles, the ones in our path, but it will get us what we think is the most valuable tile as far as the system knows so far.

Since our sorting goes in ascending order, than the first element in the open list will have the lowest perceived cost, meaning it has the highest value, so we will use this tile for this iteration. We call a function that tries to add all the adjacent tiles to the open list. In this function, we check each adjacent coordinate to see if it's within the bounds of the map and if it is empty or if it is the end tile, because if the end tile is blocked, we want our system to go to that tile anyway. If these conditions are valid, we try to calculate all the values for the tile. In the calcTile function, we check if the tile we are trying to calculate is in either our open or closed lists. If not, we check if the tile is the end tile, and if so, we set the reachedTarget value to true, and set the pathTile's values to that of this ending tile. Otherwise, we calculate the values for the tile and return the new tile object. If the tile is in either the open or closed list, we return undefined. So we take the returned value of the calcTile function and we try to push it into our open list. If the tile is undefined, meaning it was not calculated, we do not push it into the array.

Back to our main loop, after we add all adjacent tiles to the open list, we move the tile we are using to the closed list, and remove it from the open list, since now it has been calculated. Finally, we check to see if the open list is empty, meaning that we have checked all possible routes, and if so we sort the closed list by it's heuristic value, meaning its distance to the target, in ascending order. This means if we can't find any path to the target, move to the closest tile that we have found. We also return the path to the tile with the lowest distance to the target. We achieve this with our getPathFromLastTile function. It takes in the parameter of a tile. Since each tile has a parent, excluding the starting tile, we can loop through each tile until there is no parent, meaning that we have hit the starting tile. How do we do this? Well the first step is to create an array that will hold the path. We put the last tile in the path to start. Since we will be starting from the last tile and going towards the start, this array will need to be reversed. Now we have a while loop that checks if the parent of the most recent tile in the path (meaning the tile currently closest to the start based on the path) is undefined. Essentially this checks if the last tile added to the list is the start tile. So while we haven't found the start tile, we add the parent of the most recent tile to the list. This means that basically, it will cascade upwards until it finds the tile that has no parent. Finally we return the reversed array, so that the starting tile is at index 0 and the ending tile is at the last index.

So back to our main function, if the while loop breaks, meaning that we have reached the target, we get the path to the pathTile, which is the path we set to the solution of the find.

This is called the A* pathfinding algorithm. It is not a perfect algorithm, it may not get you the perfect path all the time, but for a quick solution, it can work most of the time. This implementation of the algorithm does not check for impossible targets, so you may wish to check if it ever exceeds a certain time limit, to avoid checking the entire map for a solution that does not exist.

Here is another example, that combines many features together to make a nice looking scenery.

```
<!DOCTYPE html>
<html>
        <head>
                <script type="text/javascript">
                  var Main =
      {
        heights : undefined
        , nextHeights : undefined
        , starMap : undefined
        , nextStarMap : undefined
        , canvas : undefined
        , c : undefined
        , offset : 0

        , init : function()
        {
          Main.heights = Main.generateHeights();
          Main.nextHeights = Main.generateHeights();
          Main.starMap = Main.generateStarMap();
          Main.nextStarMap = Main.generateStarMap();
```

```
    Main.canvas = document.getElementById('canvas');
    Main.canvas.width = window.innerWidth;
    Main.canvas.height = window.innerHeight;
    Main.c = canvas.getContext('2d');

    setInterval(Main.update, 25);
}

, update : function()
{
    if(Main.offset >= Main.canvas.width)
    {
        Main.offset = 5;
        Main.heights = Main.nextHeights;
        Main.nextHeights = Main.generateHeights();
        Main.starMap = Main.nextStarMap;
        Main.nextStarMap = Main.generateStarMap();
    }
    else
    {
        Main.offset += 5;
    }

    Main.drawHeights();
}

, generateHeights : function()
{
    var heights = [0];
    for(var iter = 0; iter < 10; iter++)
    {
        var newHeights = [];
        for(var i = 0; i < heights.length * 2; i++)
        {
            if(i % 2 == 1)
            {
                newHeights.push(heights[Math.floor(i / 2)]);
```

```
                    }
                    else
                    {
                       if(i === 0)
                       {
                                  newHeights.push(heights[Math.floor(i / 2)] / 2 +
(Math.random()-0.5)*2/(Math.pow(2, iter)));
                       }
                       else
                       {
                                  newHeights.push((heights[Math.floor(i / 2)-1] +
heights[Math.floor(i / 2)]) / 2 + (Math.random()-0.5)*2/(Math.pow(2, iter)));
                       }
                    }
                 }

                 heights = newHeights;
              }

              return heights;
           }

           , generateStarMap : function()
           {
              var starMap = [];
              for(var i = 0; i < 1000; i++)
              {
                         starMap.push({x : canvas.width * Math.random(), y :
canvas.height * Math.random()});
              }

              return starMap;
           }

           , drawHeights : function()
           {
                  var grd = Main.c.createLinearGradient(Main.canvas.width/2,
Main.canvas.height, Main.canvas.width/2, -100);
              grd.addColorStop(0, 'rgb(254, 160, 60)');
              grd.addColorStop(0.3, 'rgb(153, 125, 121)');
              grd.addColorStop(0.7, 'rgb(46, 50, 88)');
              grd.addColorStop(1, '#000000');
              Main.c.fillStyle = grd;
              Main.c.fillRect(0, 0, Main.canvas.width, Main.canvas.height);

              Main.c.fillStyle = "#FFFFFF";
              for(var i = 0; i < Main.starMap.length; i++)
              {
                              Main.c.fillRect(Main.starMap[i].x  -  Main.offset,
Main.starMap[i].y, 1, 1);
              }
              for(i = 0; i < Main.nextStarMap.length; i++)
              {
                        Main.c.fillRect(Main.nextStarMap[i].x  -  Main.offset  +
Main.canvas.width, Main.nextStarMap[i].y, 1, 1);
              }

              Main.c.beginPath();
              Main.c.moveTo(-Main.offset, Main.canvas.height/2);
              for(i = 0; i < Main.heights.length; i++)
              {
                               Main.c.lineTo((i  +  1)  /  Main.heights.length  *
Main.canvas.width - Main.offset, (Main.canvas.height/2)*Main.heights[i] +
Main.canvas.height / 2);
              }
                             //Main.c.lineTo(Main.canvas.width  -  Main.offset,
Main.canvas.height);
              for(i = 0; i < Main.nextHeights.length; i++)
              {
                          Main.c.lineTo((i  +  1)  /  Main.nextHeights.length  *
Main.canvas.width       -       Main.offset       +       Main.canvas.width,
(Main.canvas.height/2)*Main.nextHeights[i] + Main.canvas.height / 2);
              }
```

```
                Main.c.lineTo(Main.canvas.width, Main.canvas.height);
                Main.c.lineTo(0, canvas.height);
                Main.c.fillStyle='#000000';
                Main.c.fill();
            }
        };
                    </script>
            </head>
            <body style='overflow: hidden; position: fixed; margin:0; padding:
0' onload='Main.init();'>
                <canvas id='canvas' width='1600' height='900' style='border: 1px
solid black'></canvas>
            </body>
</html>
```

Let's begin with the HTML stuff. We create the html tags, and the head, which we place our script into. Below the head, we create our body. We set a style, where it prevents things like scrollbars, called overflow, and prevents scrolling by making its position fixed. We also disable its margin and padding to make our body right up against the top left. Finally, we set the body's onload event to call our script's init function, which we will get to in a minute. We also define the canvas where we will be drawing our scenery. We give it an id of canvas, a width of 1600 and a height of 900. We also set its style to have a black border with a width of 1 pixel.

In our script, we create a Main object. This simply makes it cleaner if we ever want to add other features like input or pathfinding or whatever else. It is cleaner to have objects that separate things than have all of your functions randomly grouped together. It also adds the element of scope, so you can have two variables with the same name in separate objects that do different things. In this Main object, we "define" several variables. We set each of these variables to undefined, which sounds like an oxymoron—we defined the variable as undefined—but this will allow us to remember which values we will have defined later. We have 6 variables: heights, which tells us the heights of the current map; nextHeights, which tells us the heights of the map we will be scrolling to; starMap, which tells us the stars of the current map; nextStarMap, which tells us the stars of the map we will be scrolling to; canvas, which stores a reference to the canvas DOM element; and c, which stores the 2d context of the canvas. We also define offset, which will act as our scrolling factor.

In our init function, we set up our heights and nextHeights to the returned value of our generateHeights function. This function starts by creating an array with a single element set to 0. This array holds all the heights in our terrain as a 1d array, where the value is the height at that point. We assume that the start and end of the heightmap must be at a height of 0. Our array holds the end height of 0, but omits the start height. This is to make the array always powers of 2. Next, we make a loop that does 10 iterations. This loop will increase the array's size by a factor of two each iteration. In this loop, we create another array. We then start another loop that runs for twice the length of the original array. We then check if this inner loop is an odd number. If it is, then that means we take the element at that index halved from the original array and add it to the new array. If the inner loop index is not an odd number, we check to see if the index is 0. If it is 0, than we get the average between the first element in the original heightmap and 0. This means we get the middle point between the starting point, and the first point. We then add a random offset to this point, which goes between 1 and -1 and divide it by 2 to the power of the number of iterations. This makes it so that as the terrain divides, it gets more unique due to the random offset. Because we divide by 2 to the power of the number of iterations, this means that as the number of heights increases, the random offset gets smaller. This is so that the terrain does not go all over the place. Now if the loop index is not 0, we do the same thing, except get the average between the height to the left, and the height to the right of the selected height. Basically, we are putting a new, slightly-different height between the old two heights. Finally, once we have made the new heights array, we replace the old heights array with the new one, so that the next iteration will make this new array more unique than itself. Finally, once we have completed our iterations, we return this heightmap. The next thing we do in our init function is we set our starMap and nextStarMap to the returned value of the generateStarMap function.

This function creates an array and starts a loop for 1000 times. In this loop, we add a star object to the array, which contains an x position, and a y position, which are both generated randomly between the canvas' width and height. Finally it returns this array. Our init function also gets the canvas element, and gets its context. Finally, it sets an interval to run the update function every 25 milliseconds. In our update function, we check to see if our offset is equal to the canvas width (because we are scrolling to the right), and if so, we change the offset to start at the left again, move the nextHeights to be the heights, as we have scrolled to the right enough to no longer have the old heights in view, and we generate a new nextHeights. We do the same thing with the starMap and nextStarMap.

If the offset is not equal to the canvas width, we simply add 5 (an arbitrary speed for the camera). Finally, no matter what, we draw the scene using our drawHeights function. In this function, we draw a gradient which will act as our sunset, because sunsets are pretty. We set the fillStyle of the canvas' context to the grid and we fill the entire canvas with this background. After that, we set the fillStyle to be white, as we are about to draw the starMap. To do this, we loop through the starMap, and then for each we fill a 1 x 1 rectangle at the position of the star, except shifted over left by the offset amount. This is so that as the offset increases, the entire map will shift over left. This means that it will start completely in frame, and move left until it is completely off frame (because offset will max out at the width of the canvas). This excludes the background sunset, because the background sunset will be the same no matter what our offset will be. We also draw the nextStarMap, the same way as the normal starMap, except we also shift it right with an amount of the canvas width. This means that it will start completely off frame to the right and then as offset increases, it will move further left, until it is completely in frame, and then we would generate a new star map.

After drawing the star maps, we begin a path, and start it at the left side of the map, meaning the offset value going left. and at half the canvas height, as we start at 0. Then we loop through the heights array, and in each iteration, we draw a line to each height value, where the x value is the percentage through the heights array we are looking at, multiplied by the width of the canvas, meaning that the heights will be evenly spaced along the width of the canvas. The height is half the canvas multiplied by the height, so if the height is -1, it will be at the lowest height, and if the height is 1, it will be at the highest height. We do the same thing with the nextHeights, except we shift it over just as we shift the nextStarMap. Finally, we finish our terrain by drawing lines to the bottom right of the canvas and the bottom left of the canvas. We then fill it with the colour black. This makes a rather pretty, infinitely-scrolling, randomly generated terrain.

Chapter 18

In Closing

This is the start of your journey as a Javascript programmer. You have barely scratched the surface with this guide as learning the syntax and conventions of a language is just the beginning. The most important part of programming is the logical aspect of it. Sure, you may know how to loop through an array of variables like a list of shopping items but if someone asks you to process an image using your knowledge of programming, and with the help of an API and some thinking, you can figure out how you are able to invert colors of an image, flip it, rotate it, scale it, etc.

The real programming comes in the logical portion of the mind. It's similar to when you're learning any other language, like English for example. You may understand the grammar rules and the conventions like adding periods to the end of sentences, but to be able to write clean and logical thought-out and structured essays is where the true skill lies. The same concept applies to programming where the person writing the code, must know how to apply his knowledge of the rules in the considered language, like Javascript, and use it to his advantage to come up with neat programs.

The knowledge and understanding of programming is truly great because it's the closest thing to having a power. You can literally create something out of an empty notepad, from scratch and have it function to do things you want it to do. Whether it be a bot to analyze the stock market and come up with predictions or creating a game. That choice is yours.

In this guide, you have learned the fundamentals of Javascript. You haven't learned all the possible methods that can be used in the language, but that isn't the point. The point of this guide was to set you on a journey to discover objects and methods that you need in order to help you to create programs that you desire. You have been given the optimum knowledge to understand reading an API and be able to understand what it is saying and adding to your code.

Good luck as a new-born Javascript programmer!

Eprogramy

Preview of JAVA CRASH COURSE - The Ultimate Beginner's Course to Learning Java Programming in Under 12 Hours

Introduction

Welcome to Your New Programming Language

So, you've decided to learn Java Programming? Well, congratulations and welcome to your new Programming Language! You're going to love it!

In Eprogramy we believe that the foundation of a good education is to provide the tools able to open the doors of the future. It is indisputable that the world has entered an era in which we all have everything at the fingertips. Social needs have changed. Thus, today communication has changed. Communication opens doors, so it is our challenge to learn how to communicate. In Eprogramy we want you to learn how to communicate with a new language: The programming language. This language allows us to use our PC or notebook to create.

Maybe the road is difficult and hard, but we will give the reader the tools to make learning enjoyable and fruitful. We'll show all aspects necessary to learn how to program. From the installation of software to the solution of the most common programming problems and much, much more. Always with the help of numerous examples that will contribute to a better understanding

We know what we do and believe that you are fully capable of incorporating our teachings.

The doors of the future are here. Let's go step by step together.

Let´s get started!

Tim Markus
Eprogramy Team

Chapter 1

Java Programing Language

History of Java

Java is a programming language that was first publicly available in 1995, created by James Gosling from Sun Microsystems (Sun) in 1991. Sun Microsystems was then inherited by Oracle, and is the corporation in charge of the programming language's faith. Java then became open source available under the GNU General Public License (GPL).

The language gets much of its syntax from C and C++ but isn't as powerful because it requires less for the user to do (less customization but more simple). Something like garbage collection, the process of reducing memory being used from the program, is automated in Java.

It was originally designed for interactive television but surpassed the technology and design of the digital cable television industry at the time. There were five principles that were used in the creation of the Java programming language:

1. It must be "simple, object-oriented and familiar"

2. It must be "robust and secure"

3. It must be "architecture-neutral and portable"

4. It must execute with "high performance"

5. It must be "interpreted, threaded, and dynamic"

Another important design goal to note is portability, which is was a key factor in Java's sudden popularity. The portability in this context means that the code written in the Java platform can be executed in any combination of hardware and operating system.

Unfortunately, Java has a reputation for being a slower programming language (requiring more memory) than other languages like C++ but as time went, Java 1.1 was introduced and program execution times were significantly improved.

Java was built exclusively as an object-oriented programming language, which doesn't mean much right now, but it will later in the guide. Object-Oriented programming allows for efficient, organized and powerful code to be created and will be seen throughout this guide.

What is Java?

Java is a programming language that has multi-platform capability, meaning that you can program Java for any type of device, whether it is an Android phone, a Windows computer or an Apple product. Due to the flexibility of Java, it has made it one of the most popular programming languages used through the globe by many programmers. Java can be used to create web applications, games, windows applications, database systems, Android apps and much more.

Java is different from other programming languages because of its simplicity and powerful nature. That combination makes the Java programming language great to use. Java is a simple programming language because it doesn't expect too much from the user in terms of memory management or dealing with a vast and complex hive of intricate classes extending from each other. Although this doesn't make much sense right now, it will make sense when reaching the point of learning inheritance in Java.

A Java program is run through a Java Virtual Machine (JVM) and is essentially a software implementation of an Operating System which is used as a way to execute Java programs. The compiler (process of converting code into readable instructions for the computer) analyzes the Java code and converts it in to byte code, allowing the computer to understand the instructions issued by the programmer and execute them accordingly.

When downloading and installing Java, the distribution of the platform comes in two ways; the Java Runtime Environment (JRE) and the Java Development Kit (JDK). The JRE is essentially the Java Virtual Machine (JVM) where the Java programs will run on. JDK on the other hand is a fully featured Software Development kit for Java which includes the JRE, compilers, tools, etc.

A casual user wanting to run Java programs on their machine would only need to install JRE as it contains the JVM which allows Java programs to be executed as explained before. On the other hand, a Java programmer must download JDK in order to actually program Java programs.

Chapter 2

Installation of Java

In order to install Java on to a machine, you must download the following:

1. IDE for Java Developers

2. Java JDK

The download of these two tools will put you on your way to becoming a Java programmer. An IDE (integrated development environment) is a packaged application program used by programmers because it contains necessary tools in order to process and execute code. An IDE contains a code editor, a compiler, a debugger, and a graphical user interface (GUI). There are many different type of IDE's but the most commonly used ones are:

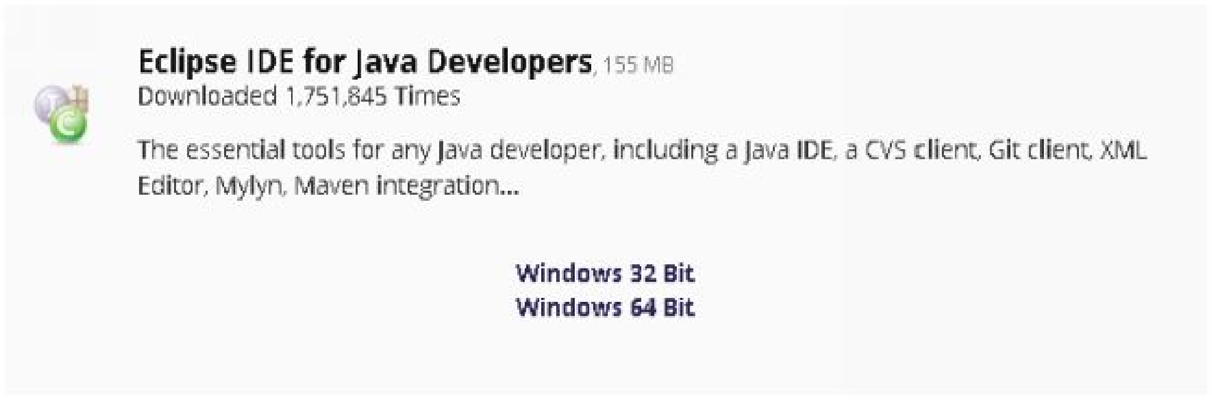
1. Netbeans

2. Eclipse

In this guide, it would be recommended to use Eclipse because of its simplistic nature. In order to download Eclipse, please use the following link:

https://eclipse.org/downloads/

Once you have reached this link, you will have to find this:


Eclipse IDE for Java Developers, 155 MB
Downloaded 1,751,845 Times
The essential tools for any Java developer, including a Java IDE, a CVS client, Git client, XML Editor, Mylyn, Maven integration...

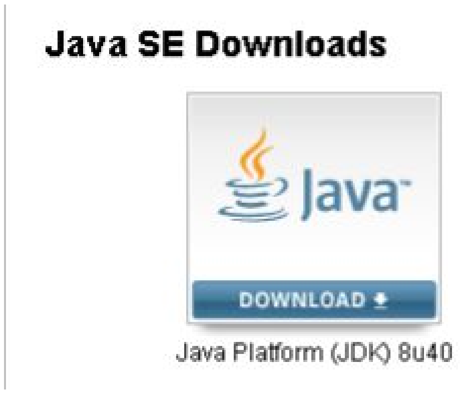
Then, select either the Windows 32 Bit OS or Windows 64 depending on the type of OS / processor you have.

Once the IDE has been installed, we'll move on to downloading and installing the JDK which will allow us to interact with a coding editor in order to execute and create Java code.

To download the JDK, go to the following link:

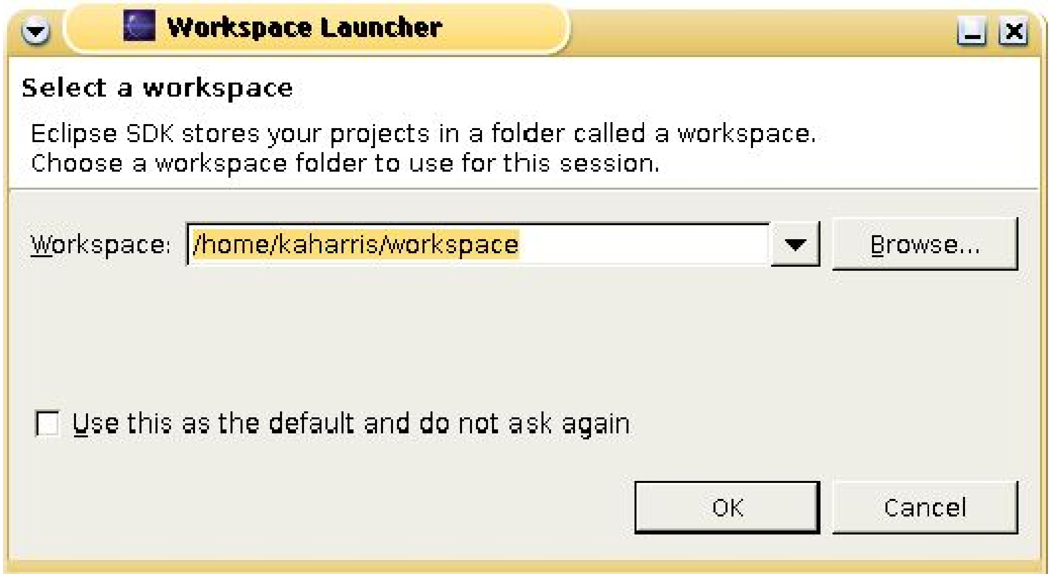
http://www.oracle.com/technetwork/java/javase/downloads/index.html

Upon entering this link, find this figure:



Once you've installed the JDK, you are able to launch Eclipse accordingly.

The folder that was extracted from the Eclipse download will contain an eclipse.exe that can be launched. Once this has been launched, you will be met with the following figure:

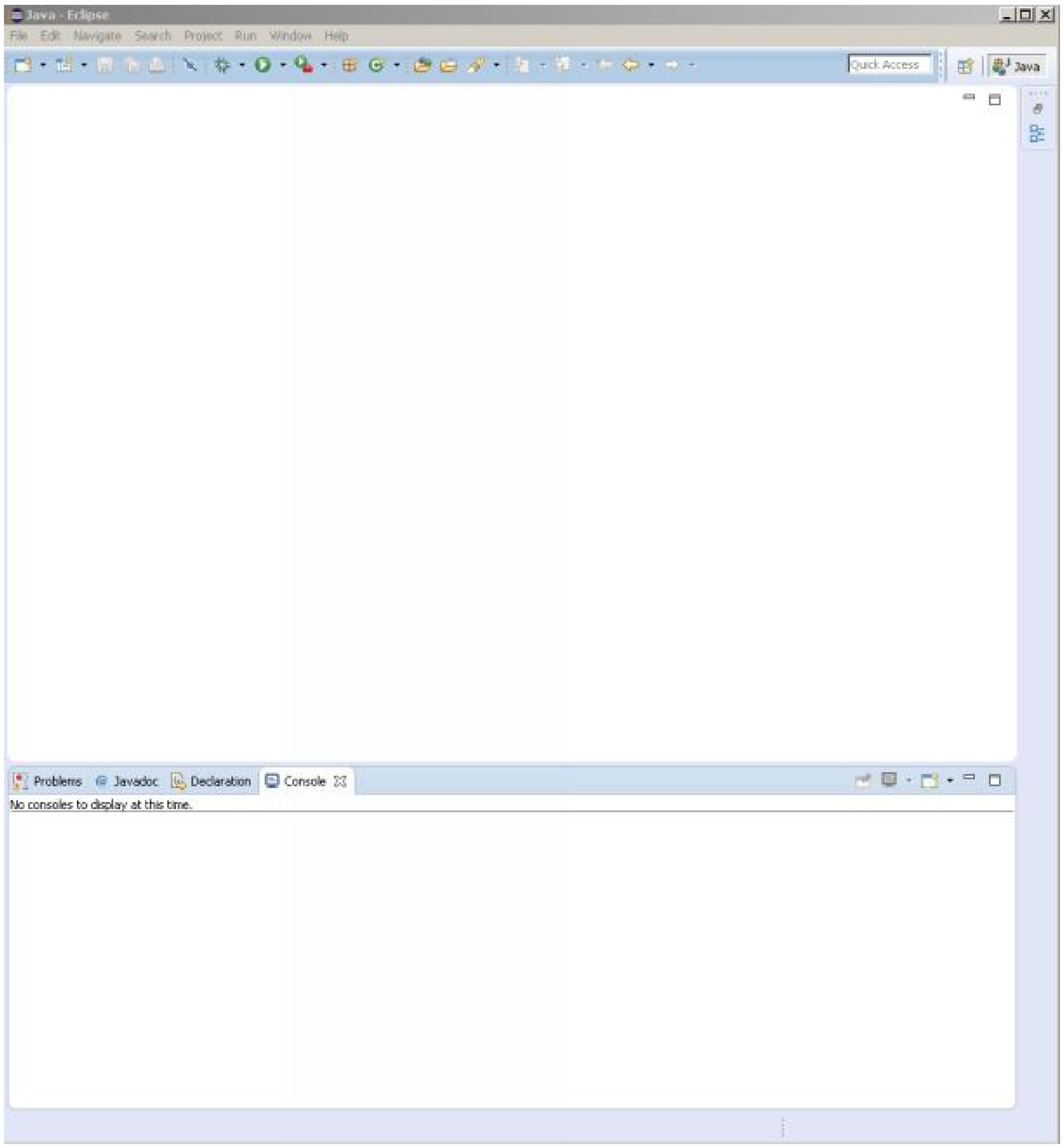


Eclipse SDK stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: /home/kaharris/workspace

All this prompt is asking for is where to set up the output directory for where all the code written by you is going to be saved. Once this has been selected accordingly, click “OK” to continue.

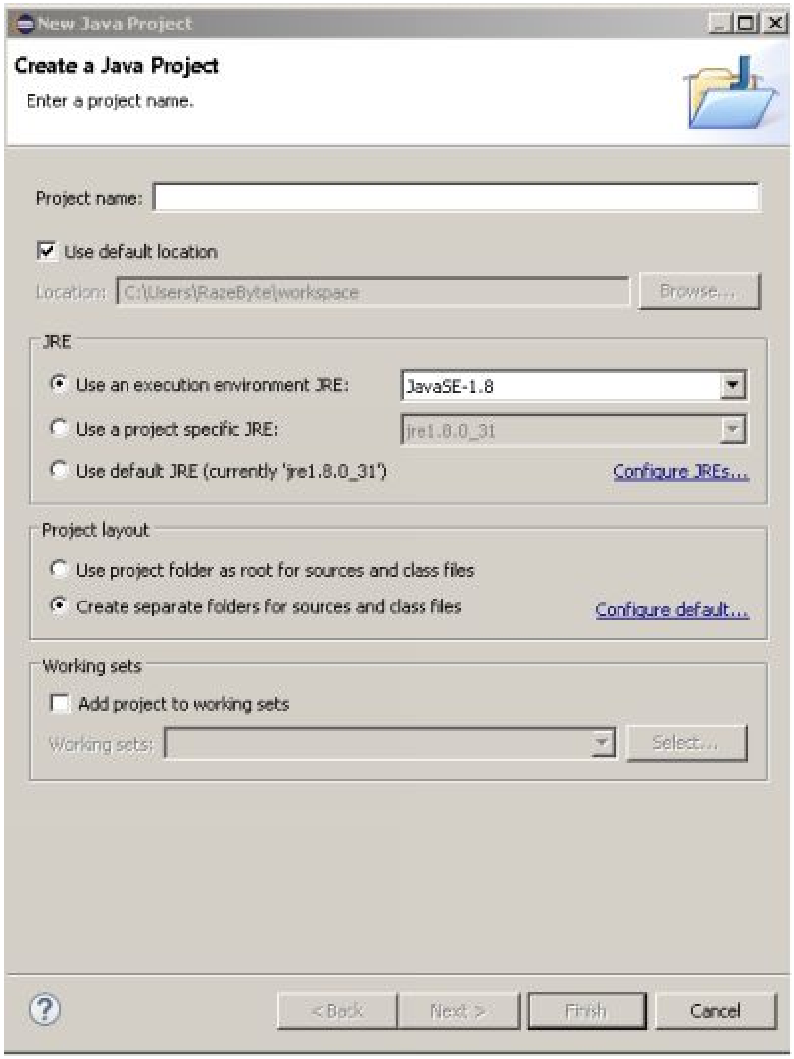
You should now be on this screen (or something similar):



No consoles to display at this time.

Instructions:

1. Click File
2. Click New → Java Project



Project name:

[✓] Use default location

Location: C:\Users\RazeByte\workspace

JRE

(●) Use an execution environment JRE: JavaSE-1.8

( ) Use a project specific JRE: jre1.8.0_31

( ) Use default JRE (currently 'jre1.8.0_31')

Project layout

( ) Use project folder as root for sources and class files

(●) Create separate folders for sources and class files

Working sets

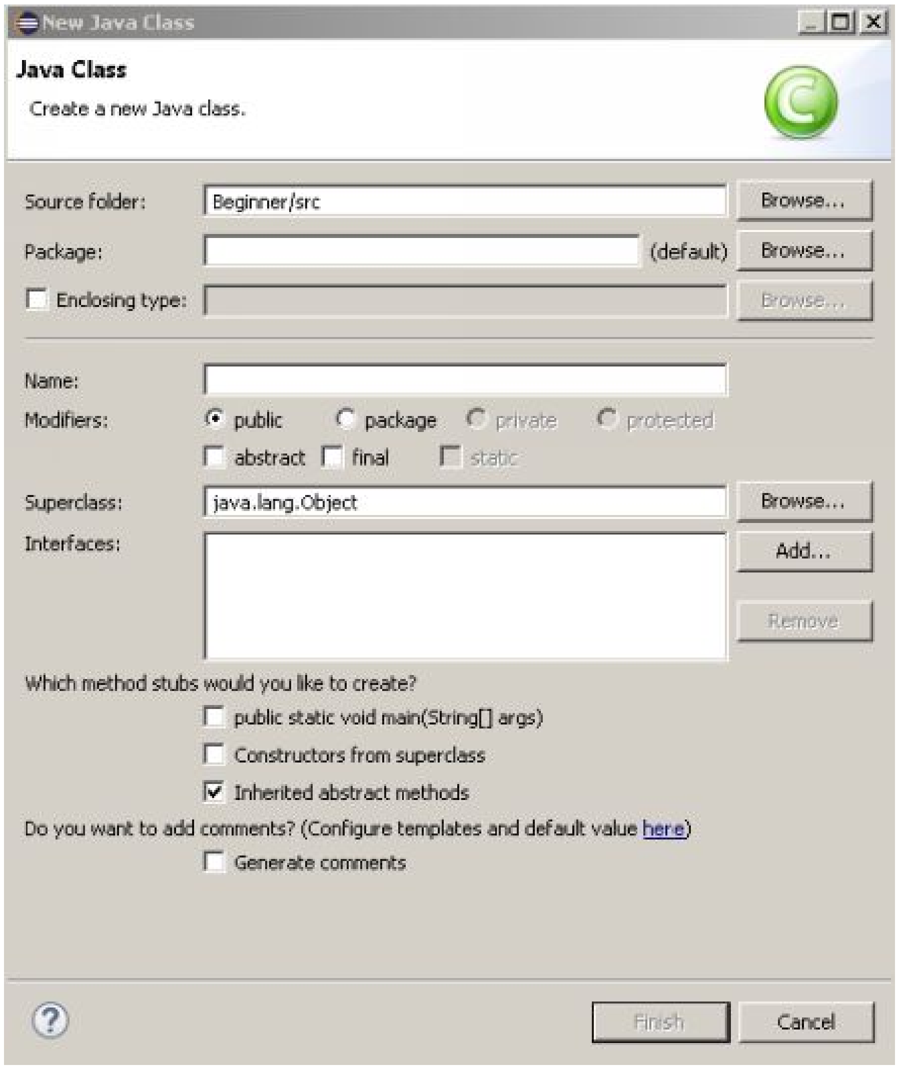
[ ] Add project to working sets

Working sets:

3. Type in a project name in the “Project name:” field

4. Click “Finish”

5. Now right click “src” → New → Class



Source folder: Beginner/src
Package: (default)
[ ] Enclosing type:
Name:
Modifiers: [✓ public] package private protected
[ ] abstract [ ] final [ ] static
Superclass: java.lang.Object
Interfaces:
Which method stubs would you like to create?
[ ] public static void main(String[] args)
[ ] Constructors from superclass
[✓] Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
[ ] Generate comments

6. Fill in the name field to anything with *letters* and no special characters. Example: “ThisIsAClass”

7. Click Finish

8. You will now be presented with a screen that says:

```
public class ThisIsAClass {

}
```

You have successfully completed the installation of Java!

About the author

Eprogramy Academy was created by a group of professionals from various areas of IT with a single purpose: To provide knowledge in the 3.0 era.

Education is changing as well as social needs. Today, in the era of information, education should provide the tools to create and to solve problems in a 3.0 world.

At Eprogramy we understand this and work to give people appropriate responses in this context.

Keeping this objective in mind, we offer a wide variety of courses to teach the basics of many programming languages. We believe that anyone can learn a programming language and apply the lessons in order to solve problems. In our academy we provide the essential tools to allow everyone to incorporate into the daily life a set of solutions obtained through programming.

Possibilities and solutions are endless.

In short, at Eprogramy we are committed to help everybody to decodify the messages of the future.