

# FULL STACK WEB DEVELOPER

## PART VI

# JavaScript

from basics to advanced



Panos M.

# Full Stack Web Developer Part VI: JavaScript

Panos Matsinopoulos

This book is for sale at <http://leanpub.com/full-stack-web-developer-part-vi-javascript>

This version was published on 2019-08-13



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2019 Tech Career Booster - Panos M.

# Contents

<b>The Bundle and the TCB Subscription . . . . .</b>	<b>1</b>
Part of Bundle . . . . .	2
Goes with a TCB Subscription . . . . .	3
Each TCB Subscription Goes with the Bundle . . . . .	4
<b>Credits To Photo on Cover Page . . . . .</b>	<b>5</b>
<b>About and Copyright Notice . . . . .</b>	<b>6</b>
<b>JavaScript and jQuery . . . . .</b>	<b>7</b>
1 - Hello World . . . . .	8
Summary . . . . .	8
Learning Goals . . . . .	9
Introduction . . . . .	10
Our JavaScript Testbeds . . . . .	10
Writing our First Program . . . . .	10
Closing . . . . .	22
Tasks and Quizzes . . . . .	23
2 - Lexical Structure . . . . .	24
Summary . . . . .	24
Learning Goals . . . . .	24
Let's Start . . . . .	24
Case Sensitivity . . . . .	24
Comments . . . . .	25
Literals . . . . .	25
Identifiers and Reserved Words . . . . .	26
Terminating Statements . . . . .	27
Tasks and Quizzes . . . . .	28
3 - Types, Values and Variables . . . . .	29
Summary . . . . .	29
Learning Goals . . . . .	30
Introduction . . . . .	31
Literals and Variables . . . . .	31
Assignment Operator . . . . .	32
Numbers . . . . .	34
Dates and Times . . . . .	35
Text . . . . .	36
Pattern Matching . . . . .	42

## CONTENTS

Boolean Values . . . . .	46
null and undefined . . . . .	48
Tasks and Quizzes . . . . .	49
4 - Expressions And Operations . . . . .	50
Summary . . . . .	50
Learning Goals . . . . .	53
Introduction . . . . .	54
Primary Expressions . . . . .	54
Array Initializers . . . . .	54
Object Initializers . . . . .	56
Function Definition Expressions . . . . .	59
Operator Overview . . . . .	60
Number Of Operands . . . . .	61
Example Uses of Operators . . . . .	61
Operator Precedence . . . . .	63
Relational Expressions . . . . .	64
Logical Expressions . . . . .	68
Assignment With Operation . . . . .	70
Tasks and Quizzes . . . . .	70
5 - Statements . . . . .	71
Summary . . . . .	71
Learning Goals . . . . .	72
Introduction . . . . .	73
Expression Statements . . . . .	73
Compound Statements . . . . .	73
Declaration Statements . . . . .	74
Conditionals . . . . .	74
Loops . . . . .	80
Jump statements . . . . .	85
Tasks and Quizzes . . . . .	86
6 - Objects . . . . .	88
Summary . . . . .	88
Learning Goals . . . . .	89
Introduction . . . . .	90
Creating Objects . . . . .	90
Querying and Setting Properties . . . . .	96
Inheritance . . . . .	98
Methods . . . . .	102
Closing Note . . . . .	103
Tasks and Quizzes . . . . .	104
7 - Arrays . . . . .	105
Summary . . . . .	105
Learning Goals . . . . .	106
Introduction . . . . .	107
Creating Arrays . . . . .	107
Reading and Writing Array Elements . . . . .	109

## CONTENTS

Adding And Deleting Elements . . . . .	111
Array Methods . . . . .	118
Strings as Arrays . . . . .	132
Tasks and Quizzes . . . . .	133
8 - Functions . . . . .	135
Summary . . . . .	135
Learning Goals . . . . .	136
Introduction . . . . .	137
Defining Functions . . . . .	137
Defining And Calling Function At the Same Time . . . . .	139
The return statement . . . . .	140
Nested Functions . . . . .	141
Invoking a Function . . . . .	142
Function Arguments and Parameters . . . . .	144
Using Objects As Arguments . . . . .	146
Recursive Functions . . . . .	148
Throwing Errors . . . . .	150
Catching Errors . . . . .	153
Throwing Custom Errors . . . . .	155
Tasks and Quizzes . . . . .	157
9 - JavaScript In Web Browsers . . . . .	159
Summary . . . . .	159
Learning Goals . . . . .	159
Introduction . . . . .	159
Execute JavaScript on a Web Page . . . . .	159
Calling JavaScript From Files . . . . .	162
The Window Object . . . . .	163
setTimeout() . . . . .	165
document . . . . .	166
document.getElementById() . . . . .	166
setInterval() and clearInterval() . . . . .	168
Dialog Boxes . . . . .	169
Opening and Closing Windows . . . . .	171
Tasks and Quizzes . . . . .	173
10 - Introduction to DOM (Document Object Model) . . . . .	175
Summary . . . . .	175
Learning Goals . . . . .	175
DOM . . . . .	176
Selecting Elements of The DOM . . . . .	177
Element Content As HTML . . . . .	182
Tasks and Quizzes . . . . .	184
11 - Introduction to Event Handling . . . . .	188
Summary . . . . .	188
Learning Goals . . . . .	188
Event-driven Programming . . . . .	189
Event Type . . . . .	189

## CONTENTS

Event Target . . . . .	189
Event Handler or Event Listener . . . . .	190
Event Object . . . . .	190
Event Propagation . . . . .	190
Default Actions and Cancelling an Event . . . . .	191
Registering Event Handlers . . . . .	191
Event Propagation . . . . .	195
Stopping the Event Propagation . . . . .	198
Tasks and Quizzes . . . . .	198
<b>12 - jQuery Basics . . . . .</b>	<b>200</b>
Summary . . . . .	200
Learning Goals . . . . .	201
Introduction . . . . .	201
Including jQuery to your HTML document . . . . .	202
jQuery and \$ . . . . .	204
Selecting And Actioning on HTML Elements . . . . .	204
Queries and Query Results . . . . .	207
Getting and Setting HTML Attributes . . . . .	212
Removing And Adding Content . . . . .	216
Getting and Setting CSS Attributes . . . . .	220
Getting and Setting CSS Classes . . . . .	224
Getting and Setting HTML Form Values . . . . .	227
Inserting and Replacing Elements . . . . .	231
<i>strict</i> mode . . . . .	239
Octal literals . . . . .	240
Reserved words . . . . .	241
Tasks and Quizzes . . . . .	242
<b>13 - JavaScript Workers . . . . .</b>	<b>244</b>
Summary . . . . .	244
Learning Goals . . . . .	244
Web Workers . . . . .	244
Prime Numbers Example . . . . .	245
Off-Main Thread Image Manipulation . . . . .	247
Features Available To Workers . . . . .	253
Handling Errors . . . . .	254
Shared Workers . . . . .	255
Closing Note . . . . .	256
Tasks and Quizzes . . . . .	256
<b>14 - ES6 - ECMAScript 2015 - Advanced JavaScript . . . . .</b>	<b>258</b>
Summary . . . . .	258
Learning Goals . . . . .	258
Introduction . . . . .	259
let Statement . . . . .	259
const Statement . . . . .	260
Exponentiation Operator . . . . .	261
Template Strings . . . . .	262

## CONTENTS

Arrows . . . . .	263
Enhanced Object Literals . . . . .	264
Destructuring Assignment . . . . .	265
Object Rest/Spread Properties . . . . .	267
Function Parameter Default Values . . . . .	267
Object.assign() . . . . .	268
Classes . . . . .	268
Sets and Maps . . . . .	274
Symbols . . . . .	278
Iterators, Generators and Iterables . . . . .	279
Modules . . . . .	283
Promises . . . . .	294
Promise Example: fetch() . . . . .	300
async/await . . . . .	301
Closing Note . . . . .	303
Tasks and Quizzes . . . . .	303

# **The Bundle and the TCB Subscription**

## Part of Bundle

This book is not sold alone. It is part of the bundle [Full Stack Web Developer](#).

## **Goes with a TCB Subscription**

When you purchase the bundle, then you have full access to the contents of the [TCB Courses](#).

## Each TCB Subscription Goes with the Bundle

Moreover, this goes vice-versa. If you purchase the subscription to the [TCB Courses](#), then you are automatically eligible for the [Full Stack Web Developer](#) bundle.

# Credits To Photo on Cover Page

We have designed the cover page, but the photo in the middle is the creation of our friend Telis Marin. He is an amateur photographer. He doesn't have an official Web site where you could find more of his amazing photos. Hence, if you want to see more of his work, the only way you can now do it is by making him an FB Friend [here](#).

Telis Marin is an author, a publisher [Edizioni Edilingua](#) and a teacher trainer. He has written more than 20 books for learning Italian, which are used by schools and universities in over 80 countries.

# About and Copyright Notice

Full Stack Web Developer - Part VI - JavaScript 1st Edition, August 2019

by Panos M. for Tech Career Booster (<https://www.techcareerbooster.com>)

Copyright (c) 2019 - Tech Career Booster and Panos M.

All rights reserved. This book may not reproduced in any form, in whole or in part, without written permission from the authors, except in brief quotations in articles or reviews.

**Limit of Liability and Disclaimer of Warranty:** The author and Tech Career Booster have used their best efforts in preparing this book, and the information provided herein “as is”. The information provided is delivered without warranty, either express or implied. Neither the author nor Tech Career Booster will be held liable for any damages to be caused either directly or indirectly by the contents of the book.

**Trademarks:** Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

For more information: <https://www.techcareerbooster.com>

# **JavaScript and jQuery**

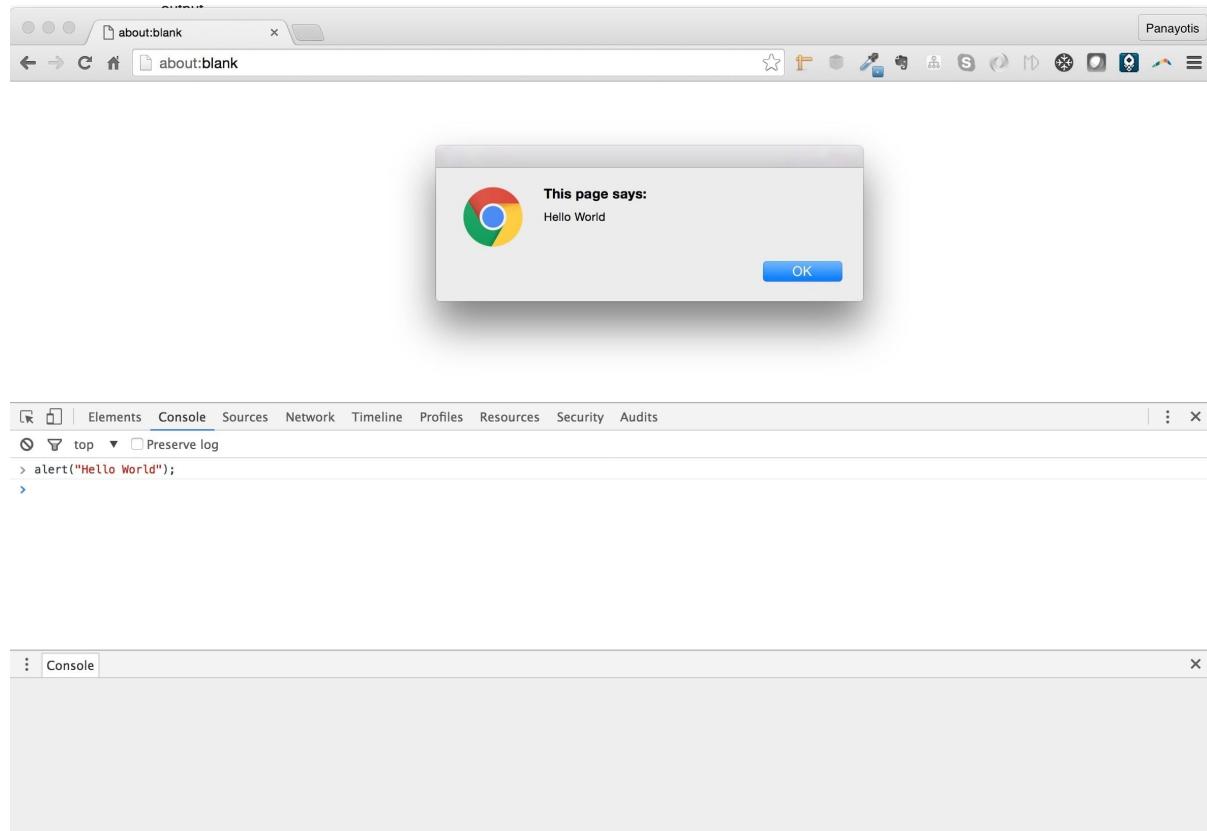
This section will teach you one of the most popular programming language out there. JavaScript. It is an absolute tool that you have to learn if you want to be a very good Web developer. JavaScript is the language that is used by Web browsers in order to add dynamic behaviour on the HTML/CSS static part of the Web page. However, it is not only a Web browser programming language. It is also used, nowadays, on the back-end tier of the Web development too. For that reason, this chapter will give you a broader knowledge of JavaScript. It will also teach you jQuery, which is a very popular JavaScript framework.

# 1 - Hello World

## Summary

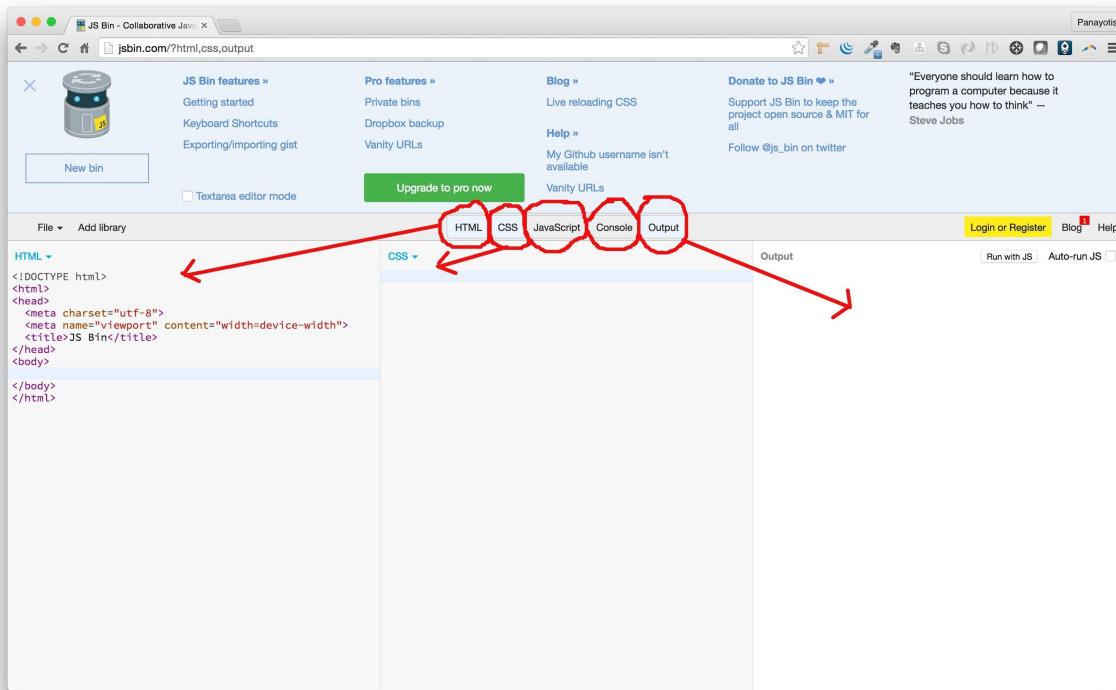
This chapter is your first encounter with JavaScript. JavaScript is the most popular programming language in the world of browsers and a language that all Full-Stack Web Developers needs to know very well.

In this chapter, you will learn how you can write JavaScript programs inside the Chrome Developer Tools:



JavaScript in Chrome Developer Tools

Then, you will learn to use JS Bin, which is a fantastic online tool that allows you to write HTML, CSS and JavaScript:



### JS Bin for Online Web Development

Finally, you will learn how you can install Node.js and execute JavaScript statement inside Node.js environment.

A screenshot of a terminal window titled 'panayotismatsinopoulos ~ \$ node'. It displays the following command and output:

```
panayotismatsinopoulos@~ $ node
> console.log("Hello World!");
Hello World!
undefined
> 
```

### JavaScript on Node.js Environment

## Learning Goals

1. Learn about some of the most popular JavaScript testbeds.
2. Learn how to write a “Hello World!” JavaScript program on
  1. Chrome Developer Tools Console
  2. JS Bin

### 3. [Node.js](#)

3. Learn how you can write HTML page, with CSS and JavaScript support on JS Bin.
4. Learn how you can include various third party libraries in JS Bin, like Twitter Bootstrap.
5. Learn how to download and install Node.js.
6. Learn how to start and exit the Node.js console.

## Introduction

This is your start to real programming and we will use JavaScript as our programming language. JavaScript is a dynamically typed script language. It is very popular because it is the language used by all browsers, to run programs on the client side. However, the last years has also become very popular as a backend language too, due to [Node.js](#).

If you want to become a very good Full-Stack Web Developer, you need to learn JavaScript very well.

## Our JavaScript Testbeds

In order for us to learn JavaScript, we are going to use an environment that will allow to execute our JavaScript code. We will use the following testbed environments:

1. Google Chrome Developers Console
2. [JS Bin](#)
3. [Node.js](#)

## Writing our First Program

Traditionally, when one starts to learn a new programming language, the first program that he writes, is a “Hello World” program. It is a program in which the phrase “Hello World” is somehow presented to the user of the program.

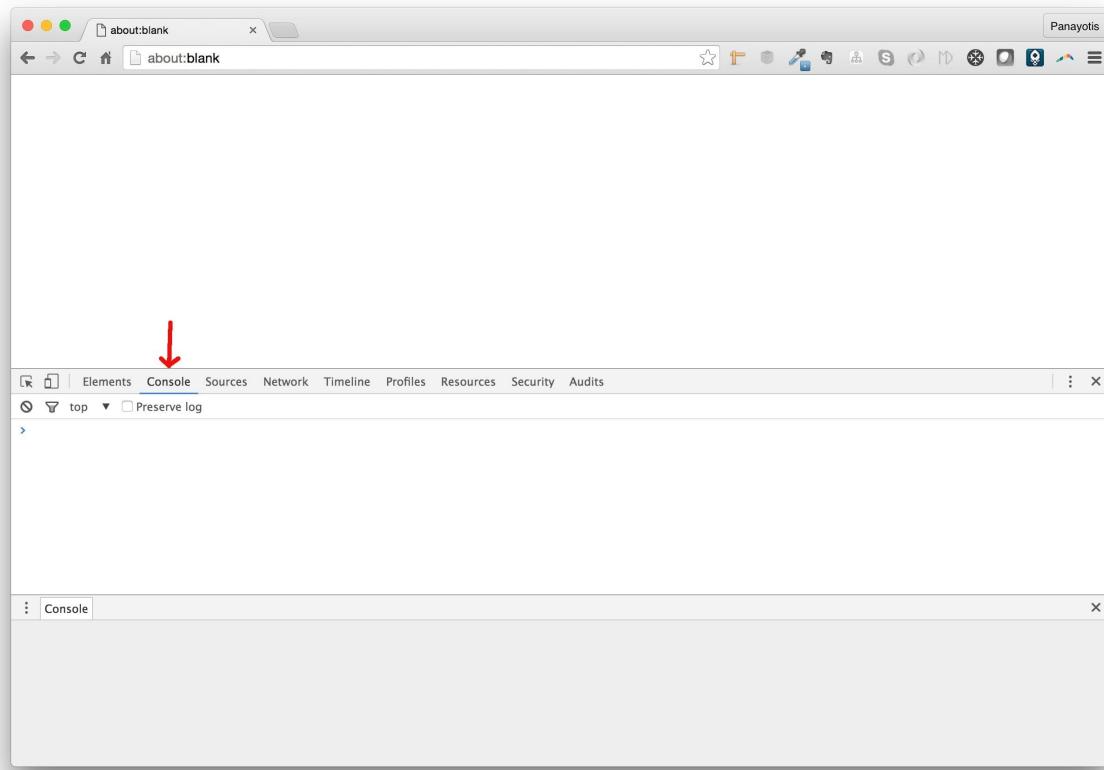
This is what we are going to do here with JavaScript.

### Hello World Program

#### Google Chrome Developers Console

Initially, we will write our hello world program using the Google Chrome Developers Console.

Open your Chrome browser and then start the developer tools.



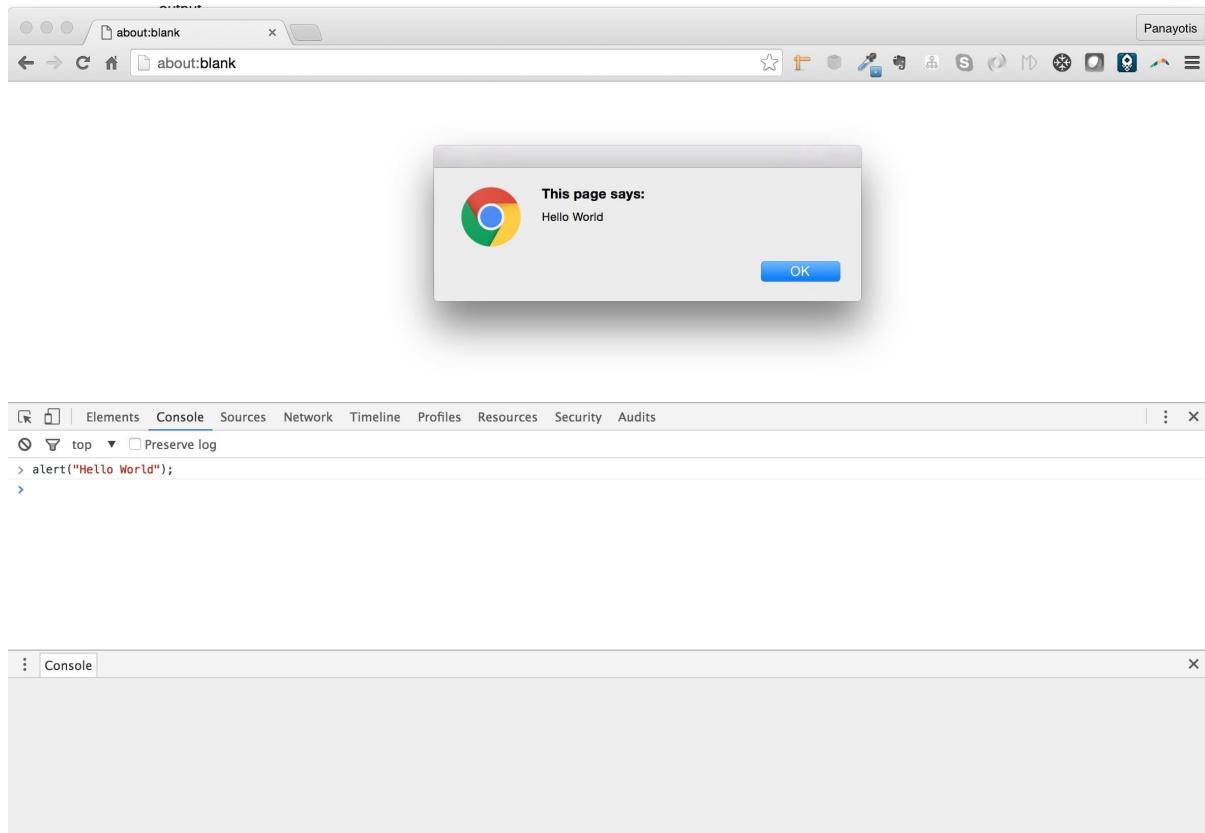
#### Chrome Developer Tools - Console Tab Selected

Make sure that the `Console` tab is selected. If not, click on that. You should see the cursor blinking in front of the JavaScript command prompt. If everything is ok, then write the following JavaScript command and click on the `Return / Enter` key on your keyboard (like when you give Operating System commands on the terminal).

```
1 alert("Hello World");
```

(the above code snippet online)

When you do that, you will see a popup with the “Hello World” phrase printed on it.



### Alert Popping Up

Congratulations! You have just written your first computer program. Your first JavaScript statement.

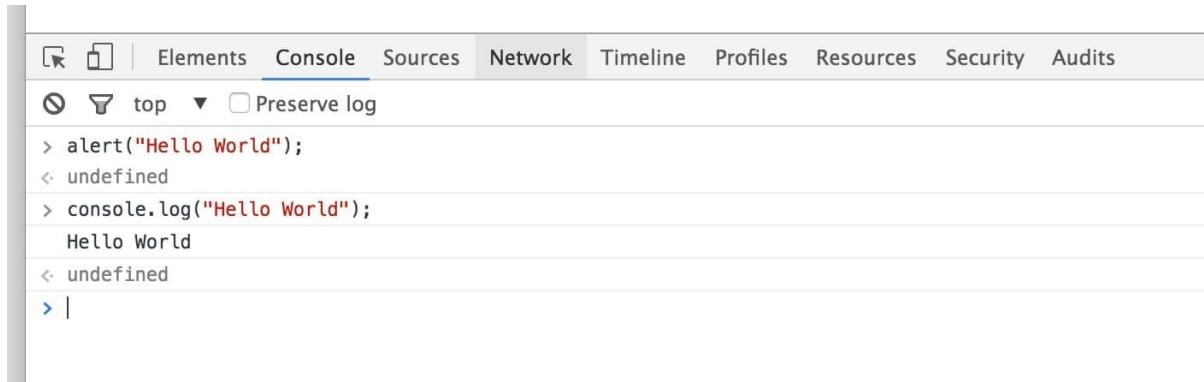
As we said earlier, JavaScript is a script language. This means that the statements are being executed one-by-one. The statement here is `alert("Hello World");`. The ; is necessary to denote to JavaScript interpreter the end of the JavaScript statement. When you click on Enter key, the JavaScript statement is submitted to the browser JavaScript execution engine and it is executed immediately. This is how Google Chrome Developers Console works. And then you are being transferred to a new console prompt, and you can give your next JavaScript command.

Let's do that. Let's give our next JavaScript command:

```
1 console.log("Hello World");
```

(the above code snippet online)

When you type in the above and you click on Enter to submit this JavaScript statement back to the execution engine, then you will see this:



The screenshot shows the Chrome DevTools interface with the 'Console' tab selected. The console log output is displayed:

```
> alert("Hello World");
< undefined
> console.log("Hello World");
Hello World
< undefined
> |
```

### console.log command

The `console.log(".....");` command outputs its argument to the console.

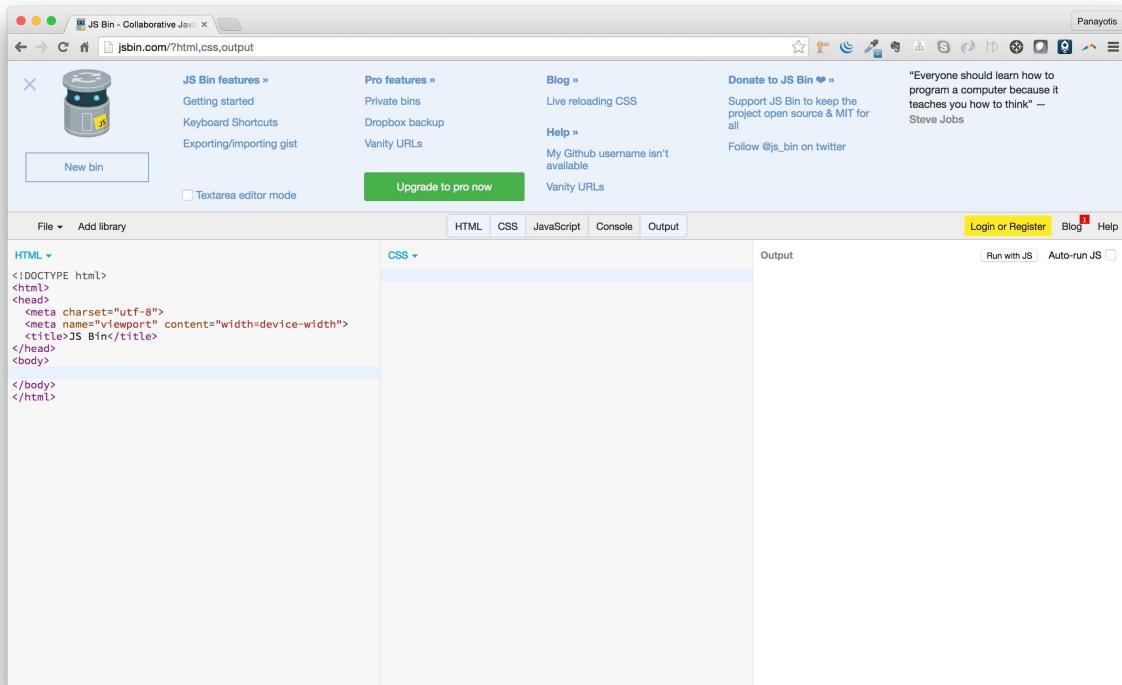
Don't worry about the `undefined` that you see, or if you do not really understand how JavaScript works, or how JavaScript commands should be constructed. We will learn that stuff in the following chapters.

For now, enjoy that you have managed to write your first JavaScript commands.

## JS Bin

Another very popular JavaScript (and not only) testbed is the [JS Bin](http://jsbin.com). JS Bin allows you to write HTML, CSS and JavaScript. Let's try that. Open your browser and visit the page <http://jsbin.com>.

You will see something like this:

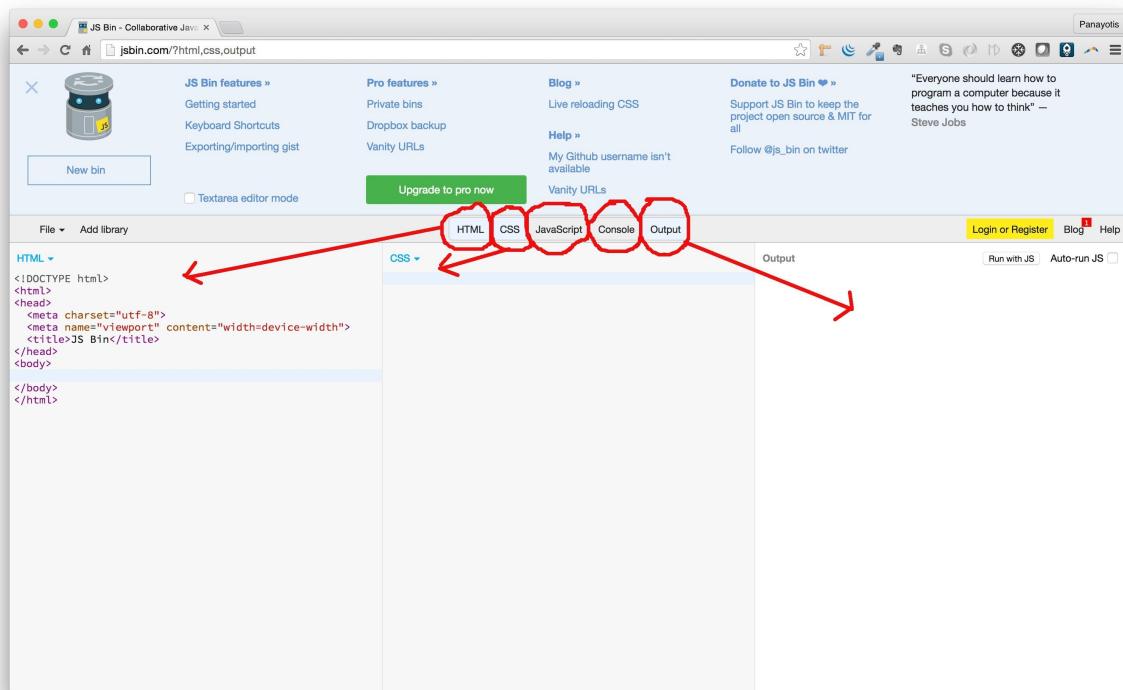


The screenshot shows the JS Bin web application. The top navigation bar includes links for 'File', 'Add library', 'HTML', 'CSS', 'JavaScript', 'Console', 'Output', 'Run with JS', 'Auto-run JS', 'Login or Register', 'Blog', and 'Help'. The main interface consists of three code editors: 'HTML', 'CSS', and 'Output'. The 'HTML' editor contains the following code:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>JS Bin</title>
</head>
<body>
</body>
</html>
```

JS Bin Page

JS Bin offers, as we said earlier, the ability to write HTML, CSS and JavaScript. In the following picture you can see the different tabs that can be switched on and off.



**JS Bin Tabs**

1. HTML. The HTML Tab sits on the left. On the above picture is switched on. It contains a bare minimum of HTML code. This is where you write your HTML code.
2. CSS. The CSS Tab sits second from left. It is where you should be writing your CSS code. It is switched on on the picture above.
3. JavaScript. The JavaScript Tab sits next to CSS. It is where you will be writing your JavaScript code. It is switched off on the picture above. You have to click on the tab in order to switch it on.
4. Console. This tab is not for a particular language. It is for the JavaScript console output messages. It is switched off on the picture above. You have to click on the tab in order to switch it on.
5. Output. This tab is not for a particular language. It is where the combination of HTML, CSS and JavaScript code is rendered by the browser in order to show you the final result.

Watch the following video to understand how we are creating a simple HTML content with basic CSS and how it is output on the right hand side tab.

### Creating an HTML Page on JS Bin

Pretty Cool! Isn't it?

As you can see, while you are editing the content of your HTML page, or changing the CSS rules, the output is automatically being updated.

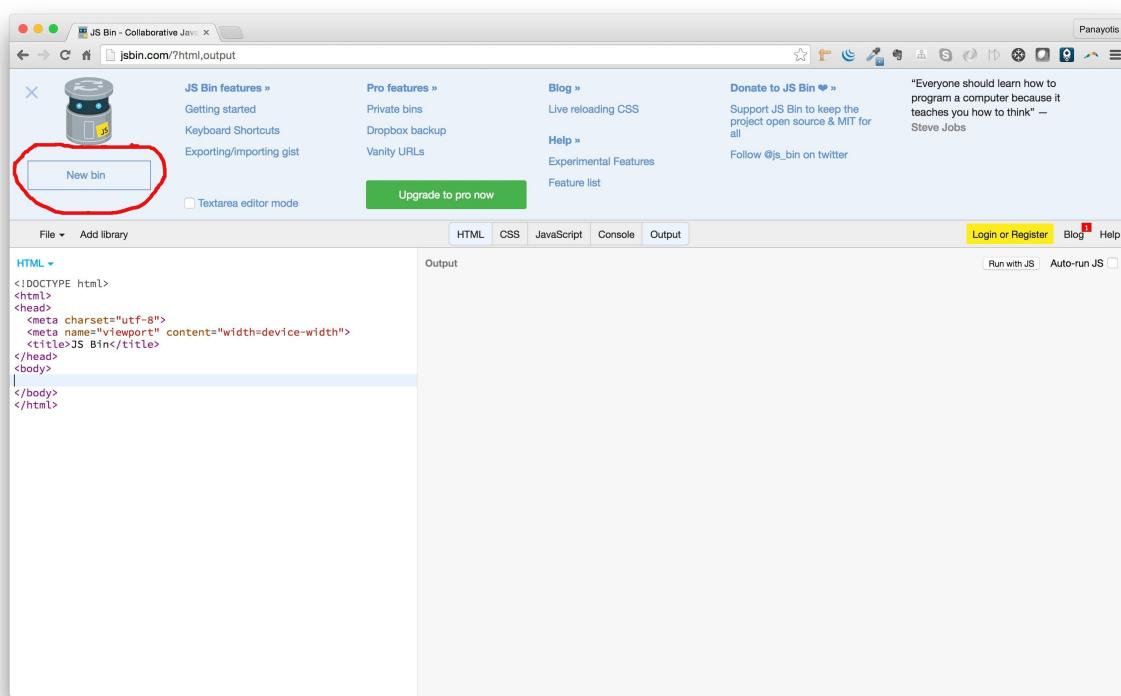
You can even include popular CSS and JavaScript libraries. Watch the following one in which we are including Twitter Bootstrap.

### Creating an HTML Page on JS Bin, including Twitter Bootstrap

During the JavaScript section, we will focus on the JavaScript tab of JS Bin. But it is very good if you learned this tool. It comes very handy when you want to experiment with front-end technologies.

Now, let's write our hello world statements, like we did with Chrome developer tools.

Make sure that you start a new bin.

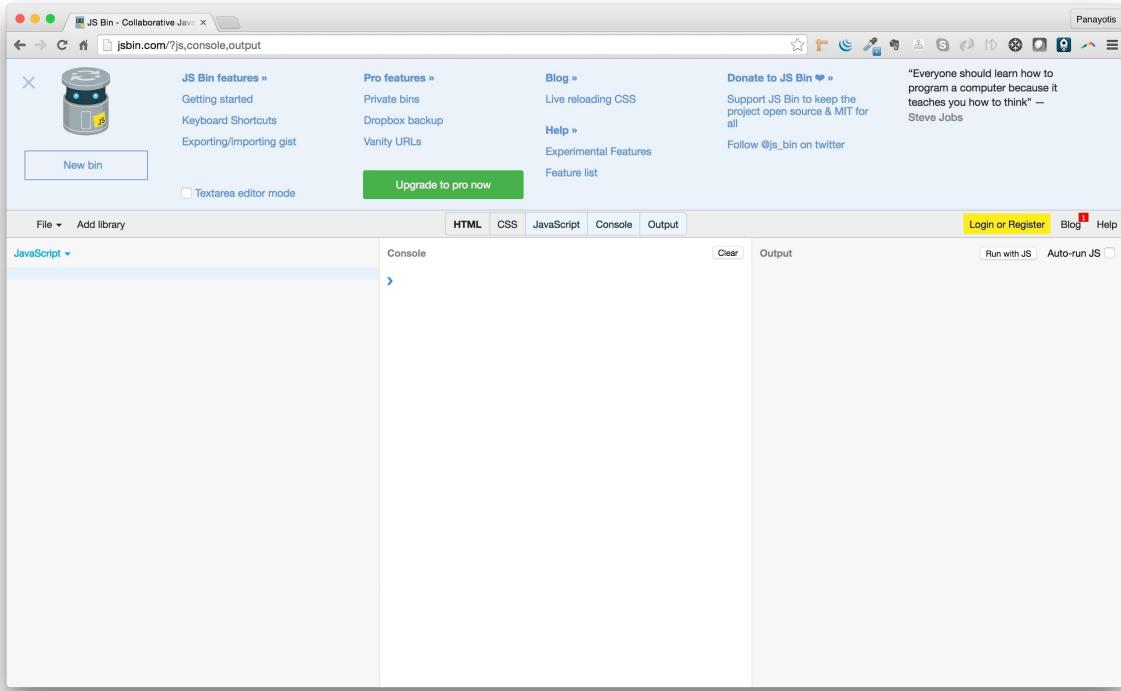


```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>JS Bin</title>
</head>
<body>
</body>
</html>
```

**Click to Create a New Bin**

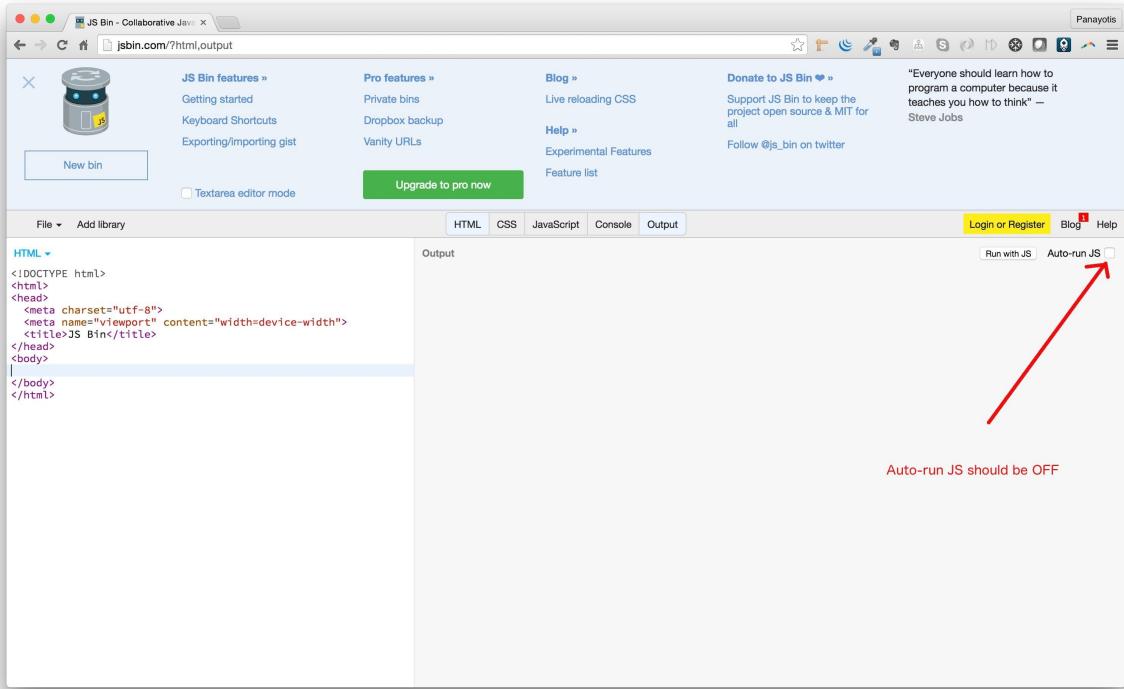
This will make sure that you start from the beginning.

Then switch off HTML and CSS tabs. We will not use them now. Switch on JavaScript tab. This is the tab that we will be using. Also, switch on Console tab. We will need that in order to see the messages that are printed on the console.



### HTML and CSS tabs are Off, JavaScript and Console are On

Also, make sure that Auto-run JS is OFF. We do not want JS Bin to automatically run the JavaScript commands that we type in the JavaScript tab.



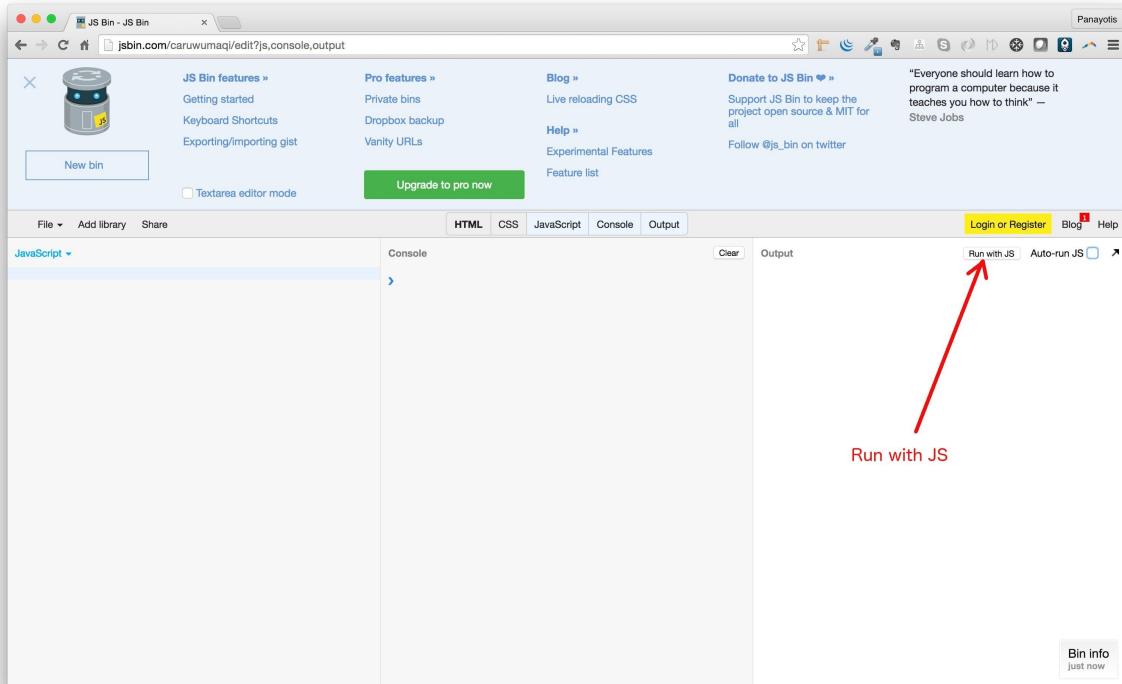
#### Make Sure Auto-run JS is OFF

Now, on the JavaScript tab, write the next JavaScript statement:

```
1 alert("Hello World");
```

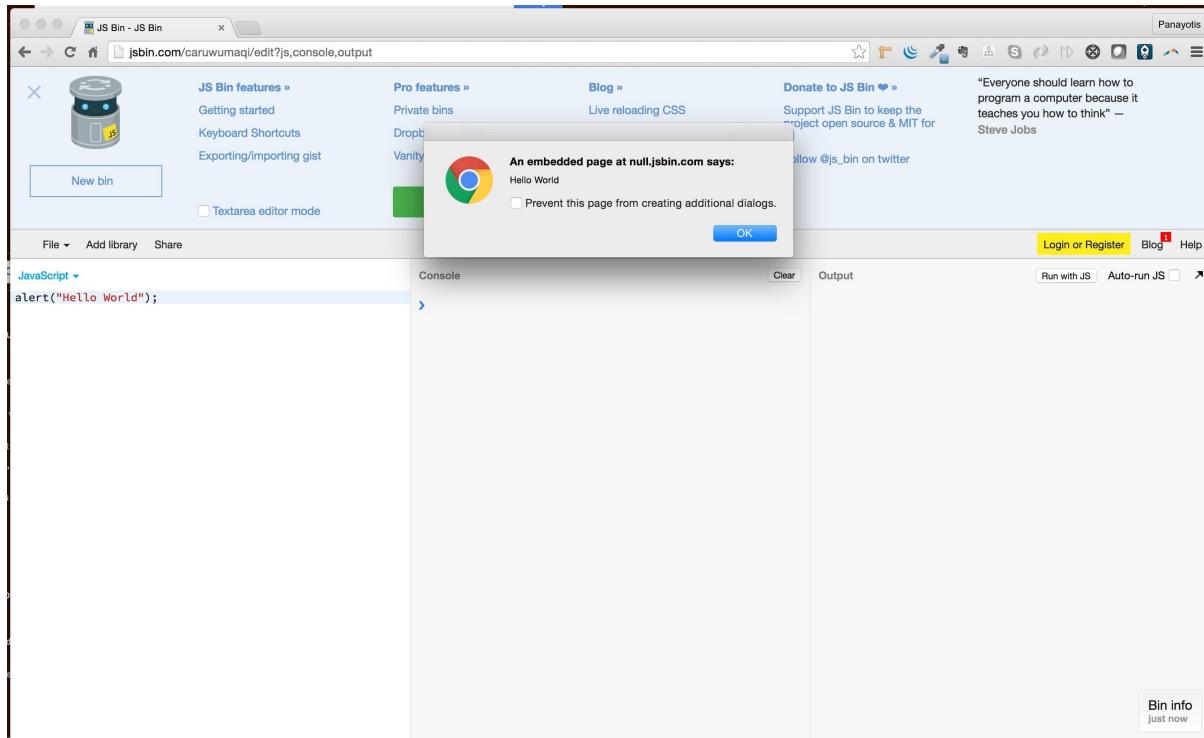
(the above code snippet online)

and then click on the button Run with JS.



### Click on the button Run with JS

This will trigger the execution of the JavaScript statements inside your JavaScript tab. So, you will see something like this:



### Alert Message with Hello World Popping Up

That behaviour is the same like the one we had with the same JavaScript statement on Chrome

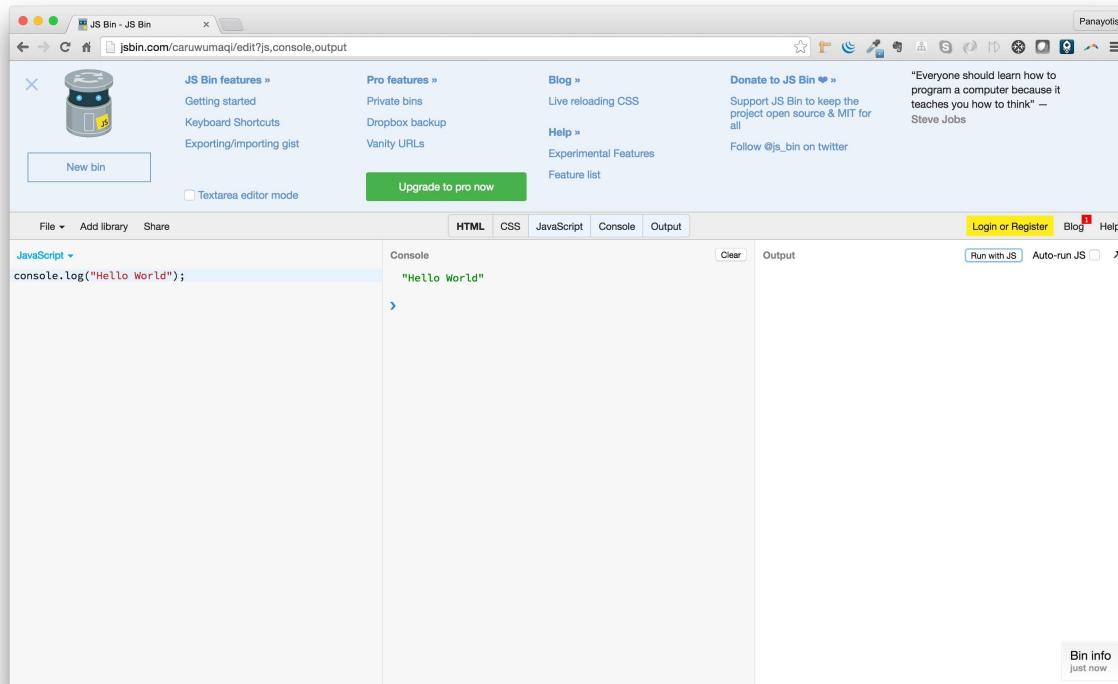
developer tools.

Let's try the other one too. Remove the `alert("Hello World");` statement and write the statement:

```
1 console.log("Hello World");
```

(the above code snippet online)

Then click the Run with JS button. You will see this:



#### Message Inside Console

Cool! We have managed to execute both statements in JS Bin, like we did with the Chrome developer tools.

#### Node.js

We have managed to implement our "Hello World!" little JavaScript programs in four different ways. Two with Chrome developer tools and two with JS Bin. We will now try to use another very popular technology which is called [Node.js](#). Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.

Being a JavaScript runtime engine, it means that it can take JavaScript code and execute it.

**Download from <https://nodejs.org/en/download/> - Install Node.js**

If you click to download Node.js for Mac OS X, you will get a .pkg file. Double click that and follow the instructions to install Node.js.

*Note:* installation will also install the `npm` which is the package manager for Node.js. `npm` is used as a program to organize the JavaScript libraries another program is using.

As soon as you finish the installation, you will see something like that:

```
1 Node.js was installed at
2
3   /usr/local/bin/node
4
5 npm was installed at
6
7   /usr/local/bin/npm
8
9 Make sure that /usr/local/bin is in your $PATH.
```

(the above code snippet online)

In order to make sure that `/usr/local/bin` is in your path, you can do the following on your terminal:

```
1 echo $PATH
```

(the above code snippet online)

Make sure that the folder `/usr/local/bin` is one of the folders listed in the output of this command.

*Note:* the `echo $PATH` command will output a long string. This string is a `:` separated list of items that are folders on your local machine. For example:

```
1 /opt/local/bin:/opt/local/sbin:/Users/panayotismatsinopoulos/.rvm/gems/ruby-1.9.3\
2 -p551/bin:/Users/panayotismatsinopoulos/.rvm/gems/ruby-1.9.3-p551@global/bin:/Use\
3 rs/panayotismatsinopoulos/.rvm/rubies/ruby-1.9.3-p551/bin:/usr/local/bin:/usr/bin\
4 :/bin:/usr/sbin:/sbin:/opt/X11/bin:/usr/local/MacGPG2/bin:/Users/panayotismatsino\
5 poulos/.rvm/bin:/Applications/Postgres.app/Contents/Versions/9.4/bin:/usr/local/b\
6 in:/usr/local/opt/go/libexec/bin:/usr/local/Cellar/elixir/1.2.3/bin
```

(the above code snippet online)

You need to search whether the folder `/usr/local/bin` is one of the items in the list.

If it is not, then you need to edit your `~/.bash_profile` file and add a line like the following:

```
1 export PATH=$PATH:/usr/local/bin
```

(the above code snippet online)

and then close and reopen your terminal.

*Note:* You can always use `nano` to edit the `~/.bash_profile` file.

## Hello World

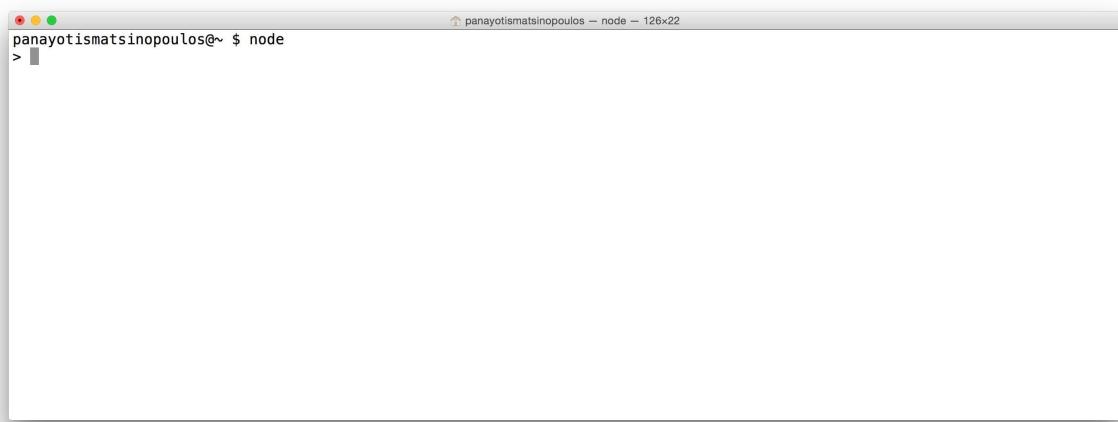
We are ready to use Node.js to print our Hello World! message. Node.js, besides the others, comes with a command line console on its own.

Open the terminal and give the following command:

```
1 node
```

(the above code snippet online)

You will be presented with the symbol > and the cursor blinking next to it. Computer is now waiting for you to give JavaScript commands.



Node console

Let's give the following JavaScript command that will print the message "Hello World!". Type the following and press the Enter key.

```
1 console.log("Hello World!");
```

(the above code snippet online)

As soon as you do that, you will see this:



```
panayotismatsinopoulos@~ $ node
> console.log("Hello World!");
Hello World!
undefined
> █
```

### Hello World on Node.js Console

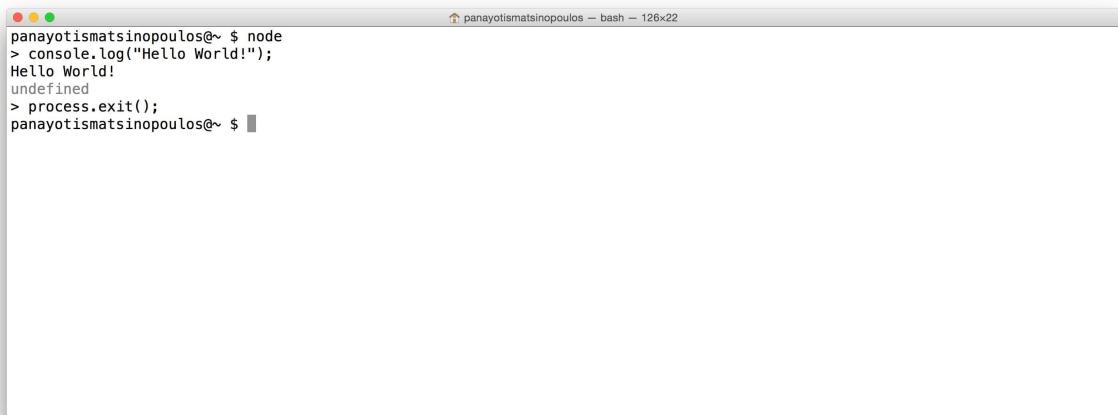
Great! It worked! You asked `Node.js` to print a message for you and it did.

*Note:* Don't worry about the `undefined` string that you see below the "Hello World!" phrase. It is something that we will explain later in the JavaScript section.

Now, before you finish, you need to exit the `Node.js` console. In order to do that, you need to issue the following JavaScript command:

```
1 process.exit();
```

(the above code snippet online)



```
panayotismatsinopoulos@~ $ node
> console.log("Hello World!");
Hello World!
undefined
> process.exit();
panayotismatsinopoulos@~ $ █
```

### Exiting Node.js Console

## Closing

Perfect! We have written our first little JavaScript program. A couple of statements actually. In the next chapters, the real programming adventure starts.

## Tasks and Quizzes

**Before you continue, you may want to know that:** You can sign up to [Tech Career Booster](#) and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

### Quiz

The quiz for this chapter can be found [here](#)

## 2 - Lexical Structure

### Summary

This is where we start learning real programming, using JavaScript.

It's a short but useful chapter because it will teach you the basics of the JavaScript syntax.

The case sensitivity of the language. For example:

Is this one `customer` ([the above code snippet online](#))

same as this one

1 `Customer`

([the above code snippet online](#))

?

Then you will learn how you can write comments inside your code.

After that, you will learn what a literal is.

Also, you will learn how to name variables and functions. The identifiers vs the Reserved Words.

Finally, you will learn how you should be terminating your JavaScript statements.

### Learning Goals

1. Learn about the cases sensitivity of JavaScript.
2. Learn about how to create
  1. line-level (or inline) comments
  2. block-level comments
3. Learn what is a Literal.
4. Learn what is an Identifier.
5. Learn what is a Reserved Word.
6. Learn about how to terminate JavaScript statements.

### Let's Start

Let's start by learning the basics of the lexical structure of JavaScript, like how the variables should be named, how the statements should be terminated e.t.c.

### Case Sensitivity

JavaScript is a case sensitive language. This means that the name `Customer` is not equal to `customer`. Keep that in mind when you name variables and functions or using a reserved word.

## Comments

Comments are pieces of text that we write in our source code file, in order to accompany our code with some descriptive explanation that would make next developer easier to understand what the code does, and basically why it does it.

Comments are not taken into account by the language interpreter. You remember that we used the characters `<!--` to start a comment in HTML and the character sequence `-->` to end it.

Similar feature exists in JavaScript too (like in all programming languages).

There are two types of comments in JavaScript:

### Line-Level Comment

Line-level comments start with the character sequence `//` and expand till the end of the line they appear in. Everything from `//` till the end of the line is ignored by the language interpreter. This is an example:

```
1 var a = 8; // This is an integer
2 var b = 10;
```

(the above code snippet online)

Whereas as `var a = 8;` is a valid JavaScript code that will be interpreted by the JavaScript interpreter, the `// This is an integer` is completely ignored. However, the `var b = 10;` is taken into account as a valid JavaScript code again, because it appears in its own line, even if it follows the line-level comment.

### Block-Level Comments

The block-level comments expand multiple lines. They are applied like the comments in CSS. We start the block with `/*` and we close the block with `*/`

See to the following example:

```
1 /*
2 These are two
3 integers
4 */
5 var a = 8;
6 var b = 10;
```

(the above code snippet online)

Whatever is inside the `/*...*/` is completely ignored.

## Literals

A literal is a data value that appears directly in a program. It explicitly and verbatim declares a value. The following are examples of literals:

```
1 12          // the number twelve
2 1.2         // the number one point two
3 "John Smith" // the text (or string) "John Smith". Literal text can be declared enclosed in double or single quotes
4 'Foo Bar'   // the text (or string) "Foo Bar". Literal text can be declared enclosed in double or single quotes
```

(the above code snippet online)

There are other more complex literals that you will learn about in the following chapters. For now, just make sure that you understand that when we explicitly declare a value, this is a literal.

## Identifiers and Reserved Words

### Identifiers

*Identifiers* are names that we give to variables and functions. You will learn about variables and functions later on. For the time being you just need to know that variables and functions (usually) have names and their names are also called identifiers.

There are some rules that need to be followed when defining an identifier.

1. An identifier needs to start with a letter, an underscore (\_) or the dollar sign \$.
2. Digits are not allowed as the first character of the identifier. This is to make it easy for the JavaScript interpreter to identify numbers.
3. An identifier cannot be a JavaScript reserved word (see next).
4. Otherwise, you can use any character from the Unicode set, but, for compatibility reasons, we usually use the [ASCII \(latin\) characters](#).

The following are valid identifiers:

- customer
- \_state
- \$tag\_element
- Product
- OrderElement
- Invoice1

whereas the 1Invoice is not a valid identifier, because it starts with the number 1.

Besides the rules for the validity of an identifier, there are some other rules that have to do with the style of your JavaScript code. For example, both customer\_number and customerNumber are valid JavaScript identifiers. But which one is preferred in the JavaScript world? It is the JavaScript style guides that define these extra rules. We recommend that you start studying the [Google JavaScript Style Guide](#), little-by-little. This is the style rules that we will be following throughout the course when writing JavaScript programs.

## Reserved Words

The reserved words are words that the JavaScript language has reserved for its internal use and developers cannot use them to name variables or functions. They are also called language keywords.

Here is a list of the words that you should avoid using in order to name your own variables and functions:

JavaScript Reserved Words

break	delete	function	return	typeof
case	do	if	switch	var
catch	else	in	this	void
continue	false	instanceof	throw	while
debugger	finally	new	true	with
default	for	null	try	
class	const	enum	export	extends
import	super			
implements	let	private	public	yield
interface	package	protected	static	
arguments	eval			
abstract	double	goto	native	static
boolean	enum	implements	package	super
byte	export	import	private	synchronized
char	extends	int	protected	throws
class	final	interface	public	transient
const	float	long	short	volatile
arguments	encodeURI	Infinity	Number	RegExp
Array	encodeURIComponent			
		isFinite	Object	String
Boolean	Error	isNaN	parseFloat	SyntaxError
Date	eval	JSON	parseInt	TypeError
decodeURI	EvalError	Math	RangeError	undefined
decodeURIComponent				
	Function	Nan	ReferenceError	URIError

You do not have to learn them by heart. You can always reference this table, but, again, you will learn the words that you should not be using for your own naming, by experience.

## Terminating Statements

Like many programming languages, JavaScript uses the ; to separate one statement from the other. However, these are not required in all cases. You can usually omit the ; symbol:

- at the end of a program
- if the next token is a closing curly brace }

However, we encourage to use ; to terminate statements even in cases in which this might be optional.

## Tasks and Quizzes

**Before you continue, you may want to know that:** You can sign up to [Tech Career Booster](#) and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

### Quiz

The quiz for this chapter can be found [here](#)

## 3 - Types, Values and Variables

### Summary

In previous chapters, we have written our *Hello World* JavaScript program and we have also learned about the basics of the JavaScript syntax. In this chapter, we introduce the way we can present our data.

With programs like this:

The screenshot shows a browser developer tools interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is active, displaying the following code:

```
JavaScript ▾  
var customer;  
customer = "John";  
console.log(customer);  
customer = "Peter";  
console.log(customer);
```

To the right, the Console tab shows the output:

```
Console  
"John"  
"Peter"
```

A blue arrow points from the code block to the corresponding output in the console.

Sample Variable Declaration And Usage

you start declaring and using variables.

And with programs like this:

The screenshot shows a browser developer tools interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is active, displaying the following code:

```
JavaScript ▾  
console.log(2 + 2);  
console.log(3 - 1);  
console.log(10 * 2);  
console.log(6 % 2);  
console.log(6 % 4);  
console.log(6 / 2);  
console.log(6.5 * 3);  
console.log(6.8 / 2.3);
```

To the right, the Console tab shows the output:

```
Console  
4  
2  
20  
0  
2  
3  
19.5  
2.956521739130435
```

A blue arrow points from the code block to the corresponding output in the console.

Demo of Arithmetic Operations In JavaScript

you start using numbers and carry out arithmetic operations.

Besides the numbers, you will learn manipulating strings:

The screenshot shows a browser's developer tools interface with the 'Console' tab selected. At the top, there are tabs for HTML, CSS, JavaScript, Console, and Output. Below the tabs, the 'JavaScript' dropdown is set to 'JavaScript'. In the main area, the following code is written:

```
var firstName = "John";
console.log(firstName + " has length: " + firstName.length);
var lastName = "Papas";
console.log(lastName + " has length: " + lastName.length);
```

When run, the console outputs:

```
"John has length: 4"
"Papas has length: 5"
```

### Working with Strings

You will also be introduced to regular expressions and pattern matching, one of the most powerful tools in any programming language.

The screenshot shows a browser's developer tools interface with the 'Console' tab selected. At the top, there are tabs for HTML, CSS, JavaScript, Console, and Output. Below the tabs, the 'JavaScript' dropdown is set to 'JavaScript'. In the main area, the following code is written:

```
var pattern = /e/;
var matches = pattern.test("George");
console.log(matches);

matches = pattern.test("John");
console.log(matches);
```

When run, the console outputs:

```
true
false
```

### Pattern Matching Example

Next, you will start getting acquainted with the boolean values, `true` and `false`.

Finally, you will get a first idea about the `null` and `undefined`.

## Learning Goals

1. Learn about literals.
2. Learn about variables.
3. Learn what content does a variable have when first declared without assigning any value to it.
4. Learn about the assignment operator.
5. Learn about numbers.
6. Learn about integer literals.
7. Learn about floating-point literals.
8. Learn about the arithmetic operations in JavaScript.
9. Learn about Dates and Times.
10. Learn about Text.
11. Learn about Literal text.
12. Learn about Escape sequences in String Literals and how the character \ can be used to introduce special characters.

13. Learn about concatenating Strings.
14. Learn about some basic String Methods:
  1. `charAt()`
  2. `substring()`
  3. `slice()`
  4. `indexOf()`
  5. `lastIndexOf()`
  6. `split()`
  7. `replace()`
  8. `toUpperCase()`
15. Learn that Strings are Immutable.
16. Learn about pattern matching and regular expressions.
17. Learn about the method `test()` that a `RegExp` object responds to.
18. Learn about the strings that can respond to methods related to regular expressions.
  1. `search()`
  2. `match()`
  3. `split()`
  4. `replace()`
19. Learn how you can do global pattern matching.
20. Learn about the boolean values, `true` and `false`.
21. Learn which literal values are equivalent to `false`.
22. Learn about the comparison operator.
23. Learn about the *falsy* and *truthy* values.
24. Learn about `null` and `undefined`.

## Introduction

Let's start working with data in programs. Data are represented with values taken from a set of values. This set of values represents the type of the datum. Also, the type defines the kind of operations that program can carry out with the particular data of that type.

## Literals and Variables

We use two methods to tell a program which are our data:

1. Literals
2. Variables

The literals, are explicit values like the number `2` or the string `Hello World`. For example, the following program uses two pieces of data: The number `10` and the string `School`:

```
1 console.log(10);
2 console.log("School");
```

(the above code snippet online)

Literals are very useful but they are not very flexible. By representing, in the program, our value as a literal we cannot reuse the same value multiple times.

Variables on the other hand, are named RAM positions. We have already learned that the RAM is a set of positions that we can store our data in. With variables, we give names to these positions and then we are free to reference that position by its name (instead of using its decimal address number).

In other words, variables are boxes that we put values inside and a label outside so that we can refer to that box and distinguish it from another box with other content.

We usually declare variables with the use of the reserved word `var`. For example:

```
1 var customer = "Peter";
```

(the above code snippet online)

On the example above, we declare a variable by giving its name `customer`. We also put some content inside this variable, which is a string value `Peter`.

## Assignment Operator

You can declare a variable without actually giving any content to it. Try, for example, the following program on JS Bin

```
1 var customer;
2 console.log(customer);
```

(the above code snippet online)

You will get this:



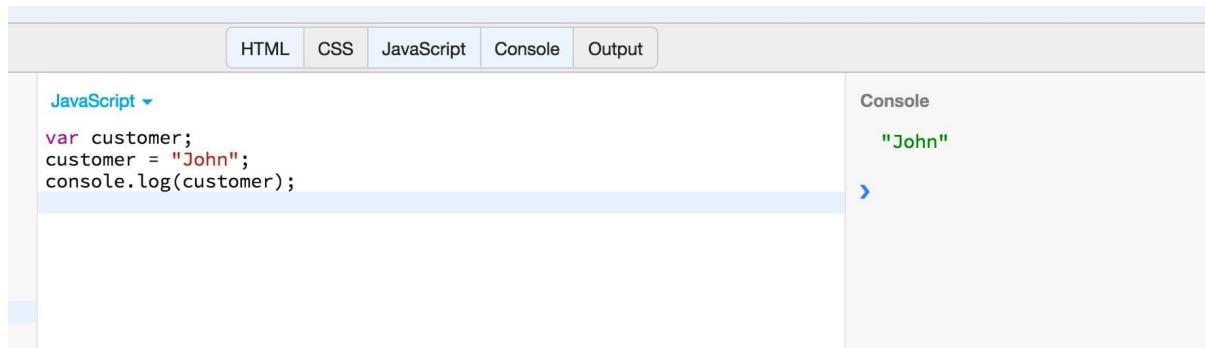
The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the code: `var customer; console.log(customer);`. The Console tab shows the output: `undefined`.

Printing A Variable That Does Not Have Content

As you can see, the output is `undefined`. This is because the variable does not have actual content; When you have already defined a variable and you want to assign some content to it, you can do that with the assignment operator: `=`. Run the following example on JS Bin:

```
1 var customer;
2 customer = "John";
3 console.log(customer);
```

(the above code snippet online)



The screenshot shows a JS Bin interface. The tabs at the top are HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is selected. The code area contains the following script:

```
var customer;
customer = "John";
console.log(customer);
```

The output in the Console tab shows the result of the `console.log` statement:

```
"John"
```

#### Content Of Variable Is No Longer Undefined

As you can see above, the content of the variable `customer` is no longer `undefined`.

But, the assignment operator can be used, not only to give a value to a variable, but also to change its existing value. Try the following program on JS Bin:

```
1 var customer;
2 customer = "John";
3 console.log(customer);
4 customer = "Peter";
5 console.log(customer);
```

(the above code snippet online)



The screenshot shows a JS Bin interface. The tabs at the top are HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is selected. The code area contains the following script:

```
var customer;
customer = "John";
console.log(customer);
customer = "Peter";
console.log(customer);
```

The output in the Console tab shows the results of the two `console.log` statements:

```
"John"
"Peter"
```

#### Using Assignment Operator to Change the Value of a Variable

As you can see above, we can use the assignment operator to change the value of a variable. This is the reason, also, that variables are called like that. Because their content is not fixed.

We have already said that values have a type. The type of a value stored inside a variable determines the type of the variable too.

We continue with studying the various types JavaScript supports.

## Numbers

Values in JavaScript can be of type `Number`. JavaScript does not make any distinction between integers and floating-point numbers. All numbers in JavaScript are floating-point numbers.

### Integer Literals

In JavaScript, an integer is written as a sequence of digits:

- `0` is the zero.
- `10` is the number ten.
- `10201` is the number ten thousand two hundred and one.

These are all literal integers.

### Floating-point Literals

Floating-point literals are usually represented by separating the integral part of the number from the fractional part of the number using a decimal point.

Here are some examples of valid floating-point literals:

- `3.14`
- `2345.17`
- `.333`. In this case, we omit the integral 0. This is optional.
- `6.02e23`. This is equal to  $6.02 \times 10^{23}$ .

## Arithmetic Operations in JavaScript

In JavaScript, like in many other popular programming languages, you can use the following arithmetic operators to carry out arithmetic operations:

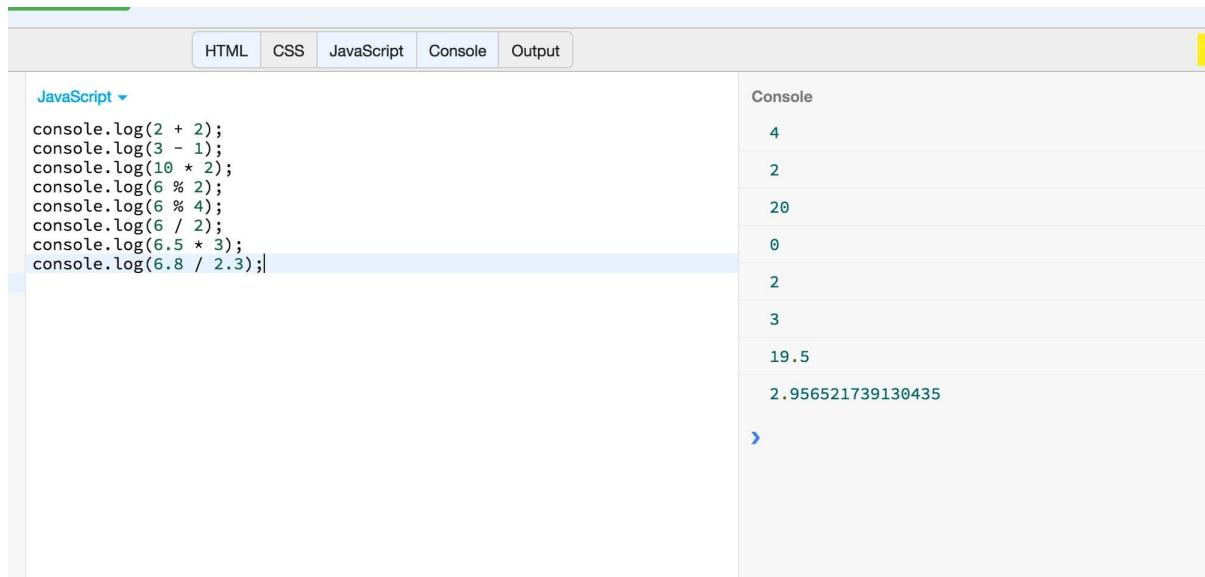
- `+` for the addition
- `-` for the subtraction
- `/` for the division
- `*` for the multiplication
- `%` for module - a.k.a. the remainder after a division

Let's write a JavaScript program to demonstrate the above. Write the following JavaScript code on JS Bin:

```
1 console.log(2 + 2);
2 console.log(3 - 1);
3 console.log(10 * 2);
4 console.log(6 % 2);
5 console.log(6 % 4);
6 console.log(6 / 2);
7 console.log(6.5 * 3);
8 console.log(6.8 / 2.3);
```

(the above code snippet online)

When you run this, you will see the following:



The screenshot shows a browser's developer tools interface with the 'Console' tab selected. The left pane contains the following JavaScript code:

```
JavaScript ▾
console.log(2 + 2);
console.log(3 - 1);
console.log(10 * 2);
console.log(6 % 2);
console.log(6 % 4);
console.log(6 / 2);
console.log(6.5 * 3);
console.log(6.8 / 2.3);
```

The right pane displays the console output:

Console
4
2
20
0
2
3
19.5
2.956521739130435

#### Demo of Arithmetic Operations In JavaScript

We have used the `console.log()` method to print the result of the output of various arithmetic operations examples. Easy and pretty straightforward.

## Dates and Times

JavaScript uses the object `Date()` to deal with the representation of date and time values. The `Date()` is actually a constructor and you can use it to create a date or date and time representation.

Here is an example:

```
1 new Date(2016, 0, 1);
```

(the above code snippet online)

It creates a Date that refers to the 1st day of the 1st month of year 2016. Note that the first month is represented by 0 and not by 1.

We will learn more about the `Date()` functionality later on in this JavaScript section.

## Text

In JavaScript, text is represented with *String* type values and variables.

### Literals

Here is a string literal:

```
1 "Hello World!"
```

(the above code snippet online)

The string literals are declared using either double quotes ("") or single quotes (''). When you use double quotes like in the above example, you can include a single quote as part of the actual value of the string. Look at the following example:

```
1 "What's your name?"
```

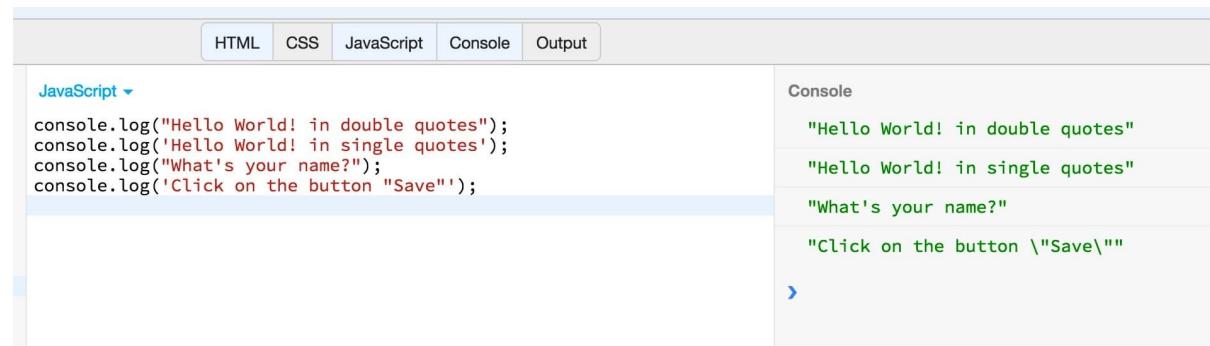
(the above code snippet online)

The opposite goes for single quotes. If you decide to enclose the literal value of a string in single quotes, then you can use a double quote inside the value itself. Try the following example in JS Bin:

```
1 console.log("Hello World! in double quotes");
2 console.log('Hello World! in single quotes');
3 console.log("What's your name?");
4 console.log('Click on the button "Save"');
```

(the above code snippet online)

If you run the above JavaScript code in JS Bin, you will get this:



The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is selected, displaying the following code:

```
console.log("Hello World! in double quotes");
console.log('Hello World! in single quotes');
console.log("What's your name?");
console.log('Click on the button "Save"');
```

The Output tab shows the results of the console.log statements:

Console
"Hello World! in double quotes"
"Hello World! in single quotes"
"What's your name?"
"Click on the button \"Save\""

Using String Literals with Single and Double Quotes

## Escape Sequences in String Literals

Sometimes, we want to include special characters inside a string literal. One very popular special character is the LINE FEED character. The special characters are usually represented with a letter of the latin alphabet prefixed with the character backslash \. For example, the line feed character is represented with the string “\n”. The backslash character is necessary to make sure that the “n” is not interpreted as the latin character n. Having the backslash character as a prefix, we tell JavaScript to translate it to the corresponding special character. The special characters are also called *escape sequences*, and the \ is called *escape character*.

Let's try the following program on JS Bin:

```
1 console.log("This is the world of JavaScript!\nHello JavaScript!");
```

(the above code snippet online)

The screenshot shows a browser-based developer tool interface for testing JavaScript. At the top, there are tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is active, showing the code: `console.log("This is the world of JavaScript!\nHello Javascript!");`. In the Console tab, the output is displayed as two separate lines: "This is the world of JavaScript!" followed by "Hello Javascript!". A small blue arrow icon is visible at the bottom right of the console area.

Line Feed Character Used in String Literal

As you can experience, the \n escape sequence has split the whole string in 2 parts, separating the second from the first by a new line.

There are some other useful escape sequences:

1. The horizontal tab \t.
2. The carriage return \r, which is used in the Windows world.
3. The backslash character \\, which should be used when we want to literally use the backslash character as is, rather than escape the next one.
4. The double quote inside a string enclosed in double quotes. Example: "The book title is: \"Of Mice And Men\"". Useful when you want to always enclose your string literals in double quotes, but you have a double quote that needs to exist in the value itself.
5. The single quote inside a string enclosed in single quotes. Example: 'The book title is: 'Of Mice And Men''. Useful when you want to always enclose your string literals in single quotes, but you have a single quote that needs to exist in the value itself.

Try the above in a JS Bin program.

## Working with Strings

The most basic action one can do with strings is to concatenate them. In order to concatenate two strings, you need to use the + operator, like you do when you want to add two numbers. Try the following on JS Bin:

```
1 var firstName = "John";
2 var lastName = "Papas";
3 console.log(firstName + lastName);
```

(the above code snippet online)

If you run the above program, you will get the following:

The screenshot shows a JS Bin interface. On the left, under the 'JavaScript' tab, is the following code:

```
var firstName = "John";
var lastName = "Papas";
console.log(firstName + lastName);
```

On the right, under the 'Console' tab, the output is:

```
"JohnPapas"
```

#### Concatenating String Using + Operator

As you can see, the operation `firstName + lastName` results in the concatenation of the two strings. So, maybe, that program could have been even better if it were:

```
1 var firstName = "John";
2 var lastName = "Papas";
3 console.log(firstName + " " + lastName);
```

(the above code snippet online)

In that case, if you ran it on JS Bin, you would get:

The screenshot shows a JS Bin interface. On the left, under the 'JavaScript' tab, is the following code:

```
var firstName = "John";
var lastName = "Papas";
console.log(firstName + " " + lastName);
```

On the right, under the 'Console' tab, the output is:

```
"John Papas"
```

#### Concatenate With a Blank In Between

A useful property of the string is the `length` property, which returns, what else?, the length of the string. Try the following program on JS Bin:

```

1 var firstName = "John";
2 console.log(firstName + " has length: " + firstName.length);
3 var lastName = "Papas";
4 console.log(lastName + " has length: " + lastName.length);

```

(the above code snippet online)

If you run that you will see the following:

The screenshot shows a browser-based developer tool interface. At the top, there are tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is selected, showing the following code:

```

var firstName = "John";
console.log(firstName + " has length: " + firstName.length);
var lastName = "Papas";
console.log(lastName + " has length: " + lastName.length);

```

In the adjacent Console tab, the output is displayed in green text:

```

"John has length: 4"
"Papas has length: 5"

```

### Strings And Their Length Property

Pretty easy!

## String Methods

JavaScript offers a list of string methods that one can invoke on a string and get some useful returned value. Try the following JavaScript program on JS Bin:

```

1 var s = "hello, world";
2
3 result = s.charAt(0);
4 console.log(result); // prints "h"
5
6 result = s.charAt(s.length-1);
7 console.log(result); // prints "d"
8
9 result = s.substring(1, 4);
10 console.log(result); // prints "ell"
11
12 result = s.slice(1, 4);
13 console.log(result); // prints "ell"
14
15 result = s.slice(-3);
16 console.log(result); // prints "rld"
17
18 result = s.indexOf("l");
19 console.log(result); // prints 2
20

```

```

21 result = s.lastIndexOf("l");
22 console.log(result); // prints 10
23
24 result = s.indexOf("l", 3);
25 console.log(result); // prints 3
26
27 result = s.split(", ");
28 console.log(result); // prints ["hello", "world"]
29
30 result = s.replace("h", "H");
31 console.log(result); // prints "Hello, world"
32
33 result = s.toUpperCase();
34 console.log(result); // prints "HELLO, WORLD"

```

(the above code snippet online)

If you run this program, you will see this:

HTML	CSS	JavaScript	Console	Output
<b>JavaScript</b> ▾				
<pre> var s = "hello, world";  result = s.charAt(0); console.log(result); // prints "h"  result = s.charAt(s.length-1); console.log(result); // prints "d"  result = s.substring(1,4); console.log(result); // prints "ell"  result = s.slice(1,4); console.log(result); // prints "ell"  result = s.slice(-3); console.log(result); // prints "rld"  result = s.indexOf("l"); console.log(result); // prints 2  result = s.lastIndexOf("l"); console.log(result); // prints 10  result = s.indexOf("l", 3); console.log(result); // prints 3  result = s.split(", "); console.log(result); // prints ["hello", "world"]  result = s.replace("h", "H"); console.log(result); // prints "Hello, world"  result = s.toUpperCase(); console.log(result); // prints "HELLO, WORLD" </pre>				<b>Console</b> <pre> "h" "d" "ell" "ell" "rld" 2 10 3 ["hello", "world"] "Hello, world" "HELLO, WORLD" </pre>
<span style="font-size: 2em;">›</span>				

### Examples of String Methods

I believe that the majority of the above are self-explanatory, if one also reads the comments next to each result.

Very briefly:

1. `s.charAt(0)` returns the character at position 0. Note that string characters are positioned starting from 0, not from 1. Hence a string with 5 characters has the 1st character at position 0, the 2nd character at position 1, the 3rd at position 2, the 4th at position 3 and the 5th at position 4.
2. `s.substring(1, 4)` returns the substring starting at position 1 up to but not including position 4, hence up to and including position 3.
3. `s.slice(1, 4)` is similar to `substring()`. But if the last argument is missing, goes till the end of the string. Also, if the first argument is negative, it works from the end character towards the first.
4. `s.indexOf("1")` returns the position of a character "1" inside a string.
5. `s.indexOf("1", 3)` returns the position of the character "1" inside `s`, starting search at position 3.
6. `s.lastIndexOf("1")` returns the last position of a character "1" inside a string.
7. `s.split(", ")`; returns an array with the parts of the original string, splitting the original string in parts according to the character sequence ", ".
8. `s.replace("h", "H")` returns a new string in which the "h" have been replaced with "H".
9. `s.toUpperCase()` returns the original string with all characters converted to their uppercase equivalent.

But, as we said, this is not the full list of string methods. Google for `JavaScript string methods` reference to get the full list of string methods offered by JavaScript.

## Strings are Immutable

You need to understand that strings are **immutable**. This means that when you call a method on a string, this does not alter the string itself, but only returns a new value corresponding to the method called.

In order to understand that, execute the following JavaScript program in JS Bin:

```
1 var s = "hello, world";
2
3 result = s.toUpperCase();
4 console.log(result); // prints "HELLO, WORLD"
5 console.log(s); // prints "hello, world"
```

(the above code snippet online)

If you run this program, you will get this:

```
JavaScript
var s = "hello, world";
result = s.toUpperCase();
console.log(result); // prints "HELLO, WORLD"
console.log(s); // prints "hello, world"
```

Console

```
"HELLO, WORLD"
"hello, world"
```

### Strings Are Immutable

As you can see above, the method call `s.toUpperCase()` didn't alter the `s` string itself. The `s` string still holds the value `hello, world`. If you wanted to change the value of the string itself, you would have to assign the result to the string itself. Run the following program on JS Bin:

```
1 var s = "hello, world";
2
3 s = s.toUpperCase();
4 console.log(s); // prints "HELLO, WORLD"
```

(the above code snippet online)

If you run it, you will get this:

```
JavaScript
var s = "hello, world";
s = s.toUpperCase();
console.log(s); // prints "HELLO, WORLD"
```

Console

```
"HELLO, WORLD"
```

### Changing the Value of a String Variable

As you can see above, in order to change the value of a string variable, we have to assign the result of the method back to the variable itself.

## Pattern Matching

While working with text, a developer often comes to the situation in which he has to match a string against a text pattern. For example, assuming that we have the following list of strings:

1. John
2. Mary
3. George
4. Danny
5. Alex

## 6. Paul

one might want to get the strings that belong to this list and that they have the letter e inside. the letter e inside, or better put, includes character e, is a text pattern, because it does not only refer to text e but also refers to anything that includes e, essentially referring to a huge list of texts. The strings, of the above list, that match this pattern are strings George and Alex.

The text patterns are being declared using what we call *regular expressions*. Generally, regular expressions are declared likewise in many programming languages, with small differences.

In JavaScript, one can declare a regular expression, by enclosing the text pattern, which needs to follow the regular expression syntax, inside forward slashes /. Hence, the text pattern includes character e, is literally defined as /e/.

When one assigns a text pattern, a regular expression, to a variable, then that variable becomes a regular expression object (RegExp in JavaScript world), that responds to the method test. The method test on a regular expression object returns true or false according to whether its input string matches or does not match the regular expression.

Let's see an example. Write the following JavaScript program on JS Bin:

```
1 var pattern = /e/;
2 var matches = pattern.test("George");
3 console.log(matches);
4
5 matches = pattern.test("John");
6 console.log(matches);
```

(the above code snippet online)

If you run the above, you will see this:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the provided code. The Console tab shows the output: 'true' for 'George' and 'false' for 'John'.

Console
true
false

Pattern Matching Example

Which means that the string "George" matches the pattern, regular expression, /e/, whereas the string "John" does not.

*Note:* The true is a boolean literal value. Same goes for false. Former represents a TRUE fact, whereas the latter represents a FALSE fact. We will talk about boolean values later on in this chapter.

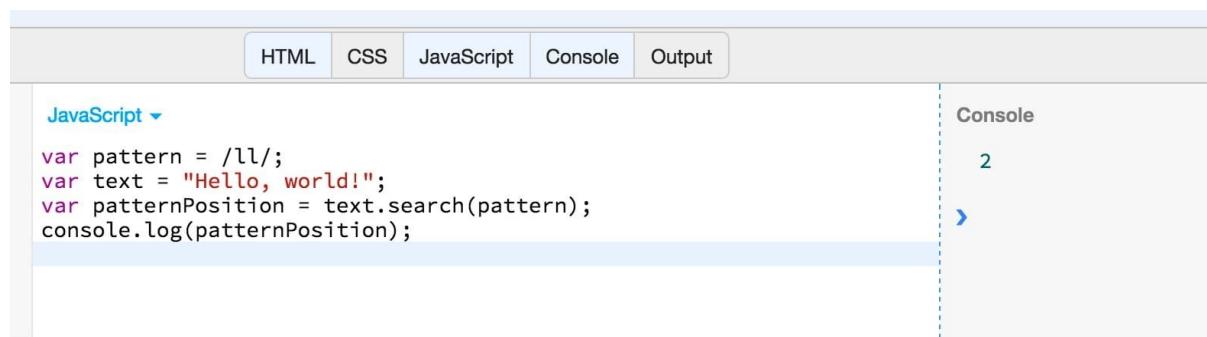
Besides the `test()` method that the regular expression responds to, strings themselves have some methods that can take as input regular expressions.

For example, the `search()` method can take as input a regular expression and return the first position index that matches the text pattern. Write the following program on JS Bin:

```
1 var pattern = /ll/;
2 var text = "Hello, world!";
3 var patternPosition = text.search(pattern);
4 console.log(patternPosition);
```

(the above code snippet online)

If you run the above program, you will get this:



The screenshot shows a JS Bin interface. At the top, there are tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is selected and contains the following code:

```
var pattern = /ll/;
var text = "Hello, world!";
var patternPosition = text.search(pattern);
console.log(patternPosition);
```

To the right, under the 'Console' tab, the output is displayed as a single line: 2.

Getting The Position Of Text Pattern Inside A String

This returns 2 which is the starting position of the pattern `/ll/` inside the string `"Hello, world!"`.

Another string function that works, also, with regular expressions is the `match()` method. The `match()` method will return the matching sub-strings of the given string. Let's write the following program on JS Bin:

```
1 var pattern = /[1-9]+/;
2 var text = "This is 1 a 23 phrase with 32 words and 4423 numbers in between. 3829\
3 ";
4 var result = text.match(pattern);
5 console.log(result);
```

(the above code snippet online)

The regular expression `/[1-9]+/` means match any digit in the range 1 up to 9, that appears one or more times consecutively.

If you run the above program, you will get this:

The screenshot shows a browser developer tools console with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is active, displaying the following code:

```
JavaScript ▾
var pattern = /[1-9]+/;
var text = "This is 1 a 23 phrase with 32 words and 4423 numbers in between. 3829";
var result = text.match(pattern);
console.log(result);
```

The Console tab shows the output:

```
Console
["1"]
>
```

### Using the match Method on a String

As you can see, the `match()` method returns an array containing all the parts of the string that match the regular expression. The point here is that, generally, a pattern matching stops at the first match. That's why you only see: `["1"]`. If you want the pattern matching to find all the matches, then you need to suffix the regular expression literal with the letter `g`, after the closing `/`. Let's see the following program:

```
1 var pattern = /[1-9]+/g;
2 var text = "This is 1 a 23 phrase with 32 words and 4423 numbers in between. 3829\
3 ";
4 var result = text.match(pattern);
5 console.log(result);
```

(the above code snippet online)

If you run the above program, you will get this:

The screenshot shows a browser developer tools console with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is active, displaying the same code as the previous example, but with the `g` suffix added to the regular expression pattern.

The Console tab shows the output:

```
Console
["1", "23", "32", "4423", "3829"]
>
```

### Global Pattern Matching

The `g` letter suffix (`g` for global) tells regular expression to work on the entire input string.

Two other popular methods that work with regular expressions are `split()` and `replace()`. The first splits a string into parts and uses as delimiter the regular expression given as argument. The latter, replaces any text part that matches a regular expression. Let's see an example of `replace()`:

```
1 var pattern = /[1-9]+/g;
2 var text = "This is 1 a 23 phrase with 32 words and 4423 numbers in between. 3829\
3 ";
4 var result = text.replace(pattern, "--");
5 console.log(result);
```

(the above code snippet online)

If you run the above program, you will get the following:

The screenshot shows a browser's developer tools console. The tabs at the top are HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is selected. The code in the console is:

```
var pattern = /[1-9]+/g;
var text = "This is 1 a 23 phrase with 32 words and 4423 numbers in between. 3829";
var result = text.replace(pattern, "--");
console.log(result);
```

The output in the Console tab is:

```
"This is -- a -- phrase with -- words and -- numbers in between. --"
```

### Global Pattern Matching and replace method

As you can see above, the `replace()` method replaced the numbers with a double dash.

You can use this technique to remove the numbers from the string:

```
1 var pattern = /[1-9]+/g;
2 var text = "This is 1 a 23 phrase with 32 words and 4423 numbers in between. 3829\
3 ";
4 var result = text.replace(pattern, "");
5 console.log(result);
```

(the above code snippet online)

Does it work?

We will study regular expressions and how you can define syntactically correct regular expressions in a later chapter.

## Boolean Values

Boolean values represent the truth of falsehood, yes or no, on or off. There are two values that are reserved words and can be used as boolean literal values:

1. `true`, which represents the truth.
2. `false`, which represents the falsehood.

Note that besides these two values, all other values can be converted to a boolean value. JavaScript has the following rules:

1. The following values are equivalent to `false`:
  - `undefined`
  - `null`
  - `0`
  - `-0`
  - `NaN`
  - `""` (which is the blank string)
2. All other literal values are equivalent to `true`
3. Also, a variable is equivalent to `false` if its value is equivalent to `false`.
4. And a variable is equivalent to `true` if its value is equivalent to `true`.

If you want to check whether a value or variable is `false`, you have to compare it against the `false` value. The comparison operator is `==` and the result of the comparison is again a boolean value (either `false` or `true`).

*Note:* We will talk about the comparison operators in a later chapter.

Similarly for the check against `true`. If you want to check whether a value or variable is `true`, you have to compare it against the `true` value.

Let's try the following program on JS Bin:

```
1 var nullValue = null;
2 console.log(nullValue == true);
3 console.log(nullValue == false);
4
5 var zeroValue = 0;
6 console.log(zeroValue == true);
7 console.log(zeroValue == false);
8
9 var blankString = "";
10 console.log(blankString == true);
11 console.log(blankString == false);
12
13 var trueValue = true;
14 console.log(trueValue == true);
15 console.log(trueValue == false);
16
17 var falseValue = false;
18 console.log(falseValue == true);
19 console.log(falseValue == false);
20
21 var aStringValue = "Hello World";
22 console.log(aStringValue == true);
23 console.log(aStringValue == false);
24
25 var anIntegerValue = 1;
26 console.log(anIntegerValue == true);
27 console.log(anIntegerValue == false);
```

(the above code snippet online)

If you run this program, you will get the following output:

HTML	CSS	JavaScript	Console	Output
<b>JavaScript ▾</b>				<b>Console</b>
		<code>var nullValue = null; console.log(nullValue == true); console.log(nullValue == false);</code>		<code>false</code>
		<code>var zeroValue = 0; console.log(zeroValue == true); console.log(zeroValue == false);</code>		<code>false</code>
		<code>var blankString = ""; console.log(blankString == true); console.log(blankString == false);</code>		<code>false</code>
		<code>var trueValue = true; console.log(trueValue == true); console.log(trueValue == false);</code>		<code>true</code>
		<code>var falseValue = false; console.log(falseValue == true); console.log(falseValue == false);</code>		<code>false</code>
		<code>var aStringValue = "Hello World"; console.log(aStringValue == true); console.log(aStringValue == false);</code>		<code>false</code>
		<code>var anIntegerValue = 1; console.log(anIntegerValue == true); console.log(anIntegerValue == false);</code>		<code>true</code>
				<code>false</code>

### Comparing To Boolean Values

As you can see above, this program shows some examples about our statement on which values are considered `true` and which ones are considered `false`.

Note that all the values and variables that behave like `false`, are called *falsy*, whereas all the values and variables that behave like `true`, are called *truthy*.

### null and undefined

The `null` keyword is used to indicate that a variable does not have any value. Same goes for `undefined`.

If you compare `null` to `undefined` with the comparison operator `==`, then you will get `true`.

However, these two have some subtle differences. For example, the `null` is a keyword, whereas the `undefined` is a global variable with value `undefined`. We will learn about those differences later on as we use the language more and more.

Generally, you might consider `undefined` to represent a system-level, unexpected, or error-like absence of value and `null` to represent program-level, normal, or expected absence of value. If you need to assign one of these values to a variable or property or pass one of these values to a function, `null` is almost always the right choice.

## Tasks and Quizzes

Before you continue, you may want to know that: You can sign up to [Tech Career Booster](#) and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

### Tasks

(1) You need to write a small JavaScript program that demonstrates how we can swap the values of two variables.

1. declares a variable `a` with value `5`.
2. declares a variable `b` with value `10`.
3. prints the contents of the two variables using `console.log()` statements. This one prints `5` and then `10`
4. swaps the values of these two variables.
5. prints the contents of the two variables using `console.log()` statements. This one prints `10` and then `5`.

**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

(2) You need to write a small JavaScript program that demonstrates the truth of the mathematical formula:

$$1 \quad (a + b)(a - b) = a^{2} - b^{2}$$

Hints:

1. You can do that by assuming that `a` has the value `5` and `b` has the value `10`.
2. Use as many temporary variables that you want in order to progressively calculate the left part of the equation.
3. Same for right part of the equation.
4. Then print the result of comparing the outputs of the previous 2 steps using the comparison operator. It should print `true`.

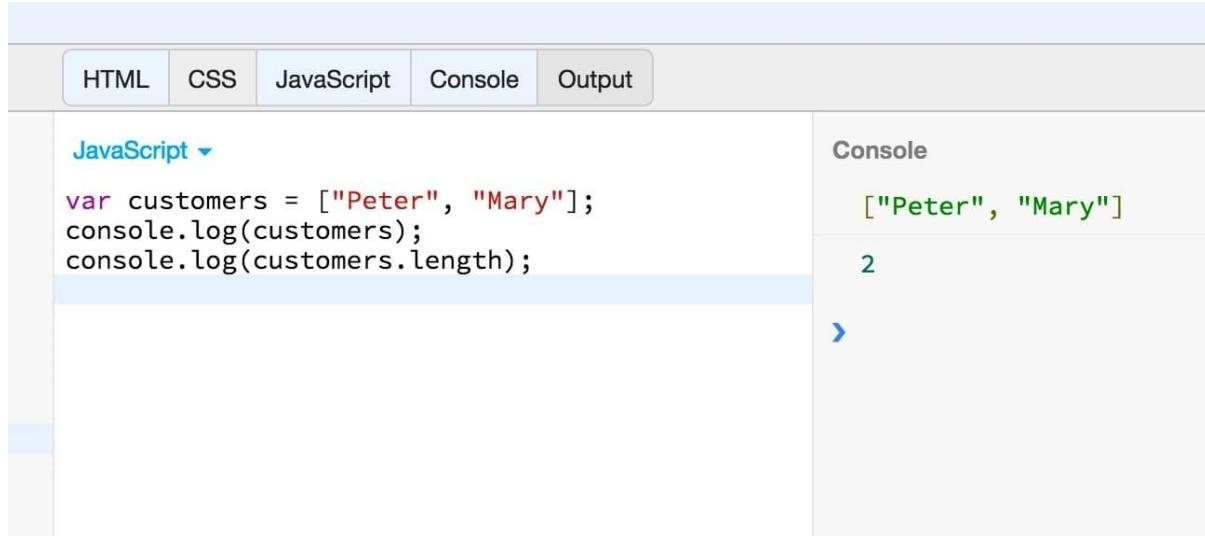
**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

## 4 - Expressions And Operations

### Summary

In this chapter we start understanding what a JavaScript expression is. We learn about simple and complex expressions. Also, we start learning about the various operators. Unary, binary and ternary operators. How they act on their operands and how they come back with a result of calculation.

For example, you will be in position to initialize an Array and calculate its length.



The screenshot shows a browser's developer tools console interface. At the top, there are tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is selected, indicated by a dropdown arrow icon. Below the tabs, the code is written in the JavaScript pane:

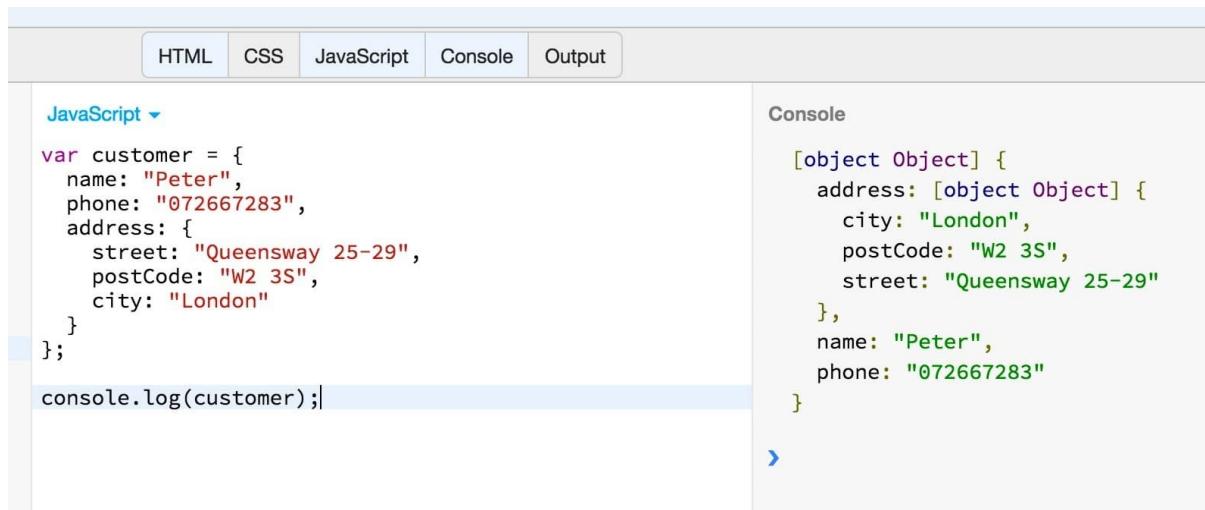
```
var customers = ["Peter", "Mary"];
console.log(customers);
console.log(customers.length);
```

The output pane to the right shows the results of the console.log statements:

```
["Peter", "Mary"]
2
```

Initializing An Array With Two Elements

Or you will learn how to initialize an object with several properties:



The screenshot shows a browser's developer tools console interface. At the top, there are tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is selected, indicated by a dropdown arrow icon. Below the tabs, the code is written in the JavaScript pane:

```
var customer = {
  name: "Peter",
  phone: "072667283",
  address: {
    street: "Queensway 25-29",
    postCode: "W2 3S",
    city: "London"
  }
};

console.log(customer);
```

The output pane to the right shows the results of the console.log statement:

```
[object Object] {
  address: [object Object] {
    city: "London",
    postCode: "W2 3S",
    street: "Queensway 25-29"
  },
  name: "Peter",
  phone: "072667283"
}
```

Initializing Object With Several Properties

Moreover, you will get a first encounter on function definitions and invocation.

The screenshot shows a browser's developer tools interface with the 'Console' tab selected. In the JavaScript pane, the following code is written:

```
var square = function(x) { return x * x; };
console.log(square);
console.log(square(5));
```

In the Console pane, the output is:

```
function (x) { return x * x; }
25
>
```

#### Function Definition And Invocation

You will have a first encounter with operators, like pre- and post-increment and decrement operators.

The screenshot shows a browser's developer tools interface with the 'Console' tab selected. In the JavaScript pane, the following code is written:

```
var number = 1;
number++;
console.log(number);

number--;
console.log(number);
```

In the Console pane, the output is:

```
2
1
>
```

#### Increment and Decrement Example

You will understand about the equality and strict equality operators, so that the following would make sense to you:

The screenshot shows a browser's developer tools interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is active, displaying the following code:

```
var a = "1";
var b = 1;

console.log(a == b);
console.log(a === b);
```

The output in the Console tab shows two rows of results:

Console
true
false

A blue arrow points from the last line of the code to the 'false' result in the console.

### Comparing with Equality and Strict Equality

You will also learn how to compare operands using greater than, greater than or equal, less than or less than or equal operators. Also, you will learn to check whether a property is part of the properties of an object.

The screenshot shows a browser's developer tools interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is active, displaying the following code:

```
var customer = {firstName: "Peter", lastName: "Woo"};
console.log("firstName" in customer);
console.log("lastName" in customer);
console.log("address" in customer);
```

The output in the Console tab shows three rows of results:

Console
true
true
false

A blue arrow points from the last line of the code to the 'false' result in the console.

### Testing Property Existence

You will also learn to use the boolean operators:

The screenshot shows a browser's developer tools interface with the 'Console' tab selected. In the JavaScript panel, the following code is written:

```
var a = true;
var b = false;
console.log(a && b);
console.log(a || b);
console.log(!a);
console.log(!b);
```

In the Console panel, the output is:

Output
false
true
false
true

A blue arrow points from the bottom right of the console area towards the right edge of the screen.

Logical Operators Example

Finally, you will learn how to assign a value of a variable and applying an operator at the same time.

## Learning Goals

1. Learn about JavaScript expressions.
2. Learn about primary expressions.
3. Learn about array initializers.
  1. How to initialize an empty array.
  2. How to initialize an array with items.
4. Learn about initializing an object.
  1. How to initialize an object with properties.
  2. How to initialize an object without properties. Add them later.
5. Learn how you can define a function.
6. Learn about the Operators.
  1. Unary operators.
  2. Binary operators.
  3. Operators precedence.
7. Learn more about the pre and post increment and decrement operators.
8. Learn about the Relational Expressions
  1. equality and inequality.
  2. strict and loose equality.
  3. less than operator.
  4. less than or equal operator.
  5. greater than operator.
  6. greater than or equal operator.
  7. Learn about the `in` operator.
9. Learn about logical expressions.
10. Learn about assignment that combines an operator too.

## Introduction

An expression is a phrase of JavaScript that a JavaScript interpreter can evaluate to produce a value. For example, a constant embedded literally in your program, e.g "Hello World", is a very simple kind of expression. A variable name is also a simple expression that evaluates to whatever value has been assigned to that variable.

Complex expressions are built from simpler expressions. The most common way to build a complex expression out of simpler expressions is with an **operator**. An operator combines the values of its operands (usually two of them) in some way and evaluates to a new value.

For example, the multiplication operator \*. The expression

```
1 x * y
```

(the above code snippet online)

evaluates to the product of the values of the expressions x and y.

For simplicity, we sometimes say that an operator *returns* a value rather than *evaluates to* a value.

Let's see various expressions and operators that JavaScript supports.

## Primary Expressions

The primary expressions are the simplest expressions, the ones that do not include any simpler expression. These are either constants/literals or certain language reserved words and the variable names/references.

All of the following are examples of primary expressions:

```
1 1.23
2 "hello"
3 /pattern/
4 true
5 false
6 null
7 this
8 customer // where "customer" is a variable name/reference.
```

(the above code snippet online)

## Array Initializers

Array initializers are expressions that return a value of an array. An array is a construct that holds a sequence of items.

The following piece of code initializes the variable customers to an empty array:

```
1 var customers = [];
```

(the above code snippet online)

As you can see, the empty array is represented with the square brackets.

Write the following program on JS Bin:

```
1 var customers = [];
2 console.log(customers);
3 console.log(customers.length);
```

(the above code snippet online)

and then run it. You will get this:

The screenshot shows a JS Bin interface. At the top, there are tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is selected, showing the following code:

```
var customers = [];
console.log(customers);
console.log(customers.length);
```

In the Console tab, the output is:

```
[]  
0
```

Initializing An Empty Array

The array objects have the property `length`, like the strings do. And you can see that its value is `0` for arrays initialized with the expression `[]`.

Let's try something more complex:

```
1 var customers = ["Peter", "Mary"];
2 console.log(customers);
3 console.log(customers.length);
```

(the above code snippet online)

If you try this on JS Bin, you will get:

The screenshot shows a browser-based developer tool interface for running JavaScript code. At the top, there are tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is selected, indicated by a blue background. Below the tabs, the code area contains the following JavaScript code:

```
JavaScript ▾
var customers = ["Peter", "Mary"];
console.log(customers);
console.log(customers.length);
```

The output area, titled "Console", shows the results of the code execution:

```
["Peter", "Mary"]
2
>
```

### Initializing An Array With Two Elements

You can even have more complex expressions initializing the items of an Array:

```
1 var powersOf2 = [2, 2*2, 2*2*2, 2*2*2*2, 2*2*2*2*2];
2 console.log(powersOf2);
3 console.log(powersOf2.length);
```

(the above code snippet online)

If you run the above program on JS Bin, you will get this:

The screenshot shows the same browser-based developer tool interface as before. The JavaScript tab is selected. The code area contains the following JavaScript code:

```
JavaScript ▾
var powersOf2 = [2, 2*2, 2*2*2, 2*2*2*2, 2*2*2*2*2];
console.log(powersOf2);
console.log(powersOf2.length);
|
```

The output area, titled "Console", shows the results of the code execution:

```
[2, 4, 8, 16, 32]
5
>
```

### Initializing Array Elements With Complex Expressions

As you can see above, we can initialize the elements of an array, using more complex expressions. On the example above, we initialize the array with the first 5 powers of 2, using the multiplication operator to build the correct expressions per element.

## Object Initializers

Object initializers are almost similar to array initializers. But we use curly braces instead of square brackets. Also, the Object differs from the array, that it has properties with values, rather than just values. Let's see an example:

```
1 var customer = {name: "Peter", phone: "072667283"};
2 console.log(customer);
```

(the above code snippet online)

If you save the above code and run it on JS Bin, then you will get this:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the code: `var customer = {name: "Peter", phone: "072667283"}; console.log(customer);`. The Console tab shows the output: `[object Object] { name: "Peter", phone: "072667283" }`.

### Example Of Object Initialization

The object that we create on the above example, has two properties, name and phone. name has the value "Peter" and phone has the value "072667283".

Again, you can initialize an object with no properties at all:

```
1 var customer = {};
```

(the above code snippet online)

And you can assign properties and values later on, like this:

```
1 customer.name = "Peter";
2 customer.phone = "072667283";
```

(the above code snippet online)

Try the following program on JS Bin:

```
1 var customer = {};
2 console.log(customer);
3
4 customer.name = "Peter";
5 customer.phone = "072667283";
6 console.log(customer);
```

(the above code snippet online)

When you run it, you will get this:

The screenshot shows a browser's developer tools interface with a tab bar at the top labeled 'HTML', 'CSS', 'JavaScript', 'Console', and 'Output'. The 'JavaScript' tab is active, indicated by a dropdown arrow icon. Below the tabs, there are two main sections: 'JavaScript' on the left and 'Console' on the right. In the 'JavaScript' section, the following code is displayed:

```
var customer = {};
console.log(customer);

customer.name = "Peter";
customer.phone = "072667283";
console.log(customer);
```

In the 'Console' section, the output is shown in three lines:

```
[object Object] { ... }
[object Object] {
  name: "Peter",
  phone: "072667283"
}
```

### Adding Properties To An Object

Exactly like for the array, the values that we initialize the properties of an object with, do not have to be primary expressions. They can be complex expressions too. For example, the following initializes the `customer` object, the property `address` of which is initialized with another object literal.

```
1 var customer = {
2   name: "Peter",
3   phone: "072667283",
4   address: {
5     street: "Queensway 25-29",
6     postCode: "W2 3S",
7     city: "London"
8   }
9 };
10
11 console.log(customer);
```

(the above code snippet online)

If you run the above program, you will get this:

The screenshot shows a browser-based developer tool interface. At the top, there are tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is active, showing the following code:

```
JavaScript ▾
var customer = {
  name: "Peter",
  phone: "072667283",
  address: {
    street: "Queensway 25-29",
    postCode: "W2 3S",
    city: "London"
  }
};

console.log(customer);
```

In the adjacent Console tab, the output is displayed as:

```
[object Object] {
  address: [object Object] {
    city: "London",
    postCode: "W2 3S",
    street: "Queensway 25-29"
  },
  name: "Peter",
  phone: "072667283"
}
```

Initializing Object Property with Object Literal Value

## Function Definition Expressions

We continue with expressions that define the content of a function. A function definition expression typically consists of the keyword `function` followed by a comma-separated list of zero or more identifiers (the parameter names) in parentheses and a block of JavaScript code (the function body) in curly braces. For example:

```
1 var square = function(x) { return x * x; };
```

(the above code snippet online)

defines the function that returns the square of the value given as parameter. Try the following program on JS Bin:

```
1 var square = function(x) { return x * x; };
2 console.log(square);
3 console.log(square(5));
```

(the above code snippet online)

If you run the above program, you will get this:

The screenshot shows a browser's developer tools interface with the 'Console' tab selected. In the JavaScript pane, the following code is written:

```
var square = function(x) { return x * x; };
console.log(square);
console.log(square(5));
```

In the Console pane, the output is:

```
function (x) { return x * x; }
25
```

### Function Definition And Call Example

In the above example, we define a function and store its definition to the variable with name `square`. Then we print the definition (2nd line). On 3rd line, we actually call the function and print its return value.

We will elaborate on functions and how they are defined on a special chapter devoted to this very useful JavaScript tool.

## Operator Overview

Operators are used for JavaScript's arithmetic expressions, comparison expressions, logical expressions, assignment expressions, and more. The following table summarizes the operators and serves as a convenient reference:

Operator	Operation	Number Of Operands
<code>++</code>	pre- and post- increment	1
<code>--</code>	pre- and post- decrement	1
<code>-</code>	negate number	1
<code>+</code>	convert to number	1
<code>~</code>	invert bits	1
<code>!</code>	invert boolean value	1
<code>delete</code>	remove a property	1
<code>typeof</code>	determine type of operand	1
<code>void</code>	return undefined value	1
<code>*, /, %</code>	multiply, divide, remainder	2
<code>+, -</code>	add, subtract	2
<code>+</code>	concatenate strings	2
<code>&lt;&lt;</code>	shift left	2
<code>&gt;&gt;</code>	shift right with sign extension	2
<code>&gt;&gt;&gt;</code>	shift right with zero extension	2
<code>&lt;, &lt;=, &gt;, &gt;=</code>	compare in numeric or alphabetic order	2
<code>instanceof</code>	checks whether object class matches given argument	2
<code>in</code>	checks whether property exists	2
<code>==</code>	test for equality	2
<code>!=</code>	test for inequality	2
<code>===</code>	test for strict equality	2
<code>!==</code>	test for strict inequality	2

Operator	Operation	Number Of Operands
&	compute bitwise AND	2
^	compute bitwise XOR	2
	compute bitwise OR	2
&&	compute logical AND	2
	compute logical OR	2
? :	choose 2nd or 3rd operand (ternary operator)	3
=	assignment operator	2
*=, /=, %=, +=, -=, &=, ^=,  =, <<=, >>=, >>>=	operate and assign	2
,	discard 1st operand, return 2nd	2

Note that the operators, in the above table, are grouped according to their function. Within each group, there is the concept of precedence. The ones listed first have higher precedence from the ones listed last. We will explain this with some examples later on.

## Number Of Operands

Operators can be categorized based on the number of operands they expect (their *arity*).

Most JavaScript operators, like the \* multiplication operator, are *binary operators* that combine two expressions into a single, more complex expression. That is, they expect two operands.

JavaScript also supports a number of *unary operators*, which convert a single expression into a single, more complex expression. For example, The - operator in the expression  $-x$  is a unary operator that performs the operation of negation on the operand  $x$ .

Finally, JavaScript supports one ternary operator, the conditional operator ?:, which combines three expressions into a single expression.

## Example Uses of Operators

### Pre and Post Increment, Pre and Post Decrement

These operators, ++ and -- are unary operators that operate on a single operand. They increase or decrease the value of the operand. The result is actually stored back in the operand variable. Hence, they both change the value and they assign the new value back.

Try this program:

```
1 var number = 1;
2 number++;
3 console.log(number);
4
5 number--;
6 console.log(number);
```

(the above code snippet online)

If you run the above program on JS Bin, you will get this:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the following code:

```
var number = 1;
number++;
console.log(number);

number--;
console.log(number);
```

The output panel shows the results of the console.log statements:

Console
2
1

#### Increment and Decrement Example

Pretty straightforward. You can see that the `++` increases the value stored in variable `number` by 1. The `--` decreases the value by 1.

There is a gotcha however that has to do with the position you use to apply the increment or decrement operator. If you apply that after the operand (post-), then the operand is first used and then incremented (or decremented). If you apply the operator before (pre-) the operand, then the operand is first incremented (or decremented) and then used. Let's make that clear with the following piece of code:

```
1 var n = 1;
2 console.log(n++); // first used and then incremented (post-increment)
3 console.log(n);
```

(the above code snippet online)

will print 1 and then 2. Whereas the program:

```

1 var n = 1;
2 console.log(++n); // first incremented and then used (pre-increment)
3 console.log(n);

```

(the above code snippet online)

will print 2 and then 2 again.

HTML	CSS	JavaScript	Console	Output
JavaScript ▾		<pre> var n = 1; console.log(++n); // first incremented and then used (pre-increment) console.log(n);  n = 1; console.log(n++); // first used and then incremented (post-increment) console.log(n); </pre>	Console	<pre> 1 2 2 2 &gt; </pre>

#### Example of Post and Pre Increment

It works the same way for the post-decrement and the pre-decrement cases for operator ---.

We will learn how other operators are used later on in the following chapters of the JavaScript section.

## Operator Precedence

It is very important to understand the operator precedence, which defines the order in which operators are being evaluated.

Let's take as an example the following expression:

```
1 var result = 5 + 6 * 3 + 2;
```

(the above code snippet online)

What do you think is the value of `result`? One might say that it is 35 which is calculated as  $5 + 6$  which is 11, then  $11 * 3$ , which gives 33 and then plus 2, which finally gives 35. This is wrong. In JavaScript, the `*` has higher precedence than the `+` operator and it's evaluated first. Hence, JavaScript interpreter first evaluates the expression  $6 * 3$ , which give 18 and then the expression  $5 + 18$ , which gives 23 and then the expression  $23 + 2$  which gives 25.

If you want to change the order in which operators are evaluated in a particular expression, then you need to group the expressions in parentheses. For example, if your intention was to first add 5 to 6, then you should have had that in parentheses:

```
1 var result = (5 + 6) * 3 + 2;
```

(the above code snippet online)

So, parentheses, can be used to change the order the operators in an expression are being evaluated.

## Relational Expressions

These expressions use relational operators to test for equality, inequality and other order relations. The operators test whether the operands satisfy the relationship or not and return true or false accordingly.

### Equality and Inequality Operators

There are two equality operators.

1. ==
2. === which is also called strict equality operator or identity operator.

The latter checks whether the two operands are **identical**, whereas the former checks whether its two operands are **equal** in a more relaxed way, in which conversions of type take place before comparison.

Here is an example. Write the following program on JS Bin:

```
1 var a = "1";
2 var b = 1;
3
4 console.log(a == b);
5 console.log(a === b);
```

(the above code snippet online)

and run it. You will get this:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the following code:

```
JavaScript ▾
var a = "1";
var b = 1;

console.log(a == b);
console.log(a === b);
```

The Console tab shows the output:

Console
true
false

A blue arrow points from the last line of the JavaScript code to the 'false' entry in the Console.

Comparing with Equality and Strict Equality

As you can see above, the == returns true when comparing "1" to 1, whereas, the === does not. The former did a type conversion and then compared the two. The latter didn't do that conversion, hence returned false when compared a string to an integer.

To be more specific, the === operator evaluates its operands and then does a comparison without any conversion. While comparing, it takes the decision as follows:

- If the two values have different types, they are not equal.
- If both values are `null` or both values are `undefined`, they are equal.
- If both values are the boolean value `true` or both are the boolean value `false`, they are equal.
- If one or both values is `NaN`, they are not equal. The `NaN` value is never equal to any other value, including itself! To check whether a value `x` is `NaN`, use `x !== x`. `NaN` is the only value of `x` for which this expression will be `true`.
- If both values are numbers and have the same value, they are equal. If one value is `0` and the other is `-0`, they are also equal.
- If both values are strings and contain exactly the same 16-bit values in the same positions, they are equal. If the strings differ in length or content, they are not equal. Two strings may have the same meaning and the same visual appearance, but still be encoded using different sequences of 16-bit values. JavaScript performs no Unicode normalization, and a pair of strings like this are not considered equal to the `==` or to the `==` operators.
- If both values refer to the same object, array, or function, they are equal. If they refer to different objects they are not equal, even if both objects have identical properties.

Obviously, the `==` operator is less strict than the `===` operator:

- If the two values have the same type, test them for strict equality as described above. If they are strictly equal, they are equal. If they are not strictly equal, they are not equal.
- If the two values do not have the same type, the `==` operator may still consider them equal. Use the following rules and type conversions to check for equality:
  - If one value is `null` and the other is `undefined`, they are equal.
  - If one value is a number and the other is a string, convert the string to a number and try the comparison again, using the converted value.
  - If either value is `true`, convert it to `1` and try the comparison again. If either value is `false`, convert it to `0` and try the comparison again.
  - If one value is an object and the other is a number or string, convert the object to a primitive and try the comparison again. An object is converted to a primitive value by either its `toString()` method or its `valueOf()` method. The built-in classes of core JavaScript attempt `valueOf()` conversion before `toString()` conversion, except for the `Date` class, which performs `toString()` conversion. Objects that are not part of core JavaScript may convert themselves to primitive values in an implementation-defined way.
  - Any other combinations of values are not equal.

See another example:

```
1 var a = "1";
2 var b = true;
3
4 console.log(a == b);
5 console.log(a === b);
```

(the above code snippet online)

If you run this, you will see that the non-strict equality operator considers equal the "1" and the true. Whereas the strict equality does not.

You need to be careful which equality operator you are using. Our suggestion is that you use strict equality and only if your program needs to be more relaxed to use, at that particular points, the non-strict equality operator.

Note that we also have the != and !== operators that test for inequality. Otherwise, they have the same logic and strictness like the == and === operators.

## The Other Comparison Operators

The other comparison operators test the relative order (numerical or alphabetic) of their two operands.

1. Less than <
2. Less than or Equal <=
3. Greater than >
4. Greater than or Equal >=

Let's try an example. Write the following code in JS Bin:

```
1 var a = 5;
2 var b = 8;
3 console.log(a > b);
4
5 var firstCustomer = "John Woo";
6 var secondCustomer = "Abraham Foo";
7 console.log(firstCustomer < secondCustomer);
```

(the above code snippet online)

If you run the above program you will get this:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the following code:

```
JavaScript ▾

var a = 5;
var b = 8;
console.log(a > b);

var firstCustomer = "John Woo";
var secondCustomer = "Abraham Foo";
console.log(firstCustomer < secondCustomer);
```

The output in the Console tab shows:

```
Console
false
false
>
```

Example Of Comparison Operators

*Note:* that the uppercase letters are considered less than the lower case letters.

## The `in` Operator

The `in` operator expects a left-side operand that is or can be converted to a string. It expects a right-side operand that is an object. It evaluates to true if the left-side value is the name of a property of the right-side object.

Write the following example on JS Bin:

```
1 var customer = {firstName: "Peter", lastName: "Woo"};
2 console.log("firstName" in customer);
3 console.log("lastName" in customer);
4 console.log("address" in customer);
```

(the above code snippet online)

If you run the above program, you will get this:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the following code:

```
JavaScript ▾

var customer = {firstName: "Peter", lastName: "Woo"};
console.log("firstName" in customer);
console.log("lastName" in customer);
console.log("address" in customer);
|
```

The output in the Console tab shows:

```
Console
true
true
false
>
```

Example of the `in` Operator

As you can see, the first two comparisons return `true`, because both "`firstName`" and "`lastName`" are properties of the object `customer`. The last one returns `false`, because `address` is not property of the object `customer`.

## Logical Expressions

The logical operators `&&`, `||`, and `!` perform Boolean algebra and are often used in conjunction with the relational operators to combine two relational expressions into one more complex expression.

- The logical AND operator is represented with `&&` in JavaScript. It returns `true` if both operands are truthy.
- The logical OR operator is represented with `||` in JavaScript. It returns `true` if either of the operands is truthy.
- The logical NOT operator is represented with `!` in JavaScript. It is an unary operator, and returns `true` if its operand is falsy.

Try the following program:

```
1 var a = true;
2 var b = false;
3 console.log(a && b);
4 console.log(a || b);
5 console.log(!a);
6 console.log(!b);
```

(the above code snippet online)

If you run this program on JS Bin, you will get this:

The screenshot shows a browser-based code editor and console interface. The top navigation bar includes tabs for CSS, JavaScript, Console, and Output. The JavaScript tab is active, displaying the following code:

```
var a = true;
var b = false;
console.log(a && b);
console.log(a || b);
console.log(!a);
console.log(!b);
```

The output window, titled "Console", shows the results of each log statement:

Console
false
true
false
true

A blue arrow icon is located at the bottom right of the output area.

### Logical Operators Example

Note that `&&` has higher precedence than the `||` operator. This means that

```
1 console.log(true || false && false || false);
```

(the above code snippet online)

will print `true`, because it first evaluates the `false && false`, which returns `false` and then evaluates `true || false` which returns `true`, and, finally, it evaluates `true || false`, which returns `true`. We can change the order of evaluation using parentheses. For example, if we want the order to be the order the expressions are written, then we have to do this:

```
1 console.log((true || false) && false || false);
```

(the above code snippet online)

In that case, the result printed will be `false`. Try that on JS Bin, and you will see this:

The screenshot shows a browser-based developer tool interface for testing JavaScript code. At the top, there are tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is active, showing the following code:

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

The output window, titled "Console", displays the results of the code execution:

```
true
false
```

A blue arrow points from the output to the right.

Expression With Boolean Operators - Precedence

## Assignment With Operation

Besides the normal `=` assignment operator, JavaScript supports a number of other assignment operators that provide shortcuts by combining assignment with some other operation. For example, the `+=` operator performs addition and assignment. Try the following example:

```
1 var a = 1;
2 var b = 5;
3 b += a;
4
5 console.log(b);
```

(the above code snippet online)

If you run the above on JS Bin, you will see that it prints 6. This is because the statement `b += a` first adds `a` to `b` and then assigns the new value to `b`. Hence, the statement `b += a` is equivalent to `b = b + a`. It is considered a shortcut.

Refer to the table with all the operators to read about the rest of the assignment-with-operation operators.

## Tasks and Quizzes

**Before you continue, you may want to know that:** You can sign up to [Tech Career Booster](#) and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

## Quiz

The quiz for this chapter can be found [here](#)

## 5 - Statements

### Summary

This chapter is going to introduce you to JavaScript statements. Whereas the previous chapter talked about expressions, this chapter is going to teach you about how you can use expressions to build full-fledged JavaScript statements.

A JavaScript program is simply a sequence of statements, separated from one another with semicolons ; . So once you are familiar with the statements of JavaScript, you can begin writing real JavaScript programs.

You will learn about the compound statements like this:

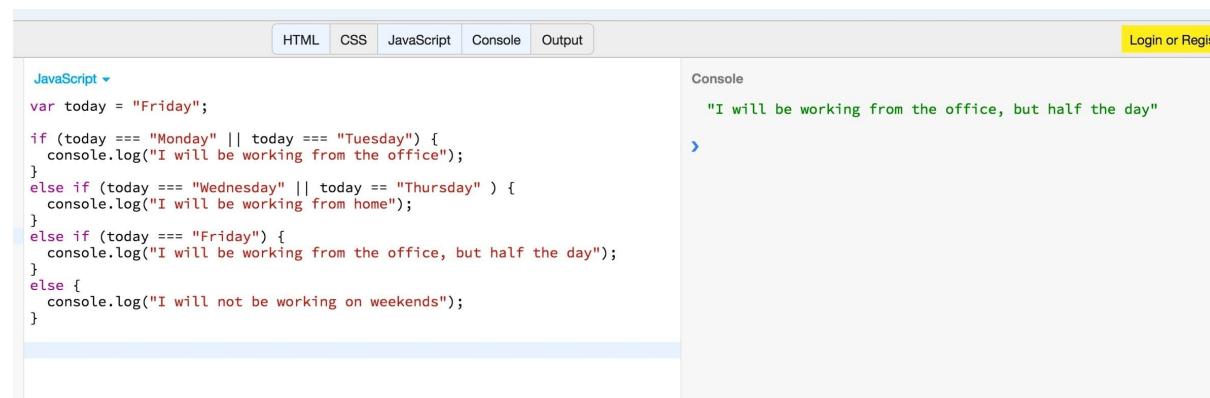
```

1  {
2      customer = "Peter";
3      order = {invoiceDate: '2016-05-16', invoiceNumber: 'ABC123'};
4  }

```

(the above code snippet online)

You are going to learn about conditional statements that help us build branch logic into our programs.



The screenshot shows a browser-based code editor interface. On the left, under the 'JavaScript' tab, there is a code editor with the following content:

```

var today = "Friday";
if (today === "Monday" || today === "Tuesday") {
    console.log("I will be working from the office");
}
else if (today === "Wednesday" || today == "Thursday" ) {
    console.log("I will be working from home");
}
else if (today === "Friday") {
    console.log("I will be working from the office, but half the day");
}
else {
    console.log("I will not be working on weekends");
}

```

On the right, under the 'Console' tab, the output is displayed:

```

I will be working from the office, but half the day
>

```

Conditional Statements

And you are going to learn about loop constructs that help us execute the same block of statements multiple times.

The screenshot shows a browser's developer tools interface with the 'Console' tab selected. In the JavaScript pane, the following code is written:

```
var number = 0;  
do {  
    if (number % 2 === 0) {  
        console.log(number);  
    }  
    number++;  
} while (number <= 10);
```

In the Console pane, the output is:

Console
0
2
4
6
8
10

A blue arrow points to the right at the bottom of the console pane.

### A Loop Example

Finally, you will learn how you can prematurely end a repetition or jump to the next iteration.

## Learning Goals

1. Learn about statements and how they are terminated.
2. Learn about compound statements.
3. Learn about declaration statements.
  1. var
  2. function
4. Learn about conditional statements.
  1. if
  2. else
  3. else if
  4. switch
5. Learn about loop statements.
  1. while
  2. do/while
  3. for
  4. for/in
6. Learn about jump statements.
  1. break
  2. continue

## Introduction

This chapter is going to introduce you to JavaScript statements. Whereas the previous chapter talked about expressions, this chapter is going to teach you about how you can use expressions to build full-fledged JavaScript statements.

A JavaScript program is simply a sequence of statements, separated from one another with semicolons ;. So once you are familiar with the statements of JavaScript, you can begin writing real JavaScript programs.

Let's start.

## Expression Statements

The simplest kind of statements in JavaScript are expressions that have side effects. Assignment statements are one major category of expression statements. For example:

```
1 greeting = "Hello " + name;  
2 i *= 3;
```

(the above code snippet online)

In the above example you can see 2 statements. First statement, `greeting = "Hello " + name;` concatenates 2 strings and assigns the result to the variable `greeting`. Second statement, multiplies `i` by 3 and saves the result back to `i` itself.

As you can see, each statement is terminated with a semicolon;

## Compound Statements

You can combine a series of statements into a compound statement using a *statement block*. A statement block is created by enclosing the statements into curly braces. This is an example of a statement block, which forms a compound statement:

```
1 {  
2     customer = "Peter";  
3     order = {invoiceDate: '2016-05-16', invoiceNumber: 'ABC123'};  
4 }
```

(the above code snippet online)

Note that statement blocks do not end with semicolon.

You can use a compound statement whenever JavaScript expects to find a statement. We will see more concrete examples of this, later in this chapter.

## Declaration Statements

### The var statement

The `var` statement is used to declare variables. One or more.

The `var` keyword is followed by a comma-separated list of variables to declare. Each variable in the list may optionally have an initializer expression that specifies its initial value.

Here are some examples of variable declarations:

```
1  var i;                                // declares a variable without initializing\\
2  it
3  var j = 0;                             // declares a variable and initializes it t\\
4  o have the value 0
5  var p, q;                            // declares two variables. They are separat\\
6  ed with a comma. They are not initialized.
7  var greeting = "Hello" + name;        // declares a variable and uses an expressi\\
8  on to initialize it.
9  var a = 1, b = 2, c = 102, d, e = 30; // declares multiple variables, some of whi\\
10 ch are initialized too.
```

(the above code snippet online)

### function

The `function` keyword is used to define functions. We have already seen that:

```
1  var square = function(x) { return x * x ; };
```

(the above code snippet online)

The above is an expression (`function(x) { return x * x ; }`) assigned to a variable.

However, the `function` keyword may be used to declare a function like this:

```
1  function square(x) { return x * x; }
```

(the above code snippet online)

As you can see, the `function` declaration has a function name. In this example `square`. Note also that the function declaration like that, is not terminated with a semi-colon.

*Note:* We will delve into functions in a later chapter.

## Conditionals

Conditional statements execute or skip other statements depending on the value of a specified expression. These statements are the decision points of your code, and they are also sometimes known as “branches.”

**if**

The very basic statement that can be used for branching is the **if** statement. Let's see an example:

```
1 var age = 35;
2 if (age >= 18)
3   console.log("You are an adult");
```

(the above code snippet online)

If you run the above example on JS Bin, you will get this:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the code: `var age = 35;  
if (age >= 18)  
 console.log("You are an adult");`. The Output tab shows the result: "You are an adult".

Simple Example of if Statement

As you can imagine, the JavaScript interpreter, evaluated the condition `age >= 18` and found it to be true (because `age` had the value 35). Condition being true, JavaScript interpreter executed the statement that is associated to the **if** block, and printed the message `You are an adult`.

The general syntax of the **if** statement is:

```
1 if (condition expression)
2   statement;
```

(the above code snippet online)

Note that, the parentheses that enclose the condition expression are mandatory. The **if** statement is terminated with a semicolon.

However, you may want to execute more than 1 statement, should the condition is true. In that case, you create a compound block of statements, using curly braces. Look at the following example:

```

1 var age = 35;
2 var money = 100;
3 var hasEnoughMoney = true;
4
5 console.log(money);
6
7 if (age >= 18 && hasEnoughMoney) {
8     console.log("You are an adult!");
9     money -= 50;
10 }
11
12 console.log(money);

```

(the above code snippet online)

Run the above in JS Bin. You will get this:

HTML	CSS	JavaScript	Console
		<b>JavaScript</b> <pre> var age = 35; var money = 100; var hasEnoughMoney = true;  console.log(money);  if (age &gt;= 18 &amp;&amp; hasEnoughMoney) {     console.log("You are an adult!");     money -= 50; }  console.log(money); </pre>	<b>Console</b> <pre> 100 "You are an adult!" 50 </pre>

#### If Example With Block Of Statements

The above example uses a more complex expression for condition, and, should the condition be true, it executes two statements. One that prints a message on the console and another one that decreases the value of the variable `money`. Note that the compound block of statements is not terminated by a semicolon.

**else**

There are times that we want to tell JavaScript interpreter to execute a statement when the condition is true and another statement when the condition is false.

For example, we can always do:

```
1 if (age >= 18) {  
2     console.log("You are an adult!");  
3 }  
4 if (age < 18) {  
5     console.log("You are not an adult!");  
6 }
```

(the above code snippet online)

But this is not efficient. Because we are asking JavaScript interpreter to evaluate two conditions that are mutually exclusive. Instead, you can use the keyword `else`, which extends the functionality of `if`. To cover for the previous example:

```
1 if (age >= 18) {  
2     console.log("You are an adult!");  
3 }  
4 else {  
5     console.log("You are not an adult!");  
6 }
```

(the above code snippet online)

I guess this is pretty simple and straightforward. Let's see a more complex example:

```
1 var age = 35;  
2 var money = 100;  
3 var hasEnoughMoney = false;  
4  
5 console.log(money);  
6  
7 if (age >= 18 && hasEnoughMoney) {  
8     console.log("You are an adult!");  
9     money -= 50;  
10 }  
11 else {  
12     console.log("You are not an adult, or you do not have enough money!");  
13 }  
14  
15 console.log(money);
```

(the above code snippet online)

Run the above program on JS Bin. You will get this:

```

HTML CSS JavaScript Console Output Login or Re
JavaScript
var age = 35;
var money = 100;
var hasEnoughMoney = false;

console.log(money);

if (age >= 18 && hasEnoughMoney) {
  console.log("You are an adult!");
  money -= 50;
} else {
  console.log("You are not an adult, or you do not have enough money!");
}

console.log(money);

```

Console

```

100
"You are not an adult, or you do not have enough money!"
100

```

### If Else Example

The JavaScript interpreter executes the statement inside the `else` block, because the condition is not true (`hasEnoughMoney` is `false`).

Note that inside the block statements, either in the `if` part or in the `else` part, you can have, almost, any kind of valid JavaScript statement.

### `else if`

Let's take it a little bit further. There are cases in which we want to check more than one condition in order to do what we want to do. Let's see the following example:

```

1 var today = "Friday";
2
3 if (today === "Monday" || today === "Tuesday") {
4   console.log("I will be working from the office");
5 }
6 else if (today === "Wednesday" || today === "Thursday") {
7   console.log("I will be working from home");
8 }
9 else if (today === "Friday") {
10   console.log("I will be working from the office, but half the day");
11 }
12 else {
13   console.log("I will not be working on weekends");
14 }

```

(the above code snippet online)

If you run this on JS Bin, you will get this:

```

    JavaScript ▾
    var today = "Friday";
    if (today === "Monday" || today === "Tuesday") {
        console.log("I will be working from the office");
    }
    else if (today === "Wednesday" || today === "Thursday" ) {
        console.log("I will be working from home");
    }
    else if (today === "Friday") {
        console.log("I will be working from the office, but half the day");
    }
    else {
        console.log("I will not be working on weekends");
    }
  
```

Console

```
"I will be working from the office, but half the day"
```

### If-Else If Example

The `else if` keywords are used to add extra branching to our logic. The condition on the `else if` will be evaluated only if the previous condition, on previous `if` or `else if` is not true.

### `switch`

When the `if-else if` conditional is testing the value of the same variable, like in the previous example in which we test the value of the variable `today`, we might want to use another version of conditional statement, which is called `switch`.

Let's explain that by writing the previous example using a `switch` statement.

```

1 var today = "Friday";
2
3 switch (today) {
4     case "Monday":
5     case "Tuesday":
6         console.log("I will be working from the office");
7         break;
8     case "Wednesday":
9     case "Thursday":
10        console.log("I will be working from home");
11        break;
12     case "Friday":
13        console.log("I will be working from the office, but half the day");
14        break;
15     default:
16        console.log("I will not be working on weekends");
17        break;
18 }
```

(the above code snippet online)

The `switch` statement above, starts with an expression that is enclosed inside parentheses. The expression here is `today` and when evaluated it returns "Friday", since this is the value of the variable `today`. Then, `switch` is broken into different blocks. Each block starts with one or

more case expressions. The case expression is evaluated and it is compared against the switch expression. The comparison is done using strict equality === operator. So, no conversion takes place. If the comparison returns true, then the block of statements that corresponds to the matching case is being executed. All the statements of the block are executed until the break statement which terminates the switch execution. On this particular example, since the condition on switch evaluates to "Friday", it is the 5th case expression that matches. So, it is its block of statements that are executed.

If no case expression matches the switch expression, then all the statements inside the default block are executed. However, the default block is not mandatory.

Note that both the switch expression and the case expressions can be any JavaScript expression that returns a result. They do not have to be simple expressions only. Also, the break statement is necessary to avoid executing statements that belong to the next case blocks (case blocks that follow the one that matched the switch expression) or to the default block. To understand what we mean, remove the break statement from the 5th matching case and run the program again. What happens?

## Loops

There are cases in which we want to execute a statement, or a series of statements, multiple times. For example, when we want to carry out an action on the elements of an Array. There are special JavaScript statements that allow us to do that.

JavaScript offers the following looping constructs:

1. while
2. do/while
3. for
4. for/in

### **while**

The while is used to carry out a set of instructions while a condition holds true. This is its basic syntax:

```
1 while (condition)
2     statement;
```

(the above code snippet online)

or for many statements:

```
1 while (condition) {  
2     statement 1;  
3     statement 2;  
4 }
```

(the above code snippet online)

The condition that goes with `while` is called a `repeat condition` because it specifies under which conditions the loop will be repeated.

Let's see the following example:

```
1 var number = 0;  
2  
3 while (number <= 10) {  
4     if (number % 2 === 0) {  
5         console.log(number);  
6     }  
7     number++;  
8 }
```

(the above code snippet online)

Write the previous example on JS Bin and run it. You will get this:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the following code:

```
JavaScript ▾  
var number = 0;  
  
while (number <= 10) {  
    if (number % 2 === 0) {  
        console.log(number);  
    }  
    number++;  
}
```

The Console tab shows the output of the code execution:

Console
0
2
4
6
8
10

A blue arrow pointing right is located at the bottom right of the console area.

While Example

The above loop is being executed while the `number` variable is less than or equal to 10. It goes from one number to the next (see: `number++`) and checks, on each repetition, whether the number is an even number. If it is, it prints it on the console.

With `while` loop, it is very important to make sure that the loop terminates at some point after some repetitions (or it never starts at all), in order to avoid infinite loops. This means that you need to make sure that the repeat condition becomes false, sooner or later. In other words, if we were to remove the line that increases the variable `number` on the previous example, then `number` would have always been less than 10, turning this `while` loop into an infinite loop that would never end:

```
1 // If you run this on JS Bin, it will never end. JS Bin will force it to quit. All \
2 so, it wouldn't test other number than 0.
3 var number = 0;
4
5 while (number <= 10) {
6   if (number % 2 === 0) {
7     console.log(number);
8   }
9 }
```

(the above code snippet online)

By the way, when JavaScript interpreter reaches the line of code on which a `while` statement is defined, and its repeat condition is already `false`, then it will never execute any of the statements inside the loop.

### do/while

The `do/while` loop is like a `while` loop, except that the loop condition is tested at the bottom of the loop rather than at the top. This means that the body of the loop is always executed at least once. And this is the difference to `while`. If you want your loop to be executed at least once, then use the `do/while` loop.

Let's write the previous example with a `do/while` block:

```
1 var number = 0;
2
3 do {
4   if (number % 2 === 0) {
5     console.log(number);
6   }
7   number++;
8 } while (number <= 10);
```

(the above code snippet online)

If you run the above program on JS Bin, you will get exactly the same output:

The screenshot shows a developer tool interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is active, displaying the following code:

```
JavaScript ▾
var number = 0;

do {
  if (number % 2 === 0) {
    console.log(number);
  }
  number++;
} while (number <= 10);
```

The Console tab shows the output of the code execution:

Console
0
2
4
6
8
10

A blue arrow points to the right at the bottom of the console output.

Do/While Example

## for

The `for` statement is another very convenient way, and actually more widely used, to write loops in JavaScript. Its syntax allows to combine the initialization of the variable that counts the loops, the repeat condition and the update of the condition variable, all at the start of the `for` statement.

Let's write the previous example with the `for` loop version:

```
1 for (var number = 0; number <= 10; number++) {
2   if (number % 2 === 0) {
3     console.log(number);
4   }
5 }
```

(the above code snippet online)

If you run the above on JS Bin, you will get this:

The screenshot shows a browser's developer tools interface with the 'Console' tab selected. In the 'JavaScript' panel, the following code is written:

```
for (var number = 0; number <= 10; number++) {
  if (number % 2 === 0) {
    console.log(number);
  }
}
```

In the 'Console' panel, the output is displayed as:

0
2
4
6
8
10

A blue arrow points to the right at the bottom of the console list.

### For Example

which is exactly the same like we had with the `while` or `do/while` version.

As you can see from the `for` example syntax above, inside the parentheses, we have 3 parts separated using semicolon ;.

1. The initialization part. On our example it is `var number = 0;`. In the initialization part we can do anything that we want, but we usually do things that will make the next part, the repeat condition, to be true, so that the loop starts and executes at least for once.
2. The repeat condition part. On our example, the repeat condition is `number <= 10;`. The repeat condition is tested exactly before entering the loop statements. Like in the `while` case. Hence, if this condition is initially `false`, then `for` loop will never be executed.
3. The update part. This is executed at the end of each execution repetition. Usually, we update the variable that the condition evaluates, so that at some point in the future, the condition becomes `false` and the loop terminates. Hence, we avoid infinite loops.

One way you could look at the `for` loop, is if you wrote it like that:

```
1 var number = 0;
2 for ( ; number <= 10; ) {
3   if (number % 2 === 0 ) {
4     console.log(number);
5   }
6   number++;
7 }
```

(the above code snippet online)

The above version of `for` loop is similar to the `while` loop and they do exactly the same. But we never write `for` loops like that.

### for/in

The `for/in` construct is used to loop through the properties of an object. Let's see that with an example:

```

1 var order = {
2   invoiceDate: "2016-05-06",
3   invoiceNumber: "ABC123",
4   invoiceAmount: 120.00
5 };
6
7 for (var p in order) {
8   console.log(p + ": " + order[p]);
9 }
```

(the above code snippet online)

If you run this on JS Bin, you will get the following:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the provided code snippet. The Console tab shows the output of the console.log statements: "invoiceDate: 2016-05-06", "invoiceNumber: ABC123", and "invoiceAmount: 120". A blue arrow points to the right at the bottom of the console output.

For/in Example

As you can see, on each iteration, the `p` variable holds the name of a property from object `order`. On first iteration, it holds the value "`invoiceDate`", on second iteration it holds the value "`invoiceNumber`" and on third iteration it holds the value "`invoiceAmount`".

Note also that calling `order[p]` allows us to get the *value* of the corresponding object property.

## Jump statements

### **break statement**

The `break` statement is usually used to prematurely end a loop. Let's see an example:

```
1 var number = 0;
2
3 while (number <= 1000) {
4     if (number % 2 === 0) {
5         console.log(number);
6     }
7     if (number === 10) {
8         break;
9     }
10    number++;
11 }
```

(the above code snippet online)

The above JavaScript code prints all the even numbers from 0 up to 10. It may have a repeat condition that implies that the loop is executed up to 1000, but this is not the case, due to the `break` statement inside `while` loop. You can see that the `break` statement is executed if the number that has been processed is equal to 10. This means, in other words, that when the number reaches 10, loop will be terminated.

### continue Statement

The `continue` statement is used to ask the loop to move to the next iteration. Let's see the following example.

```
1 for (var number = 0; number <= 100; number++) {
2     if (number < 90) {
3         continue;
4     }
5     if (number % 2 === 0) {
6         console.log(number);
7     }
8 }
```

(the above code snippet online)

What do you think that this program does? It prints all the even numbers starting from 90 and ending to 100. This is because the first numbers from 0 up to 89 are skipped, due to the `if (number < 90) { continue; }` statement. While the number is less than 90, the `continue` statement is executed, which terminates the iteration and goes to third part of the `for` loop, which is the update part, before going back inside the loop, as long as the repeat condition is true of course.

Hence, `continue` is useful when you want to skip processing specific items, while processing a set of items using a loop.

## Tasks and Quizzes

**Before you continue, you may want to know that:** You can sign up to [Tech Career Booster](#) and have a mentor evaluate your tasks, your quizzes and, generally, your

progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

## Task Details

- (1) You need to write a JavaScript program that finds the maximum number among three numbers. Here is the start of it:

```
1 var a = 5, b = 3, c = 2;  
2 ...
```

You need to fill in the rest of the program that would print the maximum number among the tree stored in a, b or c. Hint: Use if statements.

**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

- (2) You need to write a JavaScript program that converts an integer number to the week day name. Here is the start of it:

```
1 var today = 2;  
2 ...
```

Depending on the value of today it needs to print the name of the weekday. Hence, for 1, it should print Sunday, for 2, it should print Monday and so on. If the number stored in today is not in the range 1-7, then it should print the message "Invalid Date". Hint: Use switch statement.

**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

- (3) You need to write a JavaScript program that prints all the odd numbers starting from 1 up to 51. Note that it should skip the numbers 13, 23 and 45.

**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

## 6 - Objects

### Summary

This is going to be a first encounter with objects in JavaScript. You will learn the basics of objects, how we can create them and how we can reference their properties.

There are many ways you can create an object in JavaScript, and this is only an example:

The screenshot shows a browser's developer tools interface with the 'Console' tab selected. At the top, there are tabs for HTML, CSS, JavaScript, Console, and Output. Below the tabs, the JavaScript pane contains the following code:

```
function Order(invoiceDate, invoiceNumber) {
  var result = {
    invoiceDate: invoiceDate,
    invoiceNumber: invoiceNumber
  };
  return result;
}

var order = new Order("2015-05-16", "ABC2341");

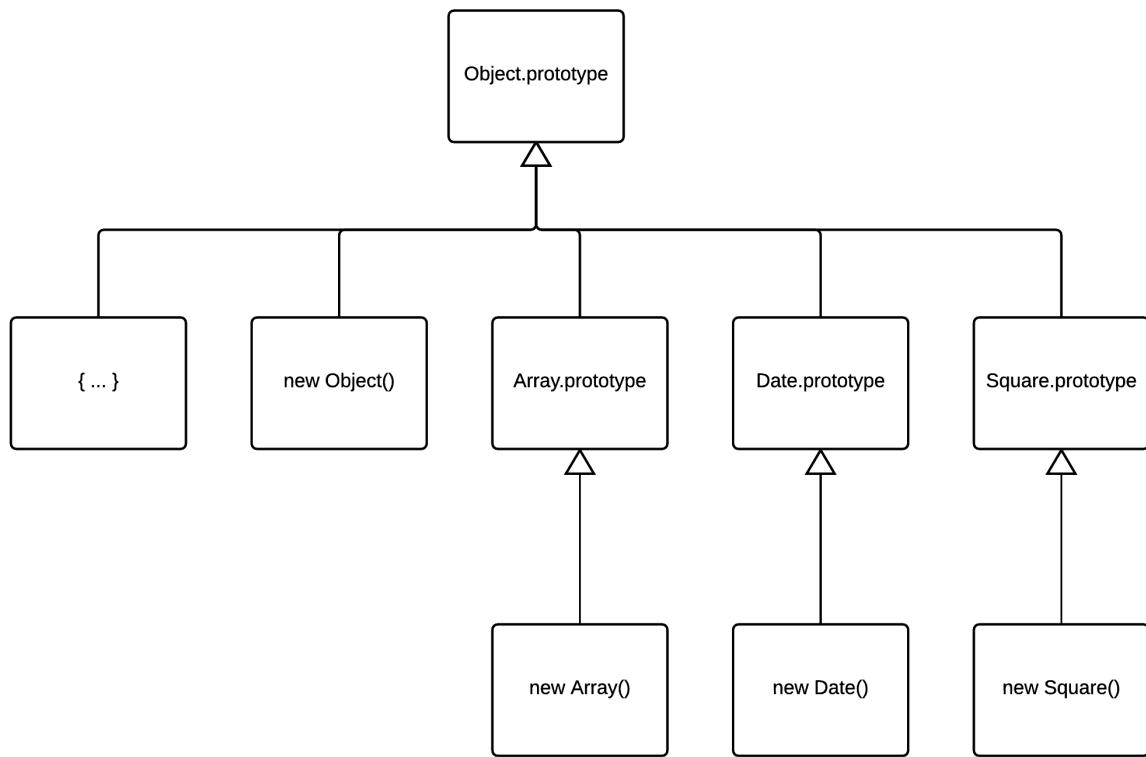
console.log(order);
```

In the 'Console' pane, the output is:

```
[object Object] {
  invoiceDate: "2015-05-16",
  invoiceNumber: "ABC2341"
}
```

Example of a Function Constructor

It is very crucial for you to understand the object inheritance rules:



### Object Prototype Chain

and how you can create methods and attach them to objects, in order to provide them with behaviour.

The screenshot shows a browser-based developer tools interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the following code:

```

var dog = {
  breed: "Beagle",
  bark: function() {
    console.log("WavWavWav");
  }
};

console.log(dog);
dog.bark();
  
```

The Console tab shows the output of this code. It first logs the object itself, then calls the `bark` method, which outputs "WavWavWav".

Creating An Object with A Method

## Learning Goals

1. Learn about objects.
2. Learn about object properties.
3. Learn to create objects using object literals.
4. Learn to create objects using the `new` keyword.
5. Learn to create objects using `Object.create()` function.

6. Learn about prototypes.
7. Learn about the `Object.prototype`.
8. Learn about the Object Prototype Chain.
9. Learn how to access the value of a property (query a property).
10. Understand the difference in accessing properties using dot notation vs square brackets.
11. Learn about object inheritance.
12. Learn about object own properties.
13. Learn about cascading inheritance.
14. Learn about properties which are methods.
15. Learn how methods are accessing object properties.

## Introduction

An object is an unordered collection of properties, each of which has a name and a value attribute. Property names are strings, so we can say that objects map strings to values. This string-to-value mapping goes by various names:

- hash
- hashtable
- dictionary
- associative array

Any value in JavaScript that is not a string, a number, `true`, `false`, `null`, or `undefined` is an object. And even though strings, numbers, and booleans are not objects, they behave like immutable objects.

## Creating Objects

Objects can be created with the following methods:

- with object literals
- with the `new` keyword
- with the `Object.create()` function

## Object Literals

The easiest way to create an object is to include an object literal in your JavaScript code. An object literal is a comma-separated list of colon-separated name:value pairs, enclosed within curly braces. A property name is a JavaScript identifier or a string literal (the empty string is allowed). A property value is any JavaScript expression. The value of the expression (it may be a primitive value or an object value) becomes the value of the property. Here are some examples:

```

1 var emptyObject = {}; // This is an object without any properties
2 var point = {x: 0, y: 0}; // Object with two properties
3 var point2 = {x: point.x, y: point.y + 1}; // Object with two properties with values calculated from other object
4
5 var book = {
6   title: "Of Mice And Men",
7   author: {
8     firstname: "John",
9     surname: "Steinbeck"
10  }
11};

```

(the above code snippet online)

## new

The `new` operator creates and initializes a new JavaScript object. It should be followed by a function invocation. In that case, the function is called a constructor, and serves to initialize a newly created object.

Here is an example JavaScript code that demonstrates the function constructors:

```

1 function Order(invDate, invNumber) {
2   this.invoicedate = invDate;
3   this.invoiceNumber = invNumber;
4 }
5
6 var order = new Order("2015-05-16", "ABC2341");
7
8 console.log(order);

```

(the above code snippet online)

If you run the above on JS Bin, you will see this:

The screenshot shows a JS Bin interface. On the left, there is a code editor with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is active, containing the following code:

```

function Order(invDate, invNumber) {
  this.invoicedate = invDate;
  this.invoiceNumber = invNumber;
}

var order = new Order("2015-05-16", "ABC2341");

console.log(order);

```

On the right, there is a 'Console' tab showing the output of the code execution. It displays the following message:

```

[object Object]
  invoicedate: "2015-05-16",
  invoiceNumber: "ABC2341"

```

Example of a Function Constructor

As you can read above, the function Order creates an object using an object literal which bases the values of its properties on the values given as arguments when calling the constructor.

You will learn more about functions in later chapters.

Let's now create two instances of the Order type:

```

1  function Order(invDate, invNumber) {
2      this.invoicedate = invDate;
3      this.invoiceNumber = invNumber;
4  }
5
6  var order1 = new Order("2017-05-16", "ABC2341");
7  var order2 = new Order("2017-08-13", "DEF2409");
8
9  console.log(order1);
10 console.log(order2);

```

(the above code snippet online)

If you run the above in JSBin, you will see this:

The screenshot shows a JSBin interface with two main sections: 'JavaScript' and 'Console'. In the 'JavaScript' section, the provided code is pasted. In the 'Console' section, two objects are logged: `[object Object]` and `[object Object]`. The first object has properties `invoicedate: "2017-05-16"` and `invoiceNumber: "ABC2341"`. The second object has properties `invoicedate: "2017-08-13"` and `invoiceNumber: "DEF2409"`.

```

JavaScript ▾
function Order(invDate, invNumber) {
    this.invoicedate = invDate;
    this.invoiceNumber = invNumber;
}
var order1 = new Order("2017-05-16", "ABC2341");
var order2 = new Order("2017-08-13", "DEF2409");
console.log(order1);
console.log(order2);

Console
[object Object] {
  invoicedate: "2017-05-16",
  invoiceNumber: "ABC2341"
}
[object Object] {
  invoicedate: "2017-08-13",
  invoiceNumber: "DEF2409"
}
>

```

Function Constructors - Creating Two Instances

Let's now also attach a behaviour to the object:

```

1  function Order(invDate, invNumber) {
2      this.invoiceDate = invDate;
3      this.invoiceNumber = invNumber;
4      this.generateInvoice = function() {
5          console.log("I will generate invoice. Date: " +
6              this.invoiceDate +
7              ", invoiceNumber: " +
8              this.invoiceNumber);
9      }
10 }
11
12 var order1 = new Order("2017-05-16", "ABC2341");

```

```

13 var order2 = new Order("2017-08-13", "DEF2409");
14
15 console.log(order1);
16 console.log(order2);
17
18 order1.generateInvoice();
19 order2.generateInvoice();

```

(the above code snippet online)

If you run the above in JSBin, you will get this:

```

JavaScript ▾
function Order(invDate, invNumber) {
  this.invoiceDate = invDate;
  this.invoiceNumber = invNumber;
  this.generateInvoice = function() {
    console.log("I will generate invoice. Date: " +
      this.invoiceDate +
      ", invoiceNumber: " +
      this.invoiceNumber);
  }
}
var order1 = new Order("2017-05-16", "ABC2341");
var order2 = new Order("2017-08-13", "DEF2409");

console.log(order1);
console.log(order2);

order1.generateInvoice();
order2.generateInvoice();
|
```

Console

```

[object Object] {
  generateInvoice: function () {
    window.runnerWindow.proxyConsole.log("I will generate invoice.
Date: " +
      this.invoiceDate +
      ", invoiceNumber: " +
      this.invoiceNumber);
  },
  invoiceDate: "2017-05-16",
  invoiceNumber: "ABC2341"
}

[object Object] {
  generateInvoice: function () {
    window.runnerWindow.proxyConsole.log("I will generate invoice.
Date: " +
      this.invoiceDate +
      ", invoiceNumber: " +
      this.invoiceNumber);
  },
  invoiceDate: "2017-08-13",
  invoiceNumber: "DEF2409"
}

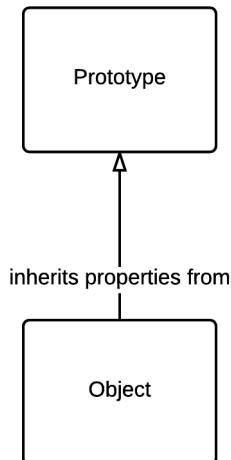
"I will generate invoice. Date: 2017-05-16, invoiceNumber: ABC2341"
"I will generate invoice. Date: 2017-08-13, invoiceNumber: DEF2409"
>
```

### Function Constructor With Methods Too

The problem with this method of creating objects, is that each instance has its own copy of the definition of the object, which is not ideal. It may be useful in some cases, but generally, is not very useful and consumes more memory.

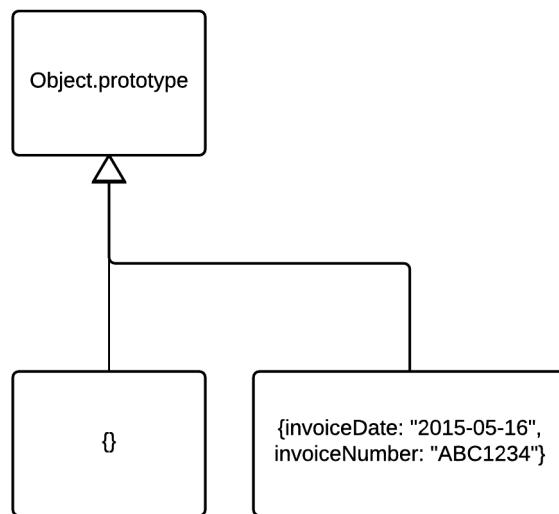
## Prototypes

Before we can cover the third method of creating objects, we need to talk a little bit about Prototypes. Every JavaScript object has a second object (or `null`, but this is rare) associated with it. The second object is known as a *prototype*. The prototype gives its properties to the actual object, or in other words, the object inherits properties from its prototype.



#### Object Inherits Properties From Its Prototype

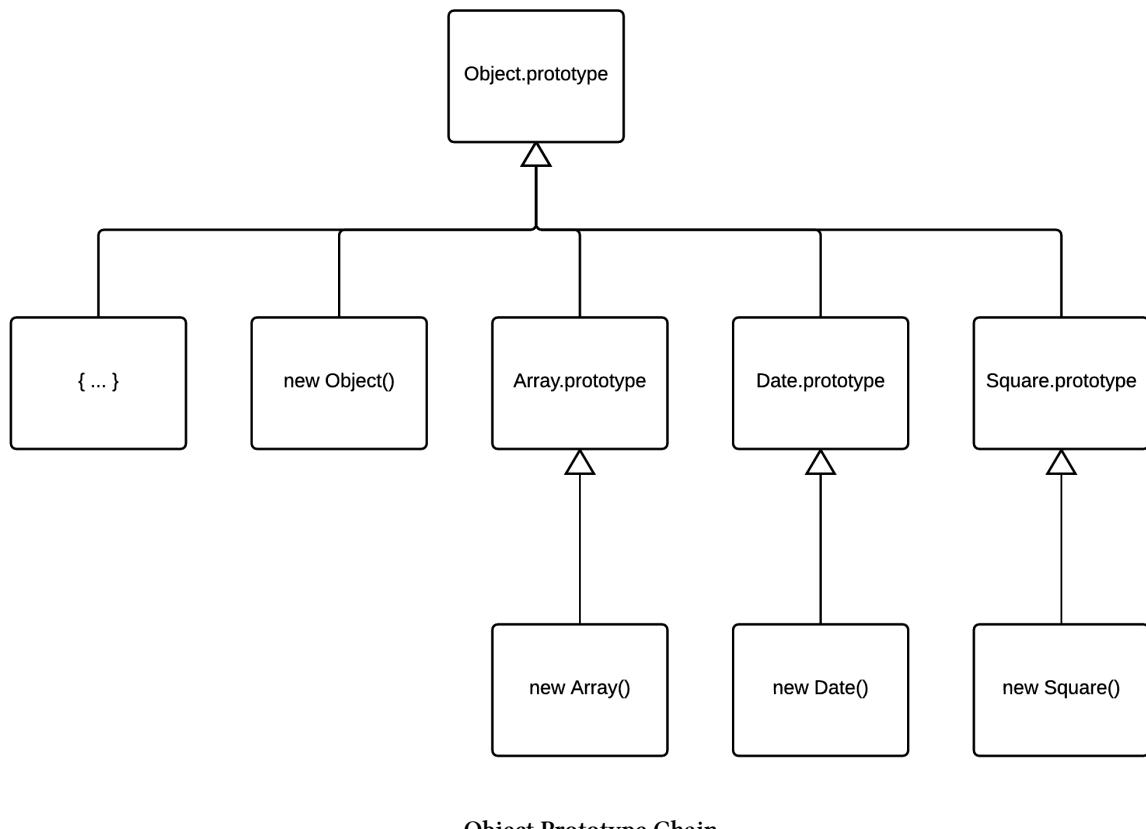
All objects created by object literals have the same prototype object, and we can refer to this prototype object in JavaScript code as `Object.prototype`.



#### Object Created by Object Literals Inherit from Object.prototype

Objects created using the `new` keyword and a constructor invocation use the value of the `prototype` property of the constructor function as their prototype. So the object created by `new Object()` inherits from `Object.prototype` just as the object created by `{}` does. Similarly, the object created by `new Array()` uses `Array.prototype` as its prototype, and the object created by `new Date()` uses `Date.prototype` as its prototype.

The `prototype` property of the constructor function is usually `Object.prototype`. Hence, `new Date()` inherits properties from `Date.prototype` which inherits properties from `Object.prototype`.



Object Prototype Chain

## Creating objects with `Object.create()`

`Object.create()` creates a new object, using its first argument as the prototype of that object. `Object.create()` also takes an optional second argument that describes the properties of the new object.

The following creates an object with two properties:

```

1 var point = Object.create({x: 0, y: 0});
2 console.log(point);

```

(the above code snippet online)

If you run the above on JS Bin you will see a properly object created:

The screenshot shows a browser's developer tools console interface. At the top, there are tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is selected, indicated by a dropdown arrow icon. Below the tabs, the code `var point = Object.create({x: 0, y: 0}); console.log(point);` is entered into the console. To the right, the Console panel displays the output: `[object Object] { x: 0, y: 0 }`. A blue arrow icon is located at the bottom right of the console area.

### Object Create Example

The `{x: 0, y: 0}` is now considered the prototype of the object `point`.

If you want to create an empty object using `Object.create()` you need to give as first argument the `Object.prototype`, the one that is the prototype of objects created with `{}`.

```
1 var x = Object.create(Object.prototype);
2 console.log(x);
```

(the above code snippet online)

If you just call `Object.create()` without giving any prototype, you create an object that does not have the basic properties of an object. For example, it does not respond to `toString()`. Generally, it is not very useful to do that.

## Querying and Setting Properties

To obtain the value of a property, use the dot `.` or square bracket `[]`. The left-hand side should be an expression whose value is an object. If using the dot operator, the right-hand must be a simple identifier that names the property. If using square brackets, the value within the brackets must be an expression that evaluates to a string that contains the desired property name.

Write the following program on JS Bin:

```
1 var order = {
2   invoiceDate: "2015-05-16",
3   invoiceNumber: "ABD38292"
4 };
5
6 console.log(order.invoiceDate);
7 console.log(order.invoiceNumber);
8 console.log(order["invoiceDate"]);
9 console.log(order["invoiceNumber"]);
```

(the above code snippet online)

You will see this:

The screenshot shows a browser-based developer tool interface for running JavaScript code. At the top, there are tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is active, showing the following code:

```
JavaScript ▾
var order = {
  invoiceDate: "2015-05-16",
  invoiceNumber: "ABD38292"
};

console.log(order.invoiceDate);
console.log(order.invoiceNumber);
console.log(order["invoiceDate"]);
console.log(order["invoiceNumber"]);
```

To the right, under the 'Console' tab, the output is displayed in green text:

```
"2015-05-16"
"ABD38292"
"2015-05-16"
"ABD38292"
```

### Examples Querying Properties of an Object

Note that when accessing the value of a property of an object using the square brackets operator [ ], we are basically accessing the object as if it were a hash or a hashtable or a dictionary or a map. These are commonly used names for accessing a bucket elements using a key. Another name is associative array, because the [ ] is also used to access array elements by their position/index in the array.

Accessing object properties with expressions that return a string, which is the name of the property, is very flexible and can be proven very handy on dynamic programming. String keys to access object properties are dynamic whereas dot notation uses identifiers that are static.

With JavaScript you can dynamically add a new property to an object. You only have to assign a new value to it. Let's add the property `customerName` on `order` object:

```
1 var order = {
2   invoiceDate: "2015-05-16",
3   invoiceNumber: "ABD38292"
4 };
5
6 console.log(order.invoiceDate);
7 console.log(order.invoiceNumber);
8 console.log(order["invoiceDate"]);
9 console.log(order["invoiceNumber"]);
10
11 order.customerName = "Peter Pan";
12 console.log(order);
```

(the above code snippet online)

We have added two more lines to the previous program. If you run that on JS Bin, you will get this:

The screenshot shows a browser's developer tools with the 'Console' tab selected. In the JavaScript pane, the following code is run:

```

var order = {
  invoiceDate: "2015-05-16",
  invoiceNumber: "ABD38292"
};

console.log(order.invoiceDate);
console.log(order.invoiceNumber);
console.log(order["invoiceDate"]);
console.log(order["invoiceNumber"]);

order.customerName = "Peter Pan";
console.log(order);

```

In the Console pane, the output is:

```

"2015-05-16"
"ABD38292"
"2015-05-16"
"ABD38292"
[object Object] {
  customerName: "Peter Pan",
  invoiceDate: "2015-05-16",
  invoiceNumber: "ABD38292"
}

```

### Adding A Property To An Existing Object

Of course, using the assignment operator, you can always update the value of an existing property.

## Inheritance

We will now talk a little bit about inheritance, i.e. how some properties of an object are inherited from its prototype. Note that there is a difference with regards to inheritance behaviour when we are querying a property vs when we are setting a property. We will examine this difference here with some examples.

An object has a set of own properties. But, besides the “own” properties, an object inherits properties from its *prototype* object. Let’s try to explain that using some examples:

```
1 var o = {};
```

(the above code snippet online)

Object o does not have any own property. Let’s create a property x on this object.

```
1 var o = {};
2 o.x = 1;
```

(the above code snippet online)

Now , object o has one property on its own. Let’s create another object that inherits from o.

```
1 var o = {};
2 o.x = 1;
3 var p = Object.create(o);
4 console.log(p);
```

(the above code snippet online)

If you run the above, you will see that object p has the property x with value 1. This is because it inherits it from object o.

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the following code:

```
1 var o = {};
2 o.x = 1;
3 var p = Object.create(o);
4 console.log(p);
```

The Console tab shows the output of the code execution:

```
[object Object] {
  x: 1
}
```

#### Example of Property Inheritance

Now, let's set a property on p object. And create another object that inherits from p. What would that object have as properties?

```
1 var o = {};
2 o.x = 1;
3 var p = Object.create(o);
4 console.log(p);
5
6 p.y = 2;
7 var q = Object.create(p);
8 console.log(q);
```

(the above code snippet online)

If you run the above on JS Bin, you will see this:

The screenshot shows a browser's developer tools interface with a tab bar at the top labeled 'HTML', 'CSS', 'JavaScript', 'Console', and 'Output'. The 'JavaScript' tab is active. Below it, the code editor section contains the following JavaScript code:

```
JavaScript ▾
var o = {};
o.x = 1;
var p = Object.create(o);
console.log(p);

p.y = 2;
var q = Object.create(p);
console.log(q);
```

The 'Console' section to the right displays two objects. The first object is the prototype object p, which has a property x with value 1. The second object is the object q, which has properties x (value 1) and y (value 2).

### Multi-Level Inheritance

As you can see, the property x and the property y have been inherited to the object q. This is an example of cascading inheritance.

Object q, besides the properties that it has inherited from objects p and o, it has properties from `Object.prototype`. Look at this:

```
1 var o = {};
2 o.x = 1;
3 var p = Object.create(o);
4 console.log(p);
5
6 p.y = 2;
7 var q = Object.create(p);
8 console.log(q);
9
10 console.log(q.toString());
```

(the above code snippet online)

The last statement, invokes the property `toString()` which has been inherited from `Object.prototype`. This is what it prints at the end:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the following code:

```

1 var o = {};
2 o.x = 1;
3 var p = Object.create(o);
4 console.log(p);

5 p.y = 2;
6 var q = Object.create(p);
7 console.log(q);

8 console.log(q.toString());

```

The Console tab shows the output of the code:

```

[object Object] {
  x: 1
}

[object Object] {
  x: 1,
  y: 2
}

"[object Object]"

```

A blue arrow points from the last line of the output back to the `toString()` call in the code.

`toString()` has been inherited from `Object.prototype`

Now, let's set a new value on property `y` on object `q`. Remember, that `q` does now own `y`. It is inherited from `p` object. Will that assignment affect the `y` property of `p` object?

```

1 var o = {};
2 o.x = 1;
3 var p = Object.create(o);
4 console.log(p);

5
6 p.y = 2;
7 var q = Object.create(p);
8 console.log(q);

9
10 console.log(q.toString());

11
12 q.y = 3;
13 console.log("p object:");
14 console.log(p);

15
16 console.log("q object:");
17 console.log(q);

```

(the above code snippet online)

If you run the above program on JS Bin, you will see this:

The screenshot shows a browser's developer tools console tab. The JavaScript tab contains the following code:

```

var o = {};
o.x = 1;
var p = Object.create(o);
console.log(p);

p.y = 2;
var q = Object.create(p);
console.log(q);

console.log(q.toString());

q.y = 3;
console.log("p object:");
console.log(p);

console.log("q object:");
console.log(q);

```

The output tab shows the results of the console.log statements:

- [object Object] { x: 1 }
- [object Object] { x: 1, y: 2 }
- "[object Object]"
- "p object:"
- [object Object] { x: 1, y: 2 }
- "q object:"
- [object Object] { x: 1, y: 3 }

A blue arrow points from the last log entry "y: 3" towards the text "Setting property y on q object" below.

Setting property y on q object

As you can see, setting the property `y` on `q` object does not affect the `y` property on the `p` object. Actually, a new own property for `q` is created that hides the inherited `y` coming from `p`.

## Methods

Object properties, may have values which are function definitions. In that case, the functions are called methods. Let's see an example:

```

1 var dog = {
2   breed: "Beagle",
3   bark: function() {
4     console.log("WavWavWav");
5   }
6 };
7
8 console.log(dog);
9 dog.bark();

```

(the above code snippet online)

On the above program, we define the object `dog` and we set two properties. One simple property, named `breed`, with the value "Beagle", and another property, named `bark` with the value

`function() { console.log("WavWavWav"); }` being a function definition. Then we first display the whole object and then we call the method `bark()` on the object `dog`. The last statement is what we call the behaviour of the object.

Run the above program on JS Bin and you will see this:



The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the following code:

```
JavaScript
var dog = {
  breed: "Beagle",
  bark: function() {
    console.log("WavWavWav");
  }
};

console.log(dog);
dog.bark();
```

The Console tab shows the output of the code execution:

```
[object Object] {
  bark: function () {
    window.runnerWindow.proxyConsole.log("WavWavWav");
  },
  breed: "Beagle"
}
"WavWavWav"
```

#### Creating An Object with A Method

Methods usually access the object properties. Let's try another example:

```
1 var polynomial = {
2   a: 10,
3   b: 20,
4   result: function() {
5     return (this.a * this.a + 2 * this.a + this.b);
6   }
7 };
8
9 console.log(polynomial.result());
```

(the above code snippet online)

This is an object with 3 properties. Its property `result` is a method that calculates the formula  $a^2 + 2a + b$ . In order for the method to have access to the properties of the object, we use the keyword `this`.

## Closing Note

That was a very basic introduction to JavaScript objects. Objects in JavaScript have a lot of more powers than the ones presented here. However, our intention was to give you the foundation so that you can proceed with the rest of the course on front-end Web development. If you really want to master object oriented programming with JavaScript, you might want to take a JavaScript dedicated course. By the way, you are going to learn lots of new stuff on object oriented programming while we will be teaching you the Ruby language, later on in this course.

## Tasks and Quizzes

**Before you continue, you may want to know that:** You can sign up to [Tech Career Booster](#) and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

### Quiz

The quiz for this chapter can be found [here](#)

## 7 - Arrays

### Summary

This chapter is going to teach you about one of the most popular programming constructs in JavaScript. The Array.

Arrays can hold a huge number of elements and JavaScript can be used to carry out operations on each one of these elements with unconceivable speed. This is where programs and computers show their real power.

You are going to learn how to create arrays. You will also learn how to create multi-dimensional arrays. An example of such an array is a matrix like this:

The screenshot shows a browser's developer tools interface with tabs for CSS, JavaScript, Console, and Output. The JavaScript tab is active, displaying the following code:

```
JavaScript ▾  
var matrix = [  
    [ 4, 5, 6 ],  
    [ 9, 8, 2 ],  
    [ 10, 3, 25 ]  
];  
  
console.log(matrix);
```

The output in the Console tab shows the resulting matrix:

```
[[4, 5, 6], [9, 8, 2], [10, 3, 25]]  
▶
```

Two-dimensional Array Example

You will learn how to access the elements of an array using the [] operator.

The screenshot shows a browser's developer tools interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is active, displaying the following code:

```
JavaScript ▾  
var employees = [  
    {firstName: "George", lastName: "M"},  
    {firstName: "Mary", lastName: "L"},  
    {firstName: "John", lastName: "K"}  
];  
  
for (var i = 0; i < employees.length; i++) {  
    console.log("Employee (" + i + "): " + employees[i].firstName);  
}
```

The output in the Console tab shows the printed names:

```
"Employee (0): George"  
"Employee (1): Mary"  
"Employee (2): John"  
▶
```

Example Accessing Array Elements

Then you will learn about various techniques that you can use to add and delete elements from an array. Like, for example, with the use of the push() method.

The screenshot shows a browser's developer tools interface with the JavaScript tab selected. In the left panel, the following code is written:

```
var products = [];
products.push("Apples");
console.log(products.length);
console.log(products);

products.push("Oranges", "Grapes");
console.log(products.length);
console.log(products);
```

In the right panel, the 'Console' tab is selected, showing the output of the code:

```
1
["Apples"]
3
["Apples", "Oranges", "Grapes"]
```

### Using push() to Add Elements To an Array

Then you are going to learn about the most popular Array methods. Like, for example, the `slice()` method:

The screenshot shows a browser's developer tools interface with the JavaScript tab selected. In the left panel, the following code is written:

```
var products = ["Apples",
  "Tomatoes",
  "Grapes",
  "Oranges"];

console.log(products.slice(1, -1));
console.log(products);
```

In the right panel, the 'Console' tab is selected, showing the output of the code:

```
["Tomatoes", "Grapes"]
["Apples", "Tomatoes", "Grapes", "Oranges"]
```

### A slice() Example

Finally, we are going to teach you how Strings are sometimes treated like Arrays.

## Learning Goals

1. Learn how to create arrays using array literals
2. Learn about the `length` property of the arrays.
3. Learn how you can create multi-dimensional arrays.
4. Learn how you can access array elements, both for reading and for writing.
5. Learn how to add and delete elements.
6. Learn about the `push()` method.
7. Learn about the `pop()` method.
8. Learn about the `shift()` method.
9. Learn about the `splice()` method.
10. Learn about the `join()` method.
11. Learn about the `reverse()` method.
12. Learn about the `sort()` method.

13. Learn about the `concat()` method.
14. Learn about the `slice()` method.
15. Learn about the `unshift()` method.
16. Learn about the `forEach()` method.
17. Learn about the `map()` method.
18. Learn about the `filter()` method.
19. Learn about the `every()` method.
20. Learn about the `some()` method.
21. Learn about the `reduce()` method.
22. Learn about the `indexOf()` and `lastIndexOf()` methods.
23. Learn how Strings are sometimes treated as Arrays.

## Introduction

Arrays are one of the most useful structures in any programming language. They are ordered collection of items. In JavaScript, the items can be of any type. This means that the array can contain either integers, or strings or objects, or functions or any other type. Also, the items of an array do not have to be of the same type. This makes JavaScript arrays very powerful.

## Creating Arrays

The easiest method to create an array is with array literals. An array literal is a comma separated list of values that will be the items of the array. This list of values is given enclosed in square brackets.

Let's try some array creation with array literals:

```
1 var emptyArray = [];
2 console.log(emptyArray);
3
4 var arrayOfIntegers = [3, 2, 5];
5 console.log(arrayOfIntegers);
6
7 var arrayOfStringsAndIntegers = ["Hello", 5, 2, "World"];
8 console.log(arrayOfStringsAndIntegers);
9
10 var arrayOfVariousTypeItems = [1.5, 3, "H", function() {return 2*2;}, {a: "A", b:\
11   "B"}];
12 console.log(arrayOfVariousTypeItems);
```

(the above code snippet online)

Type in the above program on JS Bin. When you run it you get something like this:

The screenshot shows a browser's developer tools console tab selected. The code in the left pane demonstrates four different ways to create arrays:

```

var emptyArray = [];
console.log(emptyArray);

var arrayOfIntegers = [3, 2, 5];
console.log(arrayOfIntegers);

var arrayOfStringsAndIntegers = ["Hello", 5, 2, "World"];
console.log(arrayOfStringsAndIntegers);

var arrayOfVariousTypeItems = [1.5, 3, "H", function() {return 2*2;}, {a: "A", b: "B"}];
console.log(arrayOfVariousTypeItems);

```

The right pane shows the corresponding output in the console:

```

[]
[3, 2, 5]
["Hello", 5, 2, "World"]
[1.5, 3, "H", function () {return 2*2;}, {"a": "A", "b": "B"}]
>

```

### Creating Arrays - Examples

The first line, `var emptyArray = [];` creates an empty array. We are using the square brackets notation, without including any elements inside.

The line `var arrayOfIntegers = [3, 2, 5];` creates an array with 3 elements, 3 integers.

The line `var arrayOfStringsAndIntegers = ["Hello", 5, 2, "World"];` creates an array with 4 elements, of different types, 2 strings and 2 integers.

The line `var arrayOfVariousTypeItems = [1.5, 3, "H", function() {return 2*2;}, {"a": "A", "b": "B"}];` creates an array with 5 elements. A number, 1.5, an integer 3, a string "H", a function, and, finally, an object created using object literal.

Note that the arrays have a property `length`, which returns the number of elements in the array. Append the following line to the above program:

```
1 console.log(arrayOfVariousTypeItems.length);
```

[\(the above code snippet online\)](#)

If you run the program, you will see that it will print 5 as the length of the last array created.

An array can also contain elements that are other arrays. This is the way we can create multi-dimensional arrays. Let's say, for example, that we want to create a JavaScript variable holding this two-dimensional array of integers:

```
1 -      -
2 | 4 5 6 |
3 | 9 8 2 |
4 | 10 3 25 |
5 -      -
```

[\(the above code snippet online\)](#)

We can do that with the following JavaScript code:

```
1 var matrix = [
2   [ 4, 5, 6],
3   [ 9, 8, 2],
4   [10, 3, 25]
5 ];
6
7 console.log(matrix);
```

(the above code snippet online)

If you run the above program on JS Bin, you will get this:

The screenshot shows a JS Bin interface with four tabs: CSS, JavaScript, Console, and Output. The JavaScript tab contains the following code:

```
JavaScript ▾
var matrix = [
  [ 4, 5, 6],
  [ 9, 8, 2],
  [10, 3, 25]
];
console.log(matrix);
```

The Console tab shows the output of the code:

```
[[4, 5, 6], [9, 8, 2], [10, 3, 25]]
```

Two-dimensional Array Example

You can see that we created an array with 3 elements. Each element is another array with 3 elements too.

## Reading and Writing Array Elements

The array elements are referenced using the [] operator. Inside the [] operator we need to have an expression that evaluates to a non-negative integer. This is because the array elements are referenced by their position. The positioning is 0-based. Hence, the first element of the array is at position 0, the second element of the array is at position 1 and so on.

An array can hold up to  $2^{32}$  elements. In other words, up to 4,294,967,296 elements. And since the positioning is 0-based, the index used to access an array may be one value from 0 up to `232 - 1`, a.k.a. up to 4,294,967,295.

Let's try some examples. Write the following JavaScript program on JS Bin:

```

1 var employees = [
2   {firstName: "George", lastName: "M"}, 
3   {firstName: "Mary", lastName: "L"}, 
4   {firstName: "John", lastName: "K"} 
5 ];
6
7 for (var i = 0; i < employees.length; i++) {
8   console.log("Employee (" + i + "): " + employees[i].firstName);
9 }

```

(the above code snippet online)

If you run the above program, you will see this:

```

HTML CSS JavaScript Console Output
JavaScript ▾
var employees = [
  {firstName: "George", lastName: "M"}, 
  {firstName: "Mary", lastName: "L"}, 
  {firstName: "John", lastName: "K"} 
];
for (var i = 0; i < employees.length; i++) {
  console.log("Employee (" + i + "): " + employees[i].firstName);
}

```

Console

```

"Employee (0): George"
"Employee (1): Mary"
"Employee (2): John"

```

Example Accessing Array Elements

In the above example, we have an array of employees, i.e. array with 3 elements. The 3 elements are objects, created with object literals. All the objects have the same properties: `firstName` and `lastName`. The program uses a `for` loop to iterate through all the elements of the array and print the first names of the employees.

See how the `employees[i]` is accessing the `i`th element of the array, which is an object. And then, with the dot notation, we are accessing the property `firstName`. Note also that the first element of the array is accessed with `0` index and the last element with `employees.length - 1`. The `i` goes up to `employees.length - 1` (since the repeat condition is `i < employees.length` and `i` is increased by 1 on every iteration - `i++`).

In the following example, we are updating the first name and last name of the 2nd employee.

```

1 var employees = [
2   {firstName: "George", lastName: "M"}, 
3   {firstName: "Mary", lastName: "L"}, 
4   {firstName: "John", lastName: "K"} 
5 ];
6
7 employees[1].firstName = "Tom";
8 employees[1].lastName = "J";
9
10 console.log(employees);

```

(the above code snippet online)

If you run this program on JS Bin, you will get this:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the following code:

```
var employees = [
  {firstName: "George", lastName: "M"},
  {firstName: "Mary", lastName: "L"},
  {firstName: "John", lastName: "K"}
];

employees[1].firstName = "Tom";
employees[1].lastName = "J";

console.log(employees);
```

The Console tab shows the output of the code execution:

```
[{"firstName": "George", "lastName": "M"}, {"firstName": "Tom", "lastName": "J"}, {"firstName": "John", "lastName": "K"}]
```

Updating the Element of An Array

## Adding And Deleting Elements

We can add elements to an array, by just setting values to new indexes. Let's see the following example:

```
1 var products = [];
2 products[0] = "Apples";
3 console.log(products.length);
4
5 products[1] = "Oranges";
6 console.log(products.length);
7
8 products[2] = "Grapes";
9 console.log(products.length);
```

(the above code snippet online)

If you run the above program on JS Bin, you will see this:

The screenshot shows a browser-based developer tool interface. At the top, there are three tabs: 'JavaScript' (which is active), 'Console', and 'Output'. In the 'JavaScript' tab, the following code is written:

```
var products = [];
products[0] = "Apples";
console.log(products.length);

products[1] = "Oranges";
console.log(products.length);

products[2] = "Grapes";
console.log(products.length);
```

To the right, under the 'Console' tab, the output is displayed in three numbered lines:

- 1
- 2
- 3

A blue arrow points from the code in the JavaScript tab to the first number in the Console tab.

### Adding Elements To An Array

Sometimes, you do not know the length of the array and you want to add a new element at the end of it. Look at the following example:

```
1 var products = [];
2 products[products.length] = "Apples";
3 console.log(products.length);
4
5 products[products.length] = "Oranges";
6 console.log(products.length);
7
8 products[products.length] = "Grapes";
9 console.log(products.length);
10
11 console.log(products);
```

(the above code snippet online)

If you run the above program on JS Bin, you will get this:

The screenshot shows a browser-based developer tool interface. At the top, there are tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is selected, showing the following code:

```
1 var products = [];
2 products[products.length] = "Apples";
3 console.log(products.length);
4
5 products[products.length] = "Oranges";
6 console.log(products.length);
7
8 products[products.length] = "Grapes";
9 console.log(products.length);
10
11 console.log(products);
```

To the right, under the 'Console' tab, the output is displayed in three rows:

1	
2	
3	

Below the table, the output is shown as a single line: `["Apples", "Oranges", "Grapes"]`.

### Adding Elements At The End Of The Array

As you can see above, referencing the element position which is equal to the length of the array, we can add a new element at the end of the array. This is because the last element exists at position `<array>.length - 1`, and a new element can be added after that, i.e. at position `<array>.length`. Then, the `length` is automatically updated to reflect the new length of the array at hand.

#### `push()`

Another alternative to add new elements to an array, is the use of method `push()`. The method `push()` takes as input arguments the elements to be added at the end of the array.

```
1 var products = [];
2 products.push("Apples");
3 console.log(products.length);
4 console.log(products);
5
6 products.push("Oranges", "Grapes");
7 console.log(products.length);
8 console.log(products);
```

(the above code snippet online)

If you run the above program on JS Bin, you will get this:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the following code:

```
1 var products = [];
2 products.push("Apples");
3 console.log(products.length);
4 console.log(products);

5 products.push("Oranges", "Grapes");
6 console.log(products.length);
7 console.log(products);
```

The Console tab shows the output of the code:

Console
1
["Apples"]
3
["Apples", "Oranges", "Grapes"]

### Using push() to Add Elements To an Array

As you can see the `products.push("Oranges", "Grapes")`; added 2 elements at the end of the array.

### pop()

The `pop()` method is used to remove the last element of an array. Also, it returns the element removed. Let's try that:

```
1 var products = [];
2 products.push("Apples");
3 console.log(products.length);
4 console.log(products);

5
6 products.push("Oranges", "Grapes");
7 console.log(products.length);
8 console.log(products);

9
10 var element = products.pop();
11 console.log(element);
12 console.log(products.length);
13 console.log(products);
```

(the above code snippet online)

If you run the above program on JS Bin, you will get this:

The screenshot shows a browser-based developer tool interface. At the top, there are tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is active, showing the following code:

```
JavaScript ▾
var products = [];
products.push("Apples");
console.log(products.length);
console.log(products);

products.push("Oranges", "Grapes");
console.log(products.length);
console.log(products);

var element = products.pop();
console.log(element);
console.log(products.length);
console.log(products);
```

To the right, under the 'Console' tab, the output is displayed in a list:

- 1 ["Apples"]
- 3 ["Apples", "Oranges", "Grapes"]
- "Grapes"
- 2 ["Apples", "Oranges"]

A blue arrow points from the code area to the first item in the console list.

#### Using pop() to Delete Elements of an Array

As you can see, the line `var element = products.pop();` removes the last element of the array and, also, since `pop()` returns the value of the element deleted, that last element value is saved inside the variable `element`. Note also that the `length` property value is automatically updated.

#### shift()

There is another way you can remove an element from an array. It is the `shift()` method. It works like `pop()` but it deletes the first element of the array.

```
1 var products = ["Apples", "Oranges", "Grapes"];
2 var element = products.shift();
3 console.log(element);
4 console.log(products.length);
5 console.log(products);
```

(the above code snippet online)

If you run the above program on JS Bin, you will get this:

The screenshot shows a browser-based developer tool interface. At the top, there are tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is active, showing the following code:

```
JavaScript ▾
var products = ["Apples", "Oranges", "Grapes"];
var element = products.shift();
console.log(element);
console.log(products.length);
console.log(products);
```

To the right, under the 'Console' tab, the output is displayed in a list:

- "Apples"
- 2 ["Oranges", "Grapes"]

A blue arrow points from the code area to the first item in the console list.

#### Using shift() to Delete First Element of the Array

Pretty easy, isn't it?

### splice()

This is a more generic method that does both addition and deletion of elements from an array. Also, when it removes elements, it returns the elements removed.

It's syntax is as follows:

```
1 array.splice(index, how_many, item1, ... , itemX)
```

(the above code snippet online)

The parameters that can be sent to `splice()` are:

1. `index`. *Required*. The position of the array that the elements will be added to, or removed from. You can use a negative integer if you want to refer to elements at the end of the array.
2. `how_many`. *Required*. The number of items to be removed. If `0`, nothing will be removed.
3. `item1, ... , itemX`. *Optional*. The new items to be added to the array.

Let's see a first example.

```
1 var products = ["Apples", "Oranges", "Grapes"];
2 var element = products.splice(1, 1);
3 console.log(element);
4
5 console.log(products.length);
6 console.log(products);
```

(the above code snippet online)

If you run the above program on JS Bin, you will get this:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the provided code. The Console tab shows the output: an array with one element ("Oranges"), the number 2 (indicating two elements were removed), and the original array ["Apples", "Grapes"].

```
JavaScript
var products = ["Apples", "Oranges", "Grapes"];
var element = products.splice(1, 1);
console.log(element);

console.log(products.length);
console.log(products);

Console
["Oranges"]
2
["Apples", "Grapes"]
```

### Using splice() To Remove An Element

In the above example, we are using `splice()` to remove the 2nd element. That is why the first argument given is `1` and the second one is `1` too. Notice how the result of `splice` is an array too. Hence, the `element`, when printed, it gives: `["Oranges"]`, an array with 1 element.

Let's see now a more complicated example:

```

1 var products = ["Apples", "Oranges", "Grapes"];
2 var elements = products.splice(1, 2, "Tomatoes", "Potatoes", "Cucumbers");
3 console.log(elements);
4
5 console.log(products.length);
6 console.log(products);

```

(the above code snippet online)

If you run the above program on JS Bin, you will get this:

```

HTML CSS JavaScript Console Output
JavaScript
var products = ["Apples", "Oranges", "Grapes"];
var elements = products.splice(1, 2, "Tomatoes", "Potatoes", "Cucumbers");
console.log(elements);

console.log(products.length);
console.log(products);

```

```

Console
["Oranges", "Grapes"]
4
["Apples", "Tomatoes", "Potatoes", "Cucumbers"]

```

#### A More Complicated Example with splice()

The previous example works on the 2nd position of the original array. It removes 2 items starting from position 1 and then adds 3 new items. See how `splice()` returns an array with the two elements removed.

Before closing with slice, try the following example:

```

1 var products = ["Apples",
2                 "Oranges",
3                 "Grapes",
4                 "Tomatoes"];
5 var elements = products.splice(1, 2, "Potatoes", "Cucumbers");
6 console.log(elements);
7
8 console.log(products.length);
9 console.log(products);

```

(the above code snippet online)

If you run the above program on JS Bin, you will see this:

```

HTML CSS JavaScript Console Output
JavaScript
var products = ["Apples",
                "Oranges",
                "Grapes",
                "Tomatoes"];
var elements = products.splice(1, 2, "Potatoes", "Cucumbers");
console.log(elements);

console.log(products.length);
console.log(products);

```

```

Console
["Oranges", "Grapes"]
4
["Apples", "Potatoes", "Cucumbers", "Tomatoes"]

```

#### Another Slice Example

Notice how the two new items added, "Potatoes", "Cucumbers" are added at position 2, replacing "Oranges" and "Grapes".

## Array Methods

We will now continue our introduction to arrays, by learning some of the most useful methods that can be invoked on an array object.

### `join()`

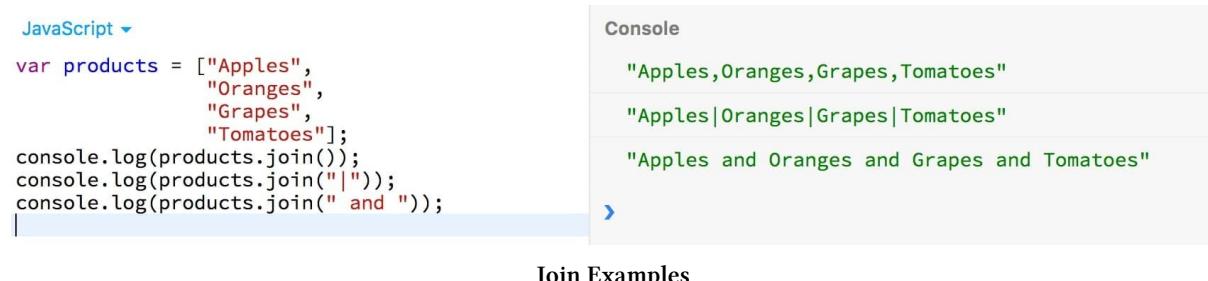
The `Array.join()` method converts all the elements of an array to strings and concatenates them, returning the resulting string.

Let's try some examples:

```
1 var products = ["Apples",
                  "Oranges",
                  "Grapes",
                  "Tomatoes"];
2 console.log(products.join());
3 console.log(products.join(" | "));
4 console.log(products.join(" and "));
```

(the above code snippet online)

If you run the above on JS Bin, you will get this:



The screenshot shows a JS Bin interface. On the left, there is a code editor with the following JavaScript code:

```
JavaScript ▾
var products = ["Apples",
                 "Oranges",
                 "Grapes",
                 "Tomatoes"];
console.log(products.join());
console.log(products.join(" | "));
console.log(products.join(" and "));
```

On the right, there is a 'Console' output window with three lines of text:

- "Apples,Oranges,Grapes,Tomatoes"
- "Apples|Oranges|Grapes|Tomatoes"
- "Apples and Oranges and Grapes and Tomatoes"

A blue arrow points from the bottom of the code editor towards the console output.

### Join Examples

As you can see, the first `join`, `products.join()` joins the elements using the `,` separator. The second `join`, `products.join(" | ")` joins the elements using the `|` separator. And, finally, the third `join`, `products.join(" and ")` joins the elements using the `" and "` string separator. The result of `join()`, in any case, is a new string.

### `reverse()`

The `Array.reverse()` method reverses the order of the elements of an array and returns the reversed array.

Let's try an example:

```
1 var products = ["Apples",
2                 "Oranges",
3                 "Grapes",
4                 "Tomatoes"];
5 console.log(products.reverse());
6 console.log(products);
```

(the above code snippet online)

If you run the above on JS Bin, you will see the following:

The screenshot shows a JS Bin interface with tabs for CSS, JavaScript, Console, and Output. The JavaScript tab contains the provided code snippet. The Console tab shows two log outputs: the first is the reversed array `["Tomatoes", "Grapes", "Oranges", "Apples"]` and the second is the original array `["Tomatoes", "Grapes", "Oranges", "Apples"]` again, indicating that `reverse()` alters the original array.

Example of reverse()

As you can see above, the `reverse()` method alters the original array. Does the job in place, as we say.

### sort()

The method `sort` is used to sort the elements of an array.

Let's try an example:

```
1 var products = ["Apples",
2                  "Tomatoes",
3                  "Grapes",
4                  "Oranges"];
5 console.log(products.sort());
6 console.log(products);
```

(the above code snippet online)

If you run the above program on JS Bin, you will get this:

The screenshot shows a browser's developer tools interface with four tabs at the top: CSS, JavaScript, Console, and Output. The JavaScript tab is selected. In the main area, there is a code editor window containing the following JavaScript code:

```
1 var products = ["Apples",
2                 "Tomatoes",
3                 "Grapes",
4                 "Oranges"];
5 console.log(products.sort());
6 console.log(products);
```

Below the code editor is a scroll bar. To the right of the code editor is a panel titled "Console" which displays the output of the code:

```
["Apples", "Grapes", "Oranges", "Tomatoes"]
["Apples", "Grapes", "Oranges", "Tomatoes"]
```

A blue arrow icon is located between the two lines of output.

A Simple sort Example

`sort()` always sorts in ascending alphabetical order. What if you want to sort the list in descending alphabetical order? In that case, you need to give an argument as comparison function. The comparison function tells `sort()` the relative sorting order of two elements. Given an element `a` and its next element `b`, if the comparison function returns something less than 0, `a` is considered to be first and `b` is considered to be second in ordering. If the comparison function returns something greater than 0, a positive number, then `a` is considered to be after `b` in the sorting order. If the comparison function returns 0, then both elements are considered to be on same sorting position.

Having said the above, try the following example:

```
1 var products = ["Apples",
2                 "Tomatoes",
3                 "Grapes",
4                 "Oranges"];
5
6 products.sort(function(a, b) {
7     if (a < b) {
8         return 1;
9     }
10    else if (a > b) {
11        return -1;
12    }
13    else return 0;
14 });
15
16 console.log(products);
```

(the above code snippet online)

This piece of code above, sorts the elements of the array `products`, using a comparison function. This comparison function compares an element `a` to its neighbouring element `b`. Hence, `a` and `b` represent two adjacent elements in the array. The comparison logic is such that if the left element `a` is less than right element `b`, the function returns 1 signifying that `a` should actually be considered greater than `b`. Also, if left element `a` is greater than right element `b`, the function returns -1 signifying that `a` should actually be considered less than `b`. These two branches of

comparison logic, for the two adjacent elements, make sure that the array is sorted in descending order.

Try to run the above program on JS Bin and you will get this:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the following code:

```

1  var products = ["Apples",
2                  "Tomatoes",
3                  "Grapes",
4                  "Oranges"];
5
6  products.sort(function(a, b) {
7    if (a < b) {
8      return 1;
9    }
10   else if (a > b) {
11     return -1;
12   }
13   else return 0;
14 });
15
16  console.log(products);

```

The Console tab shows the output: `["Tomatoes", "Oranges", "Grapes", "Apples"]`.

#### Sorting In Descending Order

Of course, this can also be written using the ternary operator:

```

1  var products = ["Apples",
2                  "Tomatoes",
3                  "Grapes",
4                  "Oranges"];
5
6  products.sort(function(a, b) {
7    return (a < b ? 1 : (a > b ? -1 : 0));
8 });
9
10 console.log(products);

```

[\(the above code snippet online\)](#)

If you run the above program on JS Bin, you will get the same result as before.

*Note:* Do you see how convenient is to use unnamed functions? Since the comparison functions are used only once, there is no need to give them a name.

#### **concat()**

The `Array.concat()` method creates and returns a new array that contains the elements of the original array on which `concat()` was invoked, followed by each of the arguments to `concat()`. If any of these arguments is itself an array, then it is the array elements that are concatenated, not the array itself. Note, however, that `concat()` does not recursively flatten arrays of arrays.

Finally, `concat()` does not modify the array on which it is invoked.

Let's try the following example:

```

1 var products = ["Apples",
2                 "Tomatoes",
3                 "Grapes",
4                 "Oranges"];
5
6 console.log(products.concat(["Endives", "Brocolli", "Celeries"]));
7
8 console.log(products);

```

(the above code snippet online)

If you run the above program on JS Bin, you will get this:

The screenshot shows a JS Bin interface with tabs for TML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the provided code. The Console tab shows the output: the original array 'products' is printed first, followed by the concatenated array, and finally the original array again, demonstrating that `concat()` does not modify the original array.

```

TML CSS JavaScript Console Output Account Blog
JavaScript ▾
var products = ["Apples",
                 "Tomatoes",
                 "Grapes",
                 "Oranges"];
console.log(products.concat(["Endives", "Brocolli", "Celeries"]));
console.log(products);

Console
["Apples", "Tomatoes", "Grapes", "Oranges",
 "Endives", "Brocolli", "Celeries"]
["Apples", "Tomatoes", "Grapes", "Oranges"]
>

```

Example of `concat()`

As you can see above, the `concat()` method does not modify the original array.

### `slice()`

The `Array.slice()` method returns a slice, or sub-array, of the specified array. Its two arguments specify the start and end of the slice to be returned. The returned array contains the element specified by the first argument and all subsequent elements up to, but not including, the element specified by the second argument. If only one argument is specified, the returned array contains all elements from the start position to the end of the array. If either argument is negative, it specifies an array element relative to the last element in the array. An argument of `-1`, for example, specifies the last element in the array, and an argument of `-3` specifies the third from last element of the array. Note that `slice()` does not modify the array on which it is invoked.

Let's try the following example:

```

1 var products = ["Apples",
2                 "Tomatoes",
3                 "Grapes",
4                 "Oranges"];
5
6 console.log(products.slice(1, -1));
7
8 console.log(products);

```

(the above code snippet online)

This program is using `slice()` to get the 2nd up to the last element of the array, without including the last element.

If you run the above program on JS Bin, you will get this:

The screenshot shows a JS Bin interface. On the left, under 'JavaScript', the code is displayed:

```

var products = ["Apples",
    "Tomatoes",
    "Grapes",
    "Oranges"];
console.log(products.slice(1, -1));
console.log(products);

```

On the right, under 'Console', the output is shown in two lines:

```

["Tomatoes", "Grapes"]
["Apples", "Tomatoes", "Grapes", "Oranges"]

```

A `slice()` Example

As you can see from the example above, the `slice()` method does not modify the original array.

### **unshift() and shift()**

These methods work like the `push()` and `pop()` which we have already seen. The `unshift()` adds an element at the beginning of the array. The `shift()` removes the first element of the array. Both, they update the array in place and they update the length property accordingly.

```

1 var products = ["Apples",
2                 "Tomatoes",
3                 "Grapes",
4                 "Oranges"];
5
6 products.unshift("Celery");
7 console.log(products);
8
9 var element = products.shift();
10 console.log(element);
11 console.log(products);

```

(the above code snippet online)

The above program first adds the "Celery" element at the beginning of the array, using the `unshift()` method. Then removes the first element using the `shift()` method.

If you run the above program on JS Bin, you will get this:

```
HTML CSS JavaScript Console Output
JavaScript ▾
var products = ["Apples",
                 "Tomatoes",
                 "Grapes",
                 "Oranges"];
products.unshift("Celery");
console.log(products);
var element = products.shift();
console.log(element);
console.log(products);
```

Console

```
["Celery", "Apples", "Tomatoes", "Grapes", "Oranges"]
"Celery"
["Apples", "Tomatoes", "Grapes", "Oranges"]
```

Example of `unshift()` and `shift()`

### forEach()

The `Array.forEach()` method can be used to carry out an action for each one of the elements of an array, iteratively. It takes as argument a function which can have 3 arguments. The first argument is the value of the element at each iteration. The second argument is the position index of the element at hand. The third argument is a reference to the array itself, and it is used when we want to alter the elements of the array.

Let's suppose that we want to sum the elements of an array of integers:

```
1 var salaries = [1000, 2000, 1500, 2500, 3500, 4000, 3800];
2 var sum = 0;
3 salaries.forEach(function(element, i, data){
4     sum += element;
5 });
6
7 console.log(sum);
```

(the above code snippet online)

If you run the above on JS Bin, you will see this:

The screenshot shows a browser-based developer tool interface. At the top, there are tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is selected, showing the following code:

```
1 var salaries = [1000, 2000, 1500, 2500, 3500, 4000, 3800];
2 var sum = 0;
3 salaries.forEach(function(element, i, data){
4     sum += element;
5 });
6
7 console.log(sum);
```

To the right, under the "Console" tab, the output is displayed as:

```
18300
```

### forEach Example to Sum

As you can see, the function given to `forEach()` does a very simple increment of the `sum` variable, by the amount of the `element` at hand (`sum += element`). Since, this function is called for each one of the elements of the array, at the end, the `sum` contains the sum of the elements of the array of integers.

Of course, `sum` can be calculated using a `for` loop:

```
1 var salaries = [1000, 2000, 1500, 2500, 3500, 4000, 3800];
2 var sum = 0;
3 for (var i = 0; i < salaries.length; i++) {
4     sum += salaries[i];
5 }
6
7 console.log(sum);
```

(the above code snippet online)

Note that `for` version is considered to be faster. Also, `for` statement has the capability to prematurely end the loop using `break`, something that cannot be accomplished with `forEach()`.

Another example of `forEach()` converts the strings of an array to upper case:

```
1 var products = ["Apples", "Grapes", "Tomatoes", "Potatoes"];
2 products.forEach(function(element, i, array){
3     array[i] = array[i].toUpperCase();
4 });
5
6 console.log(products);
```

(the above code snippet online)

If you run the above program on JS Bin, you will get this:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the following code:

```
JavaScript ▾
var products = ["Apples", "Grapes", "Tomatoes", "Potatoes"];
products.forEach(function(element, i, array){
  array[i] = array[i].toUpperCase();
});
console.log(products);
```

The Console tab shows the output:

```
["APPLES", "GRAPES", "TOMATOES", "POTATOES"]
```

forEach() to Update Original Array

### map()

The `Array.map()` method is used to carry out an action on the elements of an array. It takes as input argument the conversion function. The conversion function needs to return the element given converted. Note that the `map()` function does not convert the input elements in place. It returns a completely new array of elements.

Let's try again the example with converting all elements to upper case.

```
1 var products = ["Apples", "Grapes", "Tomatoes", "Potatoes"];
2
3 var upperCaseProducts = products.map(function(element) {
4   return element.toUpperCase();
5 });
6
7 console.log(products);
8
9 console.log(upperCaseProducts);
```

(the above code snippet online)

If you run the above program on JS Bin, you will see this:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the following code:

```
JavaScript ▾
var products = ["Apples", "Grapes", "Tomatoes", "Potatoes"];
var upperCaseProducts = products.map(function(element) {
  return element.toUpperCase();
});
console.log(products);
console.log(upperCaseProducts);
```

The Console tab shows the output:

```
["Apples", "Grapes", "Tomatoes", "Potatoes"]
["APPLES", "GRAPES", "TOMATOES", "POTATOES"]
```

A map() Example

As you can see above, the `map()` uses a conversion function that takes as input an element of the array, converts it to upper case and returns it. `map()` conversion function is applied on all elements of the array. Also, the `map()` does not modify the original array. It returns a brand new array.

### filter()

The `Array.filter()` method is used to create a sub-array from the given array. The sub-array contains all the elements for which the test function given to `filter()` returns true.

Let's try the following example, which selects the even numbers from an array with both odd and even numbers.

```
1 var arrayOfNumbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
2
3 var evenNumbers = arrayOfNumbers.filter(function(element) {
4     return element % 2 === 0;
5 });
6
7 console.log(arrayOfNumbers);
8 console.log(evenNumbers);
```

(the above code snippet online)

If you run the above on JS Bin, you will get this:

JavaScript `var arrayOfNumbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]; var evenNumbers = arrayOfNumbers.filter(function(element) { return element % 2 === 0; }); console.log(arrayOfNumbers); console.log(evenNumbers);`

Console  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
[2, 4, 6, 8, 10]

### An Example of filter()

As you can see above, the array `evenNumbers` has been constructed by the application of `filter()` onto the original array `arrayOfNumbers`. Only the elements that were making test function return true have been added to `evenNumbers`.

### every()

The `Array.every()` method returns either `true` or `false`. It returns `true` when all elements of the array satisfy the test function that is given as input argument to `every()`. Otherwise, it returns `false`.

Let's try that with two arrays, one with numbers and one with even numbers. The test function that we will give will test whether the given element is even number.

```
1 var arrayOfNumbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
2
3 var evenNumbers = [2, 4, 6, 8, 10];
4
5 console.log(arrayOfNumbers.every(function(element) {
6     return element % 2 === 0;
7 }));
8
9 console.log(evenNumbers.every(function(element) {
10    return element % 2 === 0;
11 }));
```

(the above code snippet online)

If you run the above program on JS Bin, you will get this:

HTML	CSS	JavaScript	Console	Output
JavaScript ▾		<pre>var arrayOfNumbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]; var evenNumbers = [2, 4, 6, 8, 10]; console.log(arrayOfNumbers.every(function(element) {     return element % 2 === 0; })); console.log(evenNumbers.every(function(element) {     return element % 2 === 0; }));</pre>	Console	false true

#### Example of every

As you can see above, the program prints `false` for `arrayOfNumbers` and `true` for `evenNumbers`. This is because there are elements inside `arrayOfNumbers` that do not satisfy the test function. These elements make test function return `false`. On the other hand, all elements of `evenNumbers` satisfy the test function, i.e. they make it return `true`.

BTW, there is a code repetition on the above piece of JavaScript code. This code repetition can be avoided by saving the function definition into a variable and then use that variable as input argument to `every()`. Let's see that:

```
1 var arrayOfNumbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
2
3 var evenNumbers = [2, 4, 6, 8, 10];
4
5 var isEvenNumber = function(element) {
6     return element % 2 === 0;
7 }
8
9 console.log(arrayOfNumbers.every(isEvenNumber));
10
11 console.log(evenNumbers.every(isEvenNumber));
```

(the above code snippet online)

If you run the above on JS Bin, you will, again, get false and then true.

### some()

The `Array.some()` method is similar to `Array.every()` but does not require all elements to satisfy the test function. If at least one element satisfies the test function then this method returns true.

Let's try the following example:

```
1 var numbers = [1, 2, 3, 5, 7, 9];
2
3 var isEvenNumber = function(element) {
4     return element % 2 === 0;
5 }
6
7 console.log(numbers.some(isEvenNumber));
8
9 console.log(numbers.every(isEvenNumber));
```

(the above code snippet online)

If you run the above program on JS Bin, you will get this:

```

    JavaScript ▾
    var numbers = [1, 2, 3, 5, 7, 9];
    var isEvenNumber = function(element) {
      return element % 2 === 0;
    };
    console.log(numbers.some(isEvenNumber));
    console.log(numbers.every(isEvenNumber));
  
```

Console
true
false

### Some vs Every Example

The above example depicts the difference between `every()` and `some()`. As you can see, the `some()` returns `true`, because there is one even number in the given array of numbers. Whereas the `every()` returns `false`, because not all numbers are even.

### `reduce()`

The `Array.reduce()` method parses all the elements of an array and returns back one value only. That's why it is called `reduce`. Because it reduces many values (the array) to a single value.

It is better explained with some examples:

```

1 var a = [100, 150, 200, 300, 350];
2 var sum = a.reduce(function(accumulator, element) {
3   return accumulator + element;
4 }, 0);
5
6 console.log(sum);
  
```

(the above code snippet online)

The `reduce()` method above is called with 2 arguments:

1. The first argument to `reduce()` is the function that operates on an element. This takes two arguments too:
  1. An `accumulator` argument which is initialized with the value of the 2nd argument sent to `reduce()` method. In our example `accumulator` is initialized with `0`. At the end of each iteration of `reduce()`, the `accumulator` holds the function returned value at the particular iteration.
  2. The second argument to the function is the element at hand. We said that `reduce()` works by parsing all the elements one-by-one. The `element` parameter holds the value of each element.
2. The second argument to `reduce()` is the starting value of the `accumulator`.

In other words...

1. At the 1st iteration, function is called with `0` and `100`, adding them together, returning `100` and saving that to `accumulator`.
2. At the 2nd iteration, function is called with `100` and `150`, adding them together, returning `250` and saving that to `accumulator`.
3. At the 3rd iteration, function is called with `250` and `200`, adding them together, returning `450` and saving that to `accumulator`.
4. At the 4th iteration, function is called with `450` and `300`, adding them together, returning `750` and saving that to `accumulator`.
5. At the 5th iteration, function is called with `750` and `350`, adding them together, returning `1100` and saving that to `accumulator`.

*Note:* You will often come across other names for the `reduce` feature in other languages, like `inject` and `fold`.

### **indexOf() and lastIndexOf()**

The `Array.indexOf()` searches an array for a matching element, which is given as input argument. It searches from start till the end. If the element is found, then it returns its position. If the element is not found, then it returns `-1`. The `lastIndexOf()` searches from the end of the array.

Let's try an example:

```
1 var numbers = [1, 2, 3, 5, 7, 9];
2
3 var position = numbers.indexOf(3);
4
5 console.log(position);
```

(the above code snippet online)

If you run the above program on JS Bin, you will see the number `2` being printed. This is because the number `3` is at 3rd position in the array, which corresponds to index `2` (since array positions are indexed starting from `0`).

Let's expand that example:

```
1 var numbers = [1, 2, 3, 5, 7, 9, 7, 3, 10];
2
3 var position = numbers.indexOf(3);
4
5 console.log(position);
6
7 position = numbers.lastIndexOf(3);
8
9 console.log(position);
10
11 position = numbers.indexOf(11);
12
13 console.log(position);
```

(the above code snippet online)

If you run the above on JS Bin, you will get this:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the provided code. The Console tab shows the following output:

Console
2
7
-1

A blue arrow points from the last row of the console output to the right.

Examples of indexOf() method

You can see in the above example, how `lastIndexOf()` returns the position of 3 that is closest to the end of the array. Also, you can see the returned position being `-1` for elements `(11)` that do not exist in the array.

Please, note that the comparison to locate the element inside the array, is done using strict comparison (`==`) operator.

## Strings as Arrays

JavaScript strings can be treated like arrays up to an extent. For example, you can access individual characters of a string using the square brackets operator and the index of the position that you want to access.

For example:

```
1 var greeting = "Hello World";
2
3 console.log(greeting[1]);
```

(the above code snippet online)

The above program prints out e. With the `greeting[1]`, we are accessing the 2nd character in the string given.

With strings, you can use the `indexOf()` method to get the position of a character inside a string:

For example:

```
1 var greeting = "Hello World";
2
3 console.log(greeting.indexOf("l"));
```

(the above code snippet online)

The above program will print 2, which is the index of the character l inside the `greeting` string.

## Tasks and Quizzes

**Before you continue, you may want to know that:** You can sign up to [Tech Career Booster](#) and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

### Tasks

(1) Write a program that sorts an array in case insensitive alphabetical order.

**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

(2) Write a program that demonstrates the matrix addition. Assume that you have two 2-dimensional matrices, 4 x 4. Read about [matrix addition here](#). Here are some hints to help you write this program:

1. Declare a variable A to be an array of arrays. Declare all the values of all the elements. An array literal.
2. Declare a variable B in a similar way.
3. Declare a variable C to be an empty array.
4. In order to access the element on row 2 and column 3 on table A, you write something like `A[1][2]`.
5. You will need a double nested loop construct, i.e. a `for` inside a `for`. The external `for` goes from one row to the next. The internal `for` goes from one column to the next.

6. Before starting the column internal for loop, you will need to initially the corresponding C row to an empty array [].

**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

(3) Given the array [5, 8, 2, 15, 32, 3, 46] calculate the sum of the array converted elements, when conversion is according to the formula element \* 2 + 15. In other words, write a JavaScript program that calculates the sum of  $5 * 2 + 15$ ,  $8 * 2 + 15$ ,  $2 * 2 + 15$ , ...,  $46 * 2 + 15$ . Create 2 versions of the program. One that is using `map()` and `reduce()`. And another one that does not use these two methods.

**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

## 8 - Functions

### Summary

Functions are one very important code reuse tool in every programming language. You write a piece of programming logic once, and then you can reuse it multiple times in your program.

You will learn the various ways you can use to define a function and then invoke it.

The screenshot shows a browser's developer tools interface with the 'Console' tab selected. At the top, there are tabs for 'HTML', 'CSS', 'JavaScript', 'Console', and 'Output'. The 'JavaScript' tab is active, indicated by a dropdown arrow icon. Below the tabs, there is a code editor area containing the following JavaScript code:

```
JavaScript ▾
function printProperties(object) {
  for (var prop in object) {
    console.log(prop + " has value: " + object[prop]);
  }
}

var customer = {
  firstName: "Panos",
  lastName: "M"
};

var order = {
  invoiceDate: "2015-07-16",
  invoiceNumber: "ABC1234"
};

printProperties(customer);
printProperties(order);
```

To the right of the code editor is the 'Console' output area, which displays the following log entries:

```
Console
"firstName has value: Panos"
"lastName has value: M"
"invoiceDate has value: 2015-07-16"
"invoiceNumber has value: ABC1234"
```

A blue arrow icon points from the code editor towards the console output.

Definition and Invocation of a Function

You will also learn how to define

- mandatory parameters
- optional parameters
- and variable number of parameters

The screenshot shows a browser's developer tools interface with the 'Console' tab selected. On the left, there is a code editor window containing a JavaScript function named 'maximum'. This function takes an array of arguments, iterates through them to find the maximum value, and returns it. Below the function definition, several calls are made to 'maximum' with different argument lists, and the results are printed to the console.

```

function maximum() {
  if (arguments.length >= 1) {
    var result = arguments[0];
    for (var i = 1; i < arguments.length; i++) {
      if (arguments[i] > result) {
        result = arguments[i];
      }
    }
    return result;
  }
}

console.log(maximum());
console.log(maximum(1));
console.log(maximum(3, 2, 5));
console.log(maximum(3, 5, 17, 4, 2, 8, 5, 3, 7, 8, 9, 10, 0, 12));
|
```

Console
undefined
1
5
17
>

#### Example Of A Function With Optional Arguments

You will also learn how to pass many arguments wrapping them into an object.

Next, you will learn about recursive functions and how they can be handy to write clean code.

Finally, you will learn how to

- throw errors
- catch errors, and
- throw custom errors like these:

The screenshot shows a browser's developer tools interface with the 'Console' tab selected. In the console, a custom error is thrown using the 'Error' constructor. The error message is 'error'. When the error is thrown, a detailed stack trace is printed to the console, showing the call stack from the point of the error being thrown back to the global context.

```

"error"
"Error
    at new ArgumentError (keribiwizi.js:4:19)
    at dayOfTheWeekInteger (keribiwizi.js:26:13)
    at keribiwizi.js:31:1
    at https://static.jsbin.com/js/prod/runner-3.38.13.min.js:1:13926
    at https://static.jsbin.com/js/prod/runner-3.38.13.min.js:1:10855"
|
```

#### Throwing Custom Errors

## Learning Goals

1. Learn about how you can define a function.

2. Learn how you invoke a function.
3. Learn about the function parameters and arguments.
4. Learn how you can define a function and call it immediately, at the same time.
5. Learn how the return statement works inside a function.
6. Learn more about how a function is invoked:
  1. Function Invocation
  2. Method Invocation
  3. Constructor Invocation
7. Learn more about Function Arguments and Parameters
  1. Learn about mandatory and optional parameters
  2. Learn about the functions that take variable number of parameters.
  3. Learn to use objects as arguments.
8. Learn about the recursive functions.
9. Learn about throwing errors.
10. Learn about catching errors.
11. Learn about throwing custom errors.

## Introduction

Functions are programming constructs that can group a set of programming instructions under a name. Then, we can call the functions multiple times. Every time we call a function, the instructions that are part of the function are executed. Functions may be parameterized. This means that, when defined, they include a list of identifiers, which are called *parameters*, and which work like local variables, i.e. their scope is limited within the function block definition. Also, the parameters are used when we invoke a function. For each one of the parameters, we pass an actual value, which is called an argument.

## Defining Functions

Let's see in more detail how we can define a function. Look at the following example:

```
1 function printProperties(object) {  
2     for (var prop in object) {  
3         console.log(prop + " has value: " + object[prop]);  
4     }  
5 }
```

(the above code snippet online)

This is a function definition. The function will have the name `printProperties`. When you name a function like that, you essentially create a variable with name `printProperties` that you can later on use to invoke the function.

If you run the above program on JS Bin, you will see nothing happened. This is because the program only defines the function, but does not call it, does not invoke it.

Let's expand the above program to also invoke the function.

```

1 function printProperties(object) {
2   for (var prop in object) {
3     console.log(prop + " has value: " + object[prop]);
4   }
5 }
6
7 var customer = {
8   firstName: "Panos",
9   lastName: "M"
10};
11
12 var order = {
13   invoiceDate: "2015-07-16",
14   invoiceNumber: "ABC1234"
15};
16
17 printProperties(customer);
18 printProperties(order);

```

(the above code snippet online)

This program, besides defining the function `printProperties`, it also invokes it, twice. The invocations are the ones at the last 2 lines. If you run the above program on JS Bin, you will get this:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is active, displaying the code. The Console tab shows the output of the code execution.

```

JavaScript ▾
function printProperties(object) {
  for (var prop in object) {
    console.log(prop + " has value: " + object[prop]);
  }
}

var customer = {
  firstName: "Panos",
  lastName: "M"
};

var order = {
  invoiceDate: "2015-07-16",
  invoiceNumber: "ABC1234"
};

printProperties(customer);
printProperties(order);

```

Console

```

"firstName has value: Panos"
"lastName has value: M"
"invoiceDate has value: 2015-07-16"
"invoiceNumber has value: ABC1234"

```

#### Definition and Invocation of a Function

The function takes one parameter. The name of the parameter is `object` and it is used as local variable inside the function body, in order to do the actual work. On last two lines of the above program, we actually invoke the previously defined function. The invocation requires that we pass actual values on the parameters placeholders. They are the arguments of invocation. The arguments on the above program are `customer`, on the first invocation and `order` on the second.

Note that functions do not have to have parameters. Look at the following example:

```
1 function currentDateAndTime() {  
2     var now = new Date();  
3     return now.toString();  
4 }  
5  
6 console.log(currentDateAndTime());
```

(the above code snippet online)

If you run the above program on JS Bin, you will get this:

The screenshot shows a browser-based developer tool interface. At the top, there are tabs for CSS, JavaScript, Console, and Output. The JavaScript tab is selected, showing the following code:

```
function currentDateAndTime() {  
    var now = new Date();  
    return now.toString();  
}  
  
console.log(currentDateAndTime());
```

To the right, under the Console tab, the output is displayed:

```
"Fri May 27 2016 08:11:25 GMT+0100 (BST)"
```

Function Without Parameters Example

The above program defines the function `currentDateAndTime()`. This function is defined without parameters. Also, on last line, we call that function without arguments, obviously. BTW, this program is calling two more functions that do not take arguments. The function `Date()` and the function `toString()`.

## Defining And Calling Function At the Same Time

There are cases in which you want to define a function and call it, immediately and once. In order to do that:

1. You define the function without any name. You do not need that.
2. You wrap the function definition inside parentheses.
3. Then you immediately use opening and closing parentheses, with potential arguments inside, if the function takes parameters.

Let's see an example:

```
1 var result = (function(){  
2  
3     return "Hello World";  
4  
5 })();  
6  
7 console.log(result);
```

(the above code snippet online)

This is a very simple example. If you run this on JS Bin, you will see the string "Hello World" printed on console. This is because the `result` variable holds the return value of the function that it is defined as `function() { return "Hello World"; }`. This function is wrapped into parentheses: `(function() { return "Hello World"; })` and then it is invoked:

```
(function() { return "Hello World"; })().
```

Look at another example for a function that takes parameters:

```
1 var result = (function(x) {  
2  
3     return x * 2;  
4  
5 })(4);  
6  
7 console.log(result);
```

(the above code snippet online)

If you run the above program on JS Bin, you will get the number 8. This is because the function is invoked with argument 4 and then returns its double (`return x * 2;`).

1. First the function is defined: `function(x) { return x * 2; }`
2. Then the function is wrapped inside parentheses: `(function(x) { return x * 2; })`
3. Then the function is invoked with argument 4: `(function(x) { return x * 2; })(4)`
4. The result of invocation is assigned to variable `result`.

## The return statement

When a `return` statement is encountered within a function body, then JavaScript interpreter executes whatever statement follows `return` (up to `,`), calculates the statement value, and then terminates the function block, passing to the calling piece of code (the code that actually invoked the function), the value of the statement calculated.

If a function does not have a `return` statement, then all the statements inside the function body are executed one by one and what is being returned to the calling code is `undefined`. `undefined` is also returned if function has `return` statement with no accompanying statement that would return a value.

Let's see the following example of code:

```

1  function doSomething() {
2      for (var i = 0; i < 100000; i++) {
3          if ( i === 10 ) {
4              return i;
5          }
6      }
7      return "Hello World";
8  }
9
10 var result = doSomething();
11
12 console.log(result);

```

(the above code snippet online)

If you run the above on JS Bin, you will get the number 10 printed on console. This is because the function `doSomething()` always terminates on line `return i` when the `i` is equal to 10.

## Nested Functions

In JavaScript, you can define a function inside another function. The good thing about this technique is that the inner function has access to the local variables of the outer function.

Let's see an example:

```

1  function formula(a, b) {
2      function power(x) {
3          return Math.pow(a, x) + Math.pow(b, x);
4      }
5      return power(8) - power(2);
6  }
7
8  var result = formula(4, 2);
9  console.log(result);
10
11 // equals to :
12 //
13 // (4^8 + 2^8) - (4^2 + 2^2)
14 //
15 // Note: Where "^" we symbolize the raise in the power of
16 //       Hence, for example, 4^2 means 4 * 4.

```

(the above code snippet online)

If you run the above code on JS Bin, you will get the number 65772 printed on console. Note how the function `power` is defined inside the function `formula`. Note also, how the same function `power`, is accessing the `a` and `b` parameter variables, which are, basically, local variables to the outer function `formula`.

*Note:* the `Math.pow()` is a method on `Math` JavaScript library that takes two arguments. The result of the invocation is the first number argument raised on the power of the second. You can learn more about the `Math` JavaScript library and its many more methods. Google for it. *Hint:* Play around with the above program. Change it, experiment. You need to understand what it is doing and how.

## Invoking a Function

We have created and run a lot of programs until now with function invocations. Let's see some more details about it.

Functions can be invoked in one of the following ways:

1. As functions.
2. As methods of an object.
3. As constructors.

*Note:* There are some other methods to call a function, but we will not deal with them in this course.

### As a Function.

Not too much more to say here. We have already seen that many times. Here is one more example:

```
1 function multiplyByPi(a) {  
2     return a * Math.PI;  
3 }  
4  
5 var result = multiplyByPi(5);  
6 console.log(result);
```

(the above code snippet online)

The above program defines the function `multiplyByPi` and calls it as a regular function.

### As a Method of an Object

Assigning a function definition as a value of an object property, then that makes the function a method. Then, calling the method on the particular object, constitutes the case in which the function is called as a method. A very important difference here is that method body has access to the object itself, and hence to its properties, via the keyword `this`. Hence, usually, the methods carry out actions and execute statements based on the values of the object properties they are attached to.

Let's see an example:

```

1 var employee = {
2   firstName: "John",
3   lastName: "Wayne",
4   name: function() {
5     return this.firstName + " " + this.lastName;
6   }
7 };
8
9 console.log(employee.name());

```

(the above code snippet online)

If you run the above program on JS Bin, you will get the string "John Wayne" printed on the console. The program calls the method `#name()` on the object `employee`. The method body is implemented such that it concatenates the values of the object two properties, `firstName` and `lastName`. In order to access these properties, method body needs to use the `this` keyword.

If method does no use the `this` keyword, then will not be able to access the values of these two properties.

## Constructor Invocation

We may use a function in order to construct new objects. These functions are usually named with an uppercase first letter. Let's see an example:

```

1 function Employee(firstName, lastName) {
2   var o = {
3     firstName: firstName,
4     lastName: lastName,
5     name: function() {
6       return this.firstName + " " + this.lastName;
7     }
8   };
9   return o;
10 }
11
12 var employee = new Employee("John", "Wayne");
13
14 console.log(employee.name());

```

(the above code snippet online)

The above code is producing the same result like the previous one. It will print the string "John Wayne" on the console. It declares a constructor function `Employee` and it invokes it with `new` keyword and arguments "John" and "Wayne". The constructor function body, declares the local object `o`, and initializes it with the correct properties, including the method `name()`. Then it returns the object created. When we invoke the `new Employee("John", "Wayne")`; we take back the object created and we save it in the variable `employee`. Then, we call the method `name()` on that variable, which returns "John Wayne".

## Function Arguments and Parameters

JavaScript does not do any check when you invoke a function, in regards to the parameters vs the arguments used to invoke the function. For example, if a function definition expects 3 arguments and one invokes the function with 2, this will not raise an error at the invocation point.

Let's see what is going with parameters and the actual arguments being used to invoke a function, in a little bit more detail.

### Optional Parameters

When a function is invoked with fewer arguments than declared parameters, the additional parameters are set to the `undefined` value.

Let's see an example:

```

1  function print(numberOfTimes, charToPrint) {
2      if (charToPrint === undefined) {
3          charToPrint = 'x';
4      }
5      var result = '';
6      for (var i = 1; i <= numberOfTimes; i++) {
7          result = result + charToPrint;
8      }
9      console.log(result);
10 }
11
12 print(3, '#');
13 print(3, '-');
14 print(3);

```

(the above code snippet online)

The above program is very simple. It defines the function `print`. This function prints the character given as second argument so many times as the first argument specifies. So, for example, the `print(3, '#');` will print the character # 3 times. However, we have designed the function `print` to have its second parameter to be an optional parameter. This means that the `print()` function can be invoked without giving the second argument, but only the first. If the caller does not give the second argument, then `print()` will use, by default, the character `x`. The piece of code inside the `print()` body that does this optional parameter handling is this:

```

1  if (charToPrint === undefined) {
2      charToPrint = 'x';
3 }

```

(the above code snippet online)

It checks whether the second parameter, `charToPrint`, has been passed an argument or not. If it has not been passed, then its value is `undefined`. In that case, we assign the value `x`.

Note that, it's important that, when you define a function with optional parameters, to set the optional parameters as the last parameters of the function definition. In other words, when your function definition has both mandatory and optional parameters, you need to first list inside the parentheses the mandatory parameters and then the optional ones. Otherwise, the developer that would like to call your function wouldn't have a way to omit the optional parameters.

## Variable Length Arguments List

There are cases in which you want to define a function that could take an arbitrary number of arguments. In other words, the designer of the function does not really know in advance how many arguments would be used at invocation time.

For example, let's suppose that we want to define a function that calculates the maximum number among an arbitrary list of numbers given as input. Here are some examples of invocations of such a function:

```
1 maximum(1);
2 maximum(3, 2, 5);
3 maximum(3, 5, 17, 4, 2, 8, 5, 3, 7, 8, 9, 10, 0, 12);
```

(the above code snippet online)

JavaScript is very helpful with such a problem to solve. Provides the function body with a reserved word representing an array-like object of the actual arguments given. The object is `arguments`. It has the property `length` which returns the number of arguments given and responds to the square brackets operator `[]` that would return the actual argument on specific position.

Let's design the `maximum()` function that would work as above:

```
1 function maximum() {
2     if (arguments.length >= 1) {
3         var result = arguments[0];
4         for (var i = 1; i < arguments.length; i++) {
5             if (arguments[i] > result) {
6                 result = arguments[i];
7             }
8         }
9         return result;
10    }
11 }
12
13 console.log(maximum());
14 console.log(maximum(1));
15 console.log(maximum(3, 2, 5));
16 console.log(maximum(3, 5, 17, 4, 2, 8, 5, 3, 7, 8, 9, 10, 0, 12));
```

(the above code snippet online)

If you run the above on JS Bin, you will get this:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the following code:

```

JavaScript ▾
function maximum() {
  if (arguments.length >= 1) {
    var result = arguments[0];
    for (var i = 1; i < arguments.length; i++) {
      if (arguments[i] > result) {
        result = arguments[i];
      }
    }
    return result;
  }

console.log(maximum());
console.log(maximum(1));
console.log(maximum(3, 2, 5));
console.log(maximum(3, 5, 17, 4, 2, 8, 5, 3, 7, 8, 9, 10, 0, 12));
|

```

The Console tab shows the output of the code execution:

- undefined
- 1
- 5
- 17
- >

#### Example Of A Function With Optional Arguments

We have defined the function `maximum()` so that it's doing something useful only if it is given some arguments. That's why we have the

```

1 if (arguments.length >= 1) {
2
3   .... something useful here ....
4
5 }
```

(the above code snippet online)

`if` condition block. It relies on `arguments.length` to decide whether to deal with finding the maximum number or not.

Note that when we call the `maximum()` without arguments, it returns `undefined`. As we have said, this is the default returned value when the function execution does not reach a `return` statement that would return something else. On the other hand, when we call the `maximum()` function with one or more arguments, the function returns the maximum number found, as stored in the `result` local variable.

Note that a function definition might have both some mandatory parameters alongside with some optional parameters and can also work with some extra arbitrary arguments.

## Using Objects As Arguments

Sometimes it makes our code easier to read if we define our function to take its parameters using objects. This is true, especially, when the number parameters becomes big, greater than 2.

In other words, if the function definition is like that:

```
1 function saveOrder(firstName, lastName, product, numberofItems, amount, dateOfPur\
2 chase) {  
3     // ... function body code goes here ...  
4 }
```

(the above code snippet online)

then on every point of code that we would like to invoke this function, we would have to remember to give the actual arguments in the correct order.

```
1 saveOrder("John", "Woo", "Chair", 3, 30.00, "2016-05-28");
```

(the above code snippet online)

which means, that we would have to go back to the definition to double check that we have passed the arguments in the correct order. Also, reading the `saveOrder("John", "Woo", "Chair", 3, 30.00, "2016-05-28");` statement, is not very clear the meaning of each argument. For example, we do not know which is the first name and which is the last name, unless we go back to the function definition and study how the function has been defined.

Wouldn't it be more clear and easy if we had to call that function using a method like this?

```
1 saveOrder({  
2     firstName: "John",  
3     lastName: "Woo",  
4     product: "Chair",  
5     numberofItems: 3,  
6     amount: 30.00,  
7     dateOfPurchase: "2016-05-28"  
8 });
```

(the above code snippet online)

And not only that, wouldn't it be easier if we didn't care about the order? In other words, wouldn't it be easier if the above `saveOrder` would had the same result as the next one, in which we shuffled the order of properties?

```
1 saveOrder({  
2     numberofItems: 3,  
3     lastName: "Woo",  
4     dateOfPurchase: "2016-05-28"  
5     product: "Chair",  
6     amount: 30.00,  
7     firstName: "John",  
8 });
```

(the above code snippet online)

The answer is 'yes', of course.

We are using an object to give the argument values we want. But, in order to be able to do that, the function definition needs to be able to accept object too.

```
1 function saveOrder(orderDetails) {  
2     // ... function body code goes here ...  
3     // With orderDetails.firstName  
4     //     orderDetails.lastName  
5     // ... e.t.c. we have access to the argument values.  
6 }
```

(the above code snippet online)

Having the function definition accepting an object with properties corresponding to parameters makes sure that the invocation can be done using object too.

## Recursive Functions

Sometimes we find it quite intuitive to call a function from within its body itself. Let me bring an example, in order to make more clear what I mean. Supposedly, that we want to know what is the integer number at position 5000 in the following arithmetic sequence:

```
1 1, 4, 7, 10, 13, 16, ... e.t.c. where a number is the sum of the previous number \  
2 plus 3.
```

(the above code snippet online)

In other words, we want to write a JavaScript program that returns the number of this sequence at a given position. A function like this:

```
1 function a(index) {  
2     // .... implementation of function goes here ...  
3 }  
4  
5 // here we invoke the function for the index / position 5000.  
6  
7 var result = a(5000);  
8  
9 console.log(result);
```

(the above code snippet online)

We know how the numbers of this sequence are being built. A number at position  $i$  is equal to the number at position  $i-1$  plus 3. Definitely, an implementation like this, would do the job:

```
1 function a(index) {  
2     previous = 1;  
3     for (var i = 2; i <= index; i++) {  
4         previous += 3;  
5     }  
6     return previous;  
7 }  
8  
9 // here we invoke the function for the index / position 5000.  
10  
11 var result = a(5000);  
12  
13 console.log(result);
```

(the above code snippet online)

If you run this program on JS Bin, you will get this:

The screenshot shows a JS Bin interface with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab contains the provided code. The Console tab shows the output: 14998.

```
function a(index) {  
    previous = 1;  
    for (var i = 2; i <= index; i++) {  
        previous += 3;  
    }  
    return previous;  
}  
  
// here we invoke the function for the index / position 5000.  
  
var result = a(5000);  
  
console.log(result);
```

Console  
14998

#### Calculating Number At Specific Position of Arithmetic Sequence

The above implementation uses the for statement to iterate `index - 1` times and increment, each time, the previous number by 3. Works well and it is very fast.

However, you may be tempted to implement this function in more intuitive way. Like this:

```

1 function a(index) {
2   if (index == 1)
3     return 1;
4   return a(index - 1) + 3;
5 }
6
7 // here we invoke the function for the index / position 5000.
8 var result = a(5000);
9
10 console.log(result);

```

(the above code snippet online)

This implementation, if one reads the code carefully, is more intuitive because it follows the definition of the arithmetic sequence itself. It says that if the position you are asking is 1, then the answer is 1, but, for any other position, the answer is the answer for the previous position plus 3. Something like that:

```
1 a[i] = a[i - 1] + 3
```

(the above code snippet online)

As you can see above, in order to apply this kind of implementation, we need the function to call itself (`return a(index - 1) + 3;`). Hence, for the given argument 5000, the function is called 5000 times, going backwards until it reaches 1. And the intermediary results are reserved and then flow goes back again, adding 3 to each one of them until it calculates the final result.

Note that recursive functions may be easier to implement, but they consume more memory than the non-recursive versions of them. Also, they are a little bit slower. However, they can be proven to be very handy to write code that is easy to read.

## Throwing Errors

There are cases in which a function may be called under conditions for which the function is not designed to work properly. For example, let's take the following function:

```

1 var dayOfTheWeekInteger = function(dayOfTheWeek) {
2   switch(dayOfTheWeek) {
3     case "sunday" :
4       return 0;
5     case "monday" :
6       return 1;
7     case "tuesday" :
8       return 2;
9     case "wednesday" :
10      return 3;
11     case "thursday" :

```

```

12     return 4;
13     case "friday":
14         return 5;
15     case "saturday":
16         return 6;
17     }
18 };

```

(the above code snippet online)

If one calls that function as:

```
1 console.log(dayOfTheWeekInteger("friday"));
```

(the above code snippet online)

The result would be 5.

```

JavaScript ▾
var dayOfTheWeekInteger = function(dayOfTheWeek) {
    switch(dayOfTheWeek) {
        case "sunday":
            return 0;
        case "monday":
            return 1;
        case "tuesday":
            return 2;
        case "wednesday":
            return 3;
        case "thursday":
            return 4;
        case "friday":
            return 5;
        case "saturday":
            return 6;
    }
};

console.log(dayOfTheWeekInteger("friday"));

```

Console

5

#### Calling a Function with Valid Arguments

If you call the same function with a non-valid argument, it would return `undefined`. You only have to try that:

```
1 console.log(dayOfTheWeekInteger("non-valid day of the week"));
```

(the above code snippet online)

Many developers decide to throw an error and actually make the JavaScript break at run-time. They believe that some functions need to be designed like that so that they would easier surface up bad usages of the function by the client code, the program that is calling the function.

Having said that, when a function encounters an invalid argument can do the following:

```

1 var dayOfTheWeekInteger = function(dayOfTheWeek) {
2   switch(dayOfTheWeek) {
3     case "sunday":
4       return 0;
5     case "monday":
6       return 1;
7     case "tuesday":
8       return 2;
9     case "wednesday":
10    return 3;
11    case "thursday":
12      return 4;
13    case "friday":
14      return 5;
15    case "saturday":
16      return 6;
17    default:
18      throw new Error("Invalid Day Of The Week: '" + dayOfTheWeek.toString() + "'\\
19    ");
20  }
21 };

```

(the above code snippet online)

The line that throws the error is:

```

1   throw new Error("Invalid Day Of The Week: '" + dayOfTheWeek.toString() + "'\\
2   ");

```

(the above code snippet online)

It is the statement `throw` that is followed by the instantiation of an `Error` object, by the call to `new Error( .... )`.

The `Error()` is a constructor function and takes as argument the message to accompany the error.

Try to run the following code in JS Bin:

```

1 var dayOfTheWeekInteger = function(dayOfTheWeek) {
2   switch(dayOfTheWeek) {
3     case "sunday":
4       return 0;
5     case "monday":
6       return 1;
7     case "tuesday":
8       return 2;
9     case "wednesday":
10    return 3;

```

```

11     case "thursday":
12         return 4;
13     case "friday":
14         return 5;
15     case "saturday":
16         return 6;
17     default:
18         throw new Error("Invalid Day Of The Week: '" + dayOfTheWeek.toString() + "' \
19 ");
20     }
21 };
22
23 console.log(dayOfTheWeekInteger("non-valid day of the week"));

```

(the above code snippet online)

What you will get is this:

```

JavaScript ▾
var dayOfTheWeekInteger = function(dayOfTheWeek) {
    switch(dayOfTheWeek) {
        case "sunday":
            return 0;
        case "monday":
            return 1;
        case "tuesday":
            return 2;
        case "wednesday":
            return 3;
        case "thursday":
            return 4;
        case "friday":
            return 5;
        case "saturday":
            return 6;
        default:
            throw new Error("Invalid Day Of The Week: '" + dayOfTheWeek.toString() + "'");
    }
};

console.log(dayOfTheWeekInteger("non-valid day of the week"));

```

Console

```

"error"
"Error: Invalid Day Of The Week: non-valid day of the week
    at dayOfTheWeekInteger (keribiwizi.js:18:13)
    at keribiwizi.js:22:38
    at https://static.jsbin.com/js/prod/runner-3.38.13.min.js:1:13926
    at https://static.jsbin.com/js/prod/runner-3.38.13.min.js:1:10855"
>

```

Throwing an Error When Invalid Argument

## Catching Errors

The client code that is calling a function, and knows that this function might throw an error, usually, but not always, might want to handle the error thrown, instead of letting it bubble up and displayed as error at users browser or application.

This is called *catching errors*. When we know that a function might throw an error and we want to catch it, we use a `try { } catch(ex) { }` block.

Inside the `try { }` part we call the function and inside the `catch(ex) { }` we handle the error.

Run the following code inside JS Bin:

```

1 var dayOfTheWeekInteger = function(dayOfTheWeek) {
2     switch(dayOfTheWeek) {
3         case "sunday":
4             return 0;
5         case "monday":
6             return 1;
7         case "tuesday":
8             return 2;
9         case "wednesday":
10            return 3;
11        case "thursday":
12            return 4;
13        case "friday":
14            return 5;
15        case "saturday":
16            return 6;
17        default:
18            throw new Error("Invalid Day Of The Week: '" + dayOfTheWeek.toString() + "'\\
19            ");
20    }
21 };
22
23 try {
24     dayOfTheWeekInteger("non-valid day of the week");
25 }
26 catch(ex) {
27     console.log("Something went wrong: " + ex.message);
28 }
```

(the above code snippet online)

You will NOT get an error. Instead, you will get the message "Something went wrong: Invalid Day Of The Week: 'non-valid day of the week'". You do not get an error, but you get the message that is printed by the statement:

```
1 console.log("Something went wrong: " + ex.message);
```

(the above code snippet online)

The statement

```
1 catch(ex) {
2     console.log("Something went wrong: " + ex.message);
3 }
```

(the above code snippet online)

catches the error that `dayOfTheWeekInteger("non-valid day of the week")`; statement throws and saves its details inside `ex` local variable. Then it calls `ex.message` to get the property with name `message` that contains that message that `dayOfTheWeekInteger` has assigned to the `Error` instance, and just prints the message (prefixed with “Something went wrong”) at the console.

Note that if you want to both catch/handle the error and then propagate that up in the calling stack, you can call `throw ex;`:

```

1 try {
2     dayOfTheWeekInteger("non-valid day of the week");
3 }
4 catch(ex) {
5     console.log("Something went wrong: " + ex.message);
6     throw ex;
7 }
```

(the above code snippet online)

## Throwing Custom Errors

Finally, there may be cases that you would like to throw custom error types, instead of `Error`. Custom error types are good to bear better the meaning of the error.

In order to throw a custom error, you first need to define it. This is a custom error definition, `ArgumentError`:

```

1 function ArgumentError(message) {
2     this.name = 'ArgumentError';
3     this.message = message || 'Argument Error';
4     this.stack = (new Error()).stack;
5 }
6 ArgumentError.prototype = Object.create(Error.prototype);
7 ArgumentError.prototype.constructor = ArgumentError;
```

(the above code snippet online)

We have defined a constructor function. Inside the constructor function we define 3 properties for the constructed object.

- `name`. This should be the string representation of the constructor name.
- `message`. Assign the value passed to the constructor or a default value. For example “Argument Error”.
- `stack`. This is standard.

Then we are saying that `ArgumentError` should be an object deriving from `Error`:

```
1 ArgumentError.prototype = Object.create(Error.prototype);
```

(the above code snippet online)

and that its constructor is the ArgumentError function:

```
1 ArgumentError.prototype.constructor = ArgumentError;
```

(the above code snippet online)

Let's incorporate that inside our example for dayOfTheWeekInteger function. Write the following in JS Bin:

```
1 function ArgumentError(message) {
2     this.name = 'ArgumentError';
3     this.message = message || 'Argument Error';
4     this.stack = (new Error()).stack;
5 }
6 ArgumentError.prototype = Object.create(Error.prototype);
7 ArgumentError.prototype.constructor = ArgumentError;
8
9 var dayOfTheWeekInteger = function(dayOfTheWeek) {
10    switch(dayOfTheWeek) {
11        case "sunday":
12            return 0;
13        case "monday":
14            return 1;
15        case "tuesday":
16            return 2;
17        case "wednesday":
18            return 3;
19        case "thursday":
20            return 4;
21        case "friday":
22            return 5;
23        case "saturday":
24            return 6;
25        default:
26            throw new ArgumentError("Invalid Day Of The Week: '" + dayOfTheWeek.toString\
27 g() + "'");
28    }
29 };
30
31
32 dayOfTheWeekInteger("non-valid day of the week");
```

(the above code snippet online)

If you run this, you will get:

The screenshot shows a browser's developer tools console. On the left, there is a code editor window with some JavaScript code. On the right, the 'Console' tab is active, showing the output of the code execution.

```

JavaScript <-->
function ArgumentError(message) {
    this.name = 'ArgumentError';
    this.message = message || 'Argument Error';
    this.stack = (new Error()).stack;
}
ArgumentError.prototype = Object.create(Error.prototype);
ArgumentError.prototype.constructor = ArgumentError;

var dayOfTheWeekInteger = function(dayOfTheWeek) {
    switch(dayOfTheWeek) {
        case "sunday":
            return 0;
        case "monday":
            return 1;
        case "tuesday":
            return 2;
        case "wednesday":
            return 3;
        case "thursday":
            return 4;
        case "friday":
            return 5;
        case "saturday":
            return 6;
        default:
            throw new ArgumentError("Invalid Day Of The Week: '" + dayOfTheWeek.toString() + "'");
    };
}

dayOfTheWeekInteger("non-valid day of the week");

```

Console

```

"error"
"Error
    at new ArgumentError (keribiwizi.js:4:19)
    at dayOfTheWeekInteger (keribiwizi.js:26:13)
    at keribiwizi.js:31:1
    at https://static.jsbin.com/js/prod/runner-3.38.13.min.js:1:13926
    at https://static.jsbin.com/js/prod/runner-3.38.13.min.js:1:10855"

```

ArgumentError thrown

As you can see, the error thrown is not an “Error”, it is an “ArgumentError”.

## Tasks and Quizzes

**Before you continue, you may want to know that:** You can sign up to [Tech Career Booster](#) and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

### Tasks

(1) Write a function and a program that would demonstrate its use. This function should take as input a temperature in Fahrenheit and would return it in Celsius. Make also sure that you are using the `Math.round()` function to round the result to an integer number.

**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

(2) Write the factorial function and a program that demonstrates its use. Do not use recursive method. If you do not know what a factorial function is ... google for it.

**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

(3) Write the factorial function and a program that demonstrates its use. Use recursive method.

**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

(4) Write a program in which an object, `employee`, has 3 properties:

1. A name. This is the name of the employee.

2. A `salary`. This is the salary of the employee.
3. A `numberOfYears`. This is the number of years the employee is in the company.

Then make sure that this object has a method that would return the bonus this employee would deserve according to the following logic:

1. If the employee is more than 10 years in the company, then bonus would be 2 times its salary.
2. If the employee is up to 10 years in company, including, then bonus would be 1 times its salary.
3. If the employee is up to 5 years in the company, including, then bonus would be half its salary.
4. If the employee is less than 6 months in the company, wouldn't deserve a bonus.

Demonstrate that the implementation of your method is correct, by printing the bonus for various example employees.

**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

(5) Write a function that takes a string and returns back the string with its first character uppercase. Call that method with various examples to demonstrate that it works. Note that if the function is called without arguments or it is called with a blank string argument, it should return a blank string.

**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

(6) Write a function that takes an arbitrary number of arguments, all strings, and returns one string being the concatenation of all strings given as arguments. The concatenation needs to join the strings using a blank space. Also, make sure that the first character of each individual string appears uppercase on final concatenated string. This is an example of calling this method and the result that it returns:

```
1 var result = concatenateAndUppercase("foo", "bar", "mary", "woo");
2 console.log(result); // prints "Foo Bar Mary Woo"
```

Note that function should return an empty string if no arguments are given. And it should handle arguments that are blank strings too (by ignoring them). Hint: You may want to reuse the function that you have developed on previous program (5).

**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

## 9 - JavaScript In Web Browsers

### Summary

The previous chapters on JavaScript were about some of the core features of the language. Now we start the real fun, because we are going to see how JavaScript is used by the browsers. This is what we call client-side JavaScript.

You will start executing JavaScript code from within an HTML page. We are going to have fun with timers and scheduling actions to be executed in the future, either once, or periodically.

You are going to learn how you can change the page loaded dynamically.

For example, this is one thing that you will learn to do. On this example, the date and time are dynamically updated every 1 second:

#### Date And Time Automatically Updated

You will also learn how to present information to user and get some back from him.

#### Using Dialog Methods

Finally, you will learn how to open and close windows and tabs dynamically.

#### Opening and Closing a Window

### Learning Goals

1. Learn to execute JavaScript code within an HTML Web page, using the `<script>` tag.
2. Learn how to include JavaScript code within your HTML Web page, having the JavaScript source within separate files.
3. Learn about the `window` object.
4. Learn about the `location` property of the `window` object.
5. Learn how you can change the web page displayed by setting a new value to the `location` property.
6. Learn about the `setTimeout()` method.
7. Learn about the `document` property.
8. Learn about the `document.getElementById()` method.
9. Learn about the `setInterval()` and `clearInterval()` methods.
10. Learn about the native dialog boxes:
  1. `alert()`
  2. `confirm()`
  3. `prompt()`
11. Learn about opening and closing windows (`open()` and `close()` methods).

### Introduction

#### Execute JavaScript on a Web Page

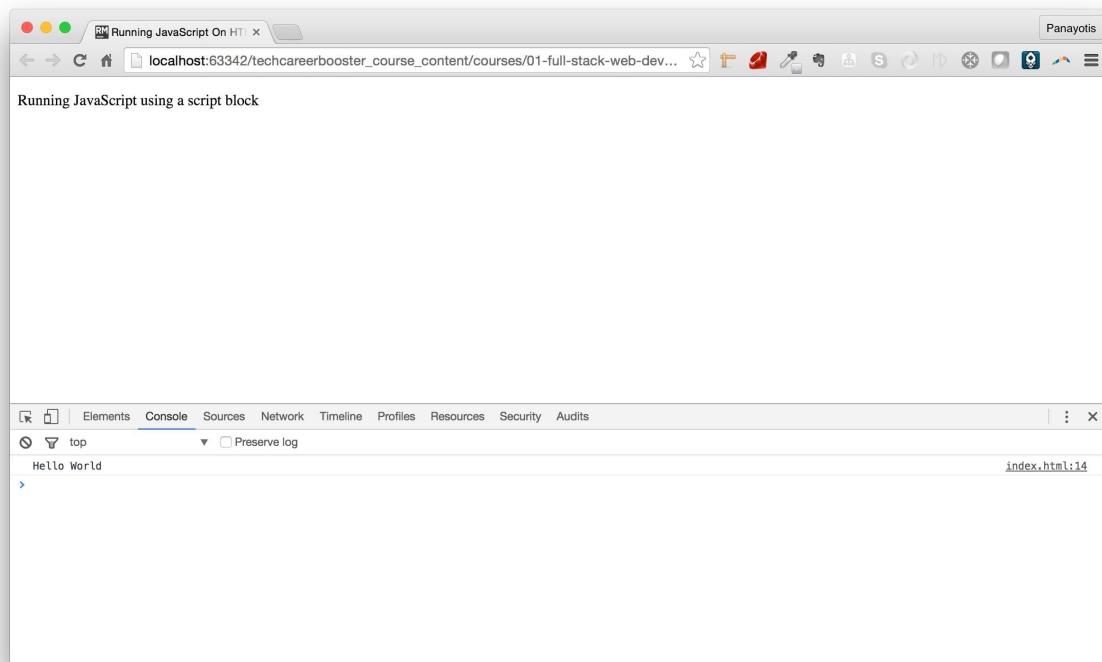
Let's see how we can execute some JavaScript code on a web page.

Write the following HTML page:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Running JavaScript On HTML Page Using Script Block</title>
7   </head>
8   <body>
9     <p>
10    Running JavaScript using a script block
11  </p>
12
13  <script>
14    console.log("Hello World");
15  </script>
16 </body>
17 </html>
```

(the above code snippet online)

Save the above content into a file `index.html` and load the page on your browser. Make sure that you have the Chrome developer tools open and the console tab enabled. What you will see will be this:



### Running JavaScript Using a Script Block

As you can see, the message "Hello World" is displayed in the console area. This is thanks to the

```
1 <script>
2   console.log("Hello World");
3 </script>
```

(the above code snippet online)

script block that is included inside the HTML page.

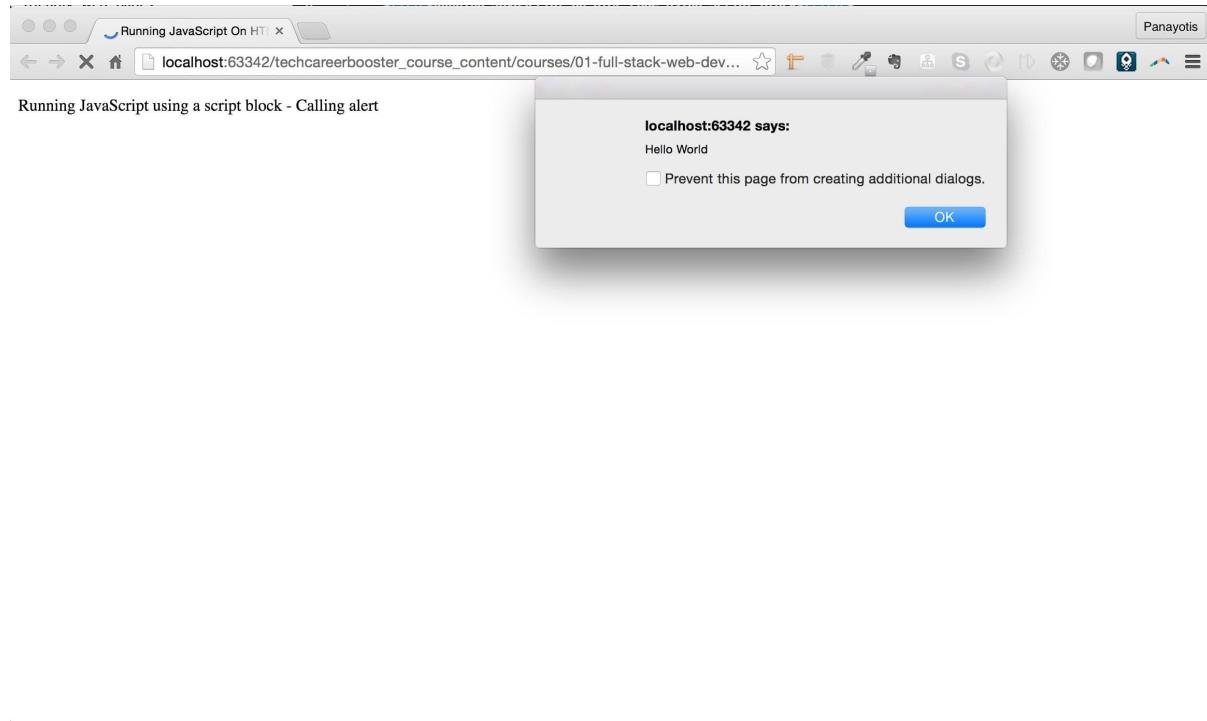
Generally, whatever appears inside a `<script>` element is considered to be JavaScript code to be executed by the browser.

Let's see another example:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Running JavaScript On HTML Page Using Script Block</title>
7   </head>
8   <body>
9     <p>
10       Running JavaScript using a script block - Calling alert
11     </p>
12
13   <script>
14     alert("Hello World");
15   </script>
16 </body>
17 </html>
```

(the above code snippet online)

If you save and load the above page on your browser, you will see this:



Calling alert() from script block

## Calling JavaScript From Files

Usually, we do not write the JavaScript inline with the HTML page source code. But, we prefer to have the JavaScript code written on separate JavaScript files, with filenames ending in .js. An HTML page usually stores its JavaScript related code inside a sub-folder named assets/javascripts.

Let's write the previous `alert()` example using a separate JavaScript file.

This is going to be the HTML file:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Running JavaScript On HTML Page Using Script Block</title>
7   </head>
8   <body>
9     <p>
10       Running JavaScript using a script block - Calling alert
11     </p>
12
13     <script src="assets/javascripts/alert.js"></script>
14   </body>
15 </html>
```

(the above code snippet online)

and this is going to be the assets/javascripts/alert.js file:

```
1 alert("Hello World");
```

(the above code snippet online)

Save all files and load the HTML page on your browser. You will experience exactly the same behaviour like you did with the previous version, with the inline JavaScript code.

We will use this technique to store our JavaScript code from now on.

Let's proceed with more client-side JavaScript code and interaction with our web page and browser window.

## The Window Object

The `window` object is the main entry point for JavaScript code that will run in a browser. It refers to a browser window or frame.

### `location` property

The `location` property contains the URL of the page that we have visited. Also, it can be used to change the page displayed.

Write the following HTML page:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Displaying the location property</title>
7   </head>
8   <body>
9     <p>
10       Location property
11     </p>
12
13     <script src="assets/javascripts/location.js"></script>
14   </body>
15 </html>
```

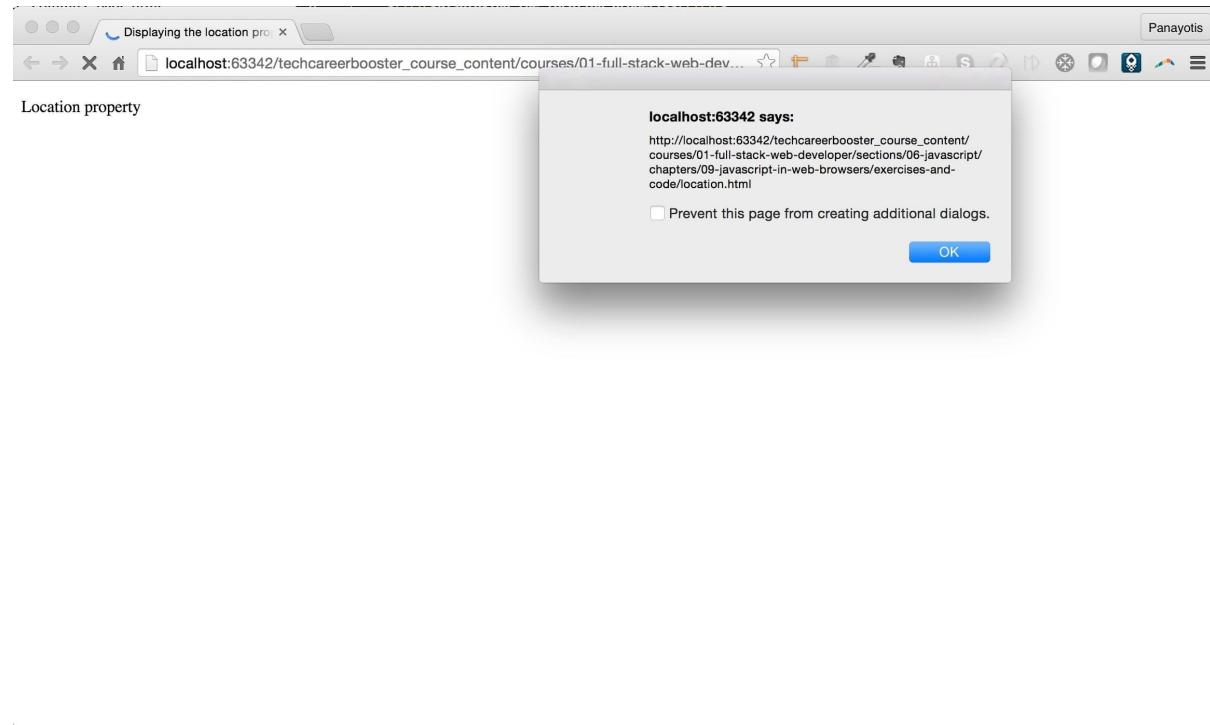
(the above code snippet online)

and the accompanying assets/javascripts/location.js file:

```
1 alert(window.location);
```

(the above code snippet online)

save both and load the page. You will see something like this:



window.location Displayed

The `alert()` displays the value of the URL that exists on your browser, the WEB address of your page loaded.

But as we said, you can change the current page displayed by setting this property to another value. Write the following HTML page

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Setting the location property</title>
7   </head>
8   <body>
9     <p>
10       Setting Location property
11     </p>
12
13     <script src="assets/javascripts/setting-location.js"></script>
14   </body>
15 </html>
```

(the above code snippet online)

and the corresponding assets/javascripts/setting-location.js file:

```
1 window.location = "https://www.google.com";
```

(the above code snippet online)

Then save all and load the page on your browser. Momentarily, you will see your page displayed, but after fractions of second, you will be looking at “https://www.google.com” and not your own page. This is done thanks to the `window.location = "https://www.google.com";` JavaScript statement inside the assets/javascripts/setting-location.js file.

*Note:* `alert()` is a method called on `window`. Same goes for other methods, like `console()`, that are called on the global namespace. They are essentially being called on the `window` object. You can try that by calling `window.alert("Hello World");` instead of `alert("Hello World");`. The result will be the same. Same goes for the `location` property. If you just set `location = "https://www.google.com";` the result will be the same.

## **setTimeout()**

The `setTimeout()` method takes two arguments.

1. A function to be executed after a specified number of milliseconds.
2. The number of milliseconds after which the function will be executed.

This is very useful because it allows us to delay the execution of a function as much as we want.

Let's see the following example of setting the location property, but only after 2 seconds have elapsed. Change the file assets/javascripts/setting-location.js to be:

```
1 setTimeout(function(){
2   window.location = "https://www.google.com";
3 }, 2000);
```

(the above code snippet online)

Then load again the page that uses this file. You will see the Google page appearing, only after 2 seconds have elapsed.

*Note:* If you call the `setTimeout()` function with a time of 0 ms, the function you specify is not invoked right away. Instead, it is placed on a queue to be invoked “as soon as possible”, after any currently pending event handlers finish running.

**document**

A very important global property when we are running JavaScript in browsers, is the `document` property. This is an object that represents the state of the HTML content being displayed on the browser window.

We will see how we can use `document` property to manipulate parts of the content dynamically.

**document.getElementById()**

The `document.getElementById()` is a very useful method that can return an object representation of an HTML element from the page that is loaded. It retrieves the element using the id value of the element.

```
1 var element = document.getElementById("current-date-and-time");
```

(the above code snippet online)

Then we can call the property `.innerHTML`, for example, and set the HTML content of the element at hand.

```
1 element.innerHTML = new Date();
```

(the above code snippet online)

Let's see the whole program. Here is the HTML page:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Get Element By Id Example</title>
7   </head>
8   <body>
9     <h2 id="current-date-and-time">
10    </h2>
11
12    <script src="assets/javascripts/setting-current-date-and-time.js"></script>
13  </body>
14 </html>
```

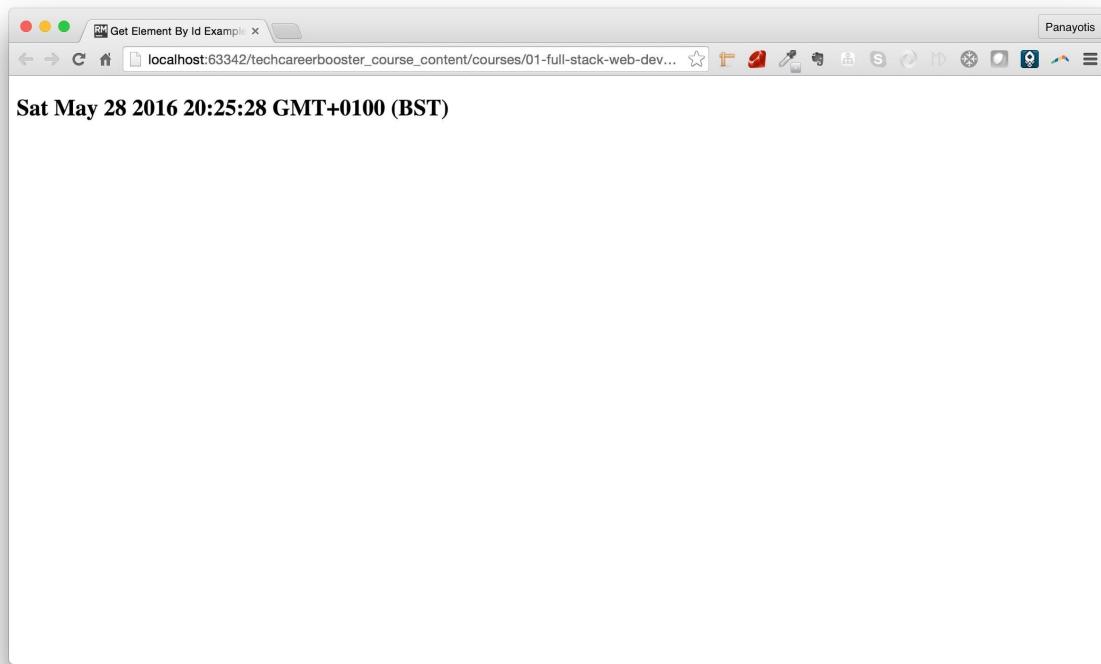
(the above code snippet online)

And the `assets/javascripts/setting-current-date-and-time.js` JavaScript file has the following content:

```
1 var element = document.getElementById('current-date-and-time');
2 element.innerHTML = new Date();
```

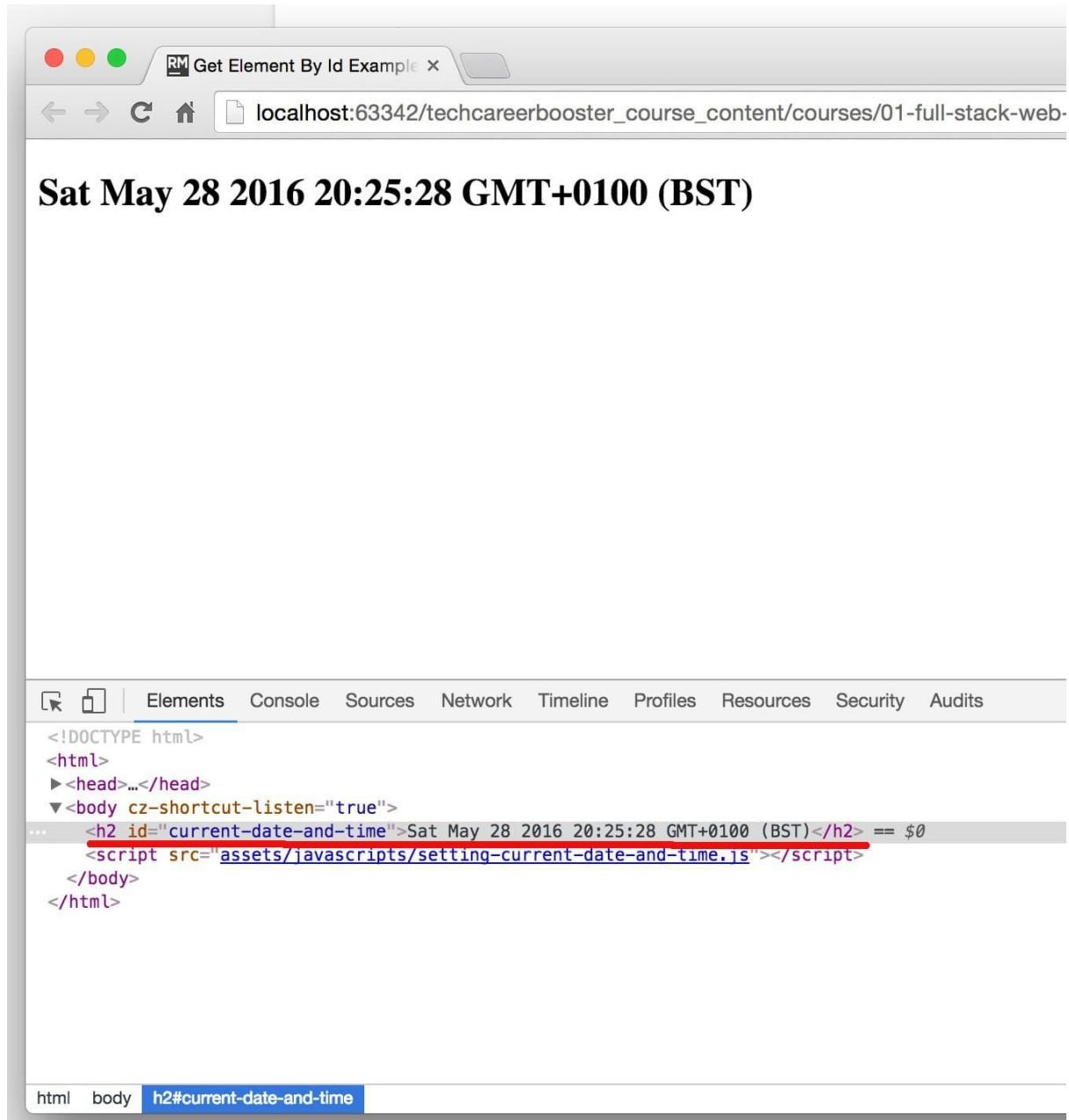
(the above code snippet online)

As you can see, initially, the element `h2` with id `current-date-and-time` does not have any content. But, when we load the page on the browser, we will see this:



#### Current Date And Time Added Dynamically

You can see the current date and time displayed as an `h2` header. This is because it has been added by the JavaScript code that we have written inside the file `assets/javascripts/setting-current-date-and-time.js`. If you see the element on Chrome developer tools after the page has been loaded, you will see this:



#### Element Now Has the Current Date And Time

But, note that, the source code of the page has not changed. In the source code of the page the h2 element is empty.

#### **setInterval() and clearInterval()**

Let's see another very useful window method, the `setInterval()`. This is similar to `setTimeout()`. But, instead of firing the function code once, it fires it repeatedly and forever. The interval between the executions of the function is equal to the duration given as last argument to `setInterval()`.

Let's change the previous program to periodically update the current date and time, every 1 second.

The HTML page content does not change. You have to change the content of the file assets/javascripts/settings.js. It has to be like this:

```
1 setInterval(function() {  
2     var element = document.getElementById('current-date-and-time');  
3     element.innerHTML = new Date();  
4 }, 1000);
```

(the above code snippet online)

If you now load the HTML page again, you will see the date and time string to being updated every 1 second. (1000 milliseconds = 1 second).

### Date And Time Automatically Updated

Note that `setInterval()`; returns an object that can be used later on to stop a repetition, using the `clearInterval()` function.

Let's modify the previous JavaScript content to stop the repetition of updating the date and time stamp, after 10 seconds, for example:

```
1 var timer = setInterval(function() {  
2     var element = document.getElementById('current-date-and-time');  
3     element.innerHTML = new Date();  
4 }, 1000);  
5  
6 setTimeout(function() {  
7     clearInterval(timer);  
8 }, 10000);
```

(the above code snippet online)

As you can see above:

1. we save the returned value of the `setInterval()` call, into the variable `timer`. Then
2. we call the `setTimeout()` function to register a new timeout handling function, that will be executed 10 seconds later. Its responsibility is to clear the timer created by the previous `setInterval()`, a.k.a. the `timer` timer. The clear action is done with `clearInterval()` method.

Try to load your page again. You will see that after 10 repetitions, the date and time is no longer updated.

## Dialog Boxes

`window` object offers 3 methods that have to do with displaying messages to user or getting input from them. We have already seen `alert()`. The other two methods are: `confirm()` and `prompt()`.

1. The `alert()` method displays a message to the user and user acknowledges and closes the dialog by clicking on the OK button.
2. The `confirm()` method displays a message and allows the user to click on either of two buttons: OK or CANCEL. In both cases the dialog closes, but if the user closes the dialog with OK, then `true` is returned by `confirm()`. If user closes the dialog with CANCEL, then `false` is returned by `confirm()`.
3. The `prompt()` displays a message, waits for the user to enter a string and then closes the dialog and returns string given.

Let's try to write a program that demonstrates the use of these dialog boxes:

The HTML page is going to be like that:

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Using Dialog Methods</title>
7   </head>
8   <body>
9     <p>
10    Using Dialog Methods
11  </p>
12
13  <script src="assets/javascripts/using-dialog-methods.js"></script>
14 </body>
15 </html>
```

(the above code snippet online)

And the JavaScript code stored inside `assets/javascripts/using-dialog-methods.js` will be like that:

```

1 var name = prompt("What is your name?");
2 if (name !== "null") {
3   var answer = confirm(name + ", do you want me to display a random number?");
4   if (answer) {
5     alert("This is a random number: " + Math.random());
6   }
7 }
```

(the above code snippet online)

If you save the above and load the page on your browser you will experience something like this:

## Using Dialog Methods

If you study the above code you will understand the following:

1. The `var name = prompt("What is your name?");` displays a dialog to the user to ask his name. Whatever user gives, is stored in the variable `name`.
2. If the user gives something as a name (`name !== "null"`), then we display a new dialog box to user, with the question whether he wants to be presented with a random number. On that new dialog box, user is allowed to select either `cancel` or `ok`. If he selects `ok`, `true` is saved on `answer` variable. Otherwise, `false` is saved on `answer` variable.
3. According to `answer` value, we either display a third dialog or not.

## Opening and Closing Windows

You can open a new window (or tab, this is a browser configuration setting) using the `open()` method on the `window` object.

Let's try the following: The HTML page content:

```

1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta charset="utf-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Opening Windows</title>
7    </head>
8    <body>
9      <p>
10        Open a blank window after 2 seconds.
11      </p>
12
13      <script src="assets/javascripts/opening-windows.js"></script>
14    </body>
15  </html>
```

(the above code snippet online)

With the `assets/javascripts/opening-windows.js` content to be:

```

1  setTimeout(function() {
2    var new_window = window.open();
3  }, 2000);
```

(the above code snippet online)

If you load the above page on your browser, you will see your page being open, and after 2 seconds, a new window (or tab) appearing, having load a blank page (or you default browser start page).

### Opening a blank window

The `window.open()` without any arguments opens a window with a blank page loaded. But, you can give as first argument the URL to a web page that you would like being opened. Let's do that. We will change our `assets/javascripts/opening-windows.js` page to open "`http://www.nationalgeographic.com`" page.

```
1 setTimeout(function() {  
2     var new_window = window.open("http://www.nationalgeographic.com");  
3 }, 2000);
```

(the above code snippet online)

If you reload your page, you will see, after 2 seconds, the National Geographic page to be open on a new window/tab.

### Opening a new window with a specific page loaded

The `window.open()` method returns a window handle that can be used, later on, to close the window. Let's try that. The following program opens a new window after 2 seconds the page loads, and then, after 10 more seconds, it closes it. Let's change the `assets/javascripts/opening-windows.js` content as follows:

```
1 setTimeout(function() {  
2     var new_window = window.open("http://www.nationalgeographic.com");  
3  
4     setTimeout(function() {  
5         new_window.close();  
6     }, 10000);  
7  
8 }, 2000);
```

(the above code snippet online)

If you save the above JavaScript code and you reload the page, you will see something like this:

### Opening and Closing a Window

The `open()` function, can take as a second argument a window name. This is something that can uniquely identify, programmatically, the window in a friendly way. Note that this is not the window title. Also, it can take a 3rd argument which are options that have to do with how the window should be created.

Let's try another example. Update the file `assets/javascripts/opening-windows.js` with the following content:

```
1 setTimeout(function () {  
2     var new_window = window.open("http://www.nationalgeographic.com",  
3         "national_geographic",  
4         {  
5             menubar: false,  
6             width: 640,  
7             height: 480,  
8             top: 250,  
9             left: 250  
10        });  
11  
12 }, 2000);
```

(the above code snippet online)

On the above content, we name the new window “national geographic”. Also, we disable the menu bar and we set the size of the window and its position.

If you save the above file and load the page on your browser, you will see something like this:

#### Opening a new window with custom size and position

As you can see, the window now is open on a new window (rather than new tab). This is because we have specified its size and its position.

## Tasks and Quizzes

**Before you continue, you may want to know that:** You can sign up to [Tech Career Booster](#) and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

### Tasks

Write a JavaScript program that starts its execution 5 seconds after the page has loaded. The program should open 10 different web pages on their own tab. Each window should be pointing to one of the following 10 web pages:

1. <https://www.google.com>
2. <http://www.nationalgeographic.com>
3. <http://cnn.com>
4. <https://www.zsl.org>
5. <http://www.telegraph.co.uk/>
6. <http://www.go2africa.com/>
7. <http://www.animalplanet.com/>
8. <http://www.theguardian.com/>
9. <http://travel.usnews.com/>
10. <http://www.independent.co.uk/>

A window should be open only after 2 seconds after the opening of the previous window. When all windows are open, then program should wait for 5 seconds before start closing the windows in the following order: first the even positioned windows (the 2nd, the 4th, the 6th, the 8th and the 10th) and then the odd positioned windows (the 1st, the 3rd, the 5th, the 7th and the 9th). Closing one window should wait for 2 seconds before closing the next one. At the end it should display a message to user whether he would like to see this sequence of openings and closings again. If the user answers yes, then the whole process should start from the beginning. Else, the program would end.

Watch the following video to see the final web page and JavaScript program in action:

#### Task to Open and Close Windows

Some hints that might be helpful to you:

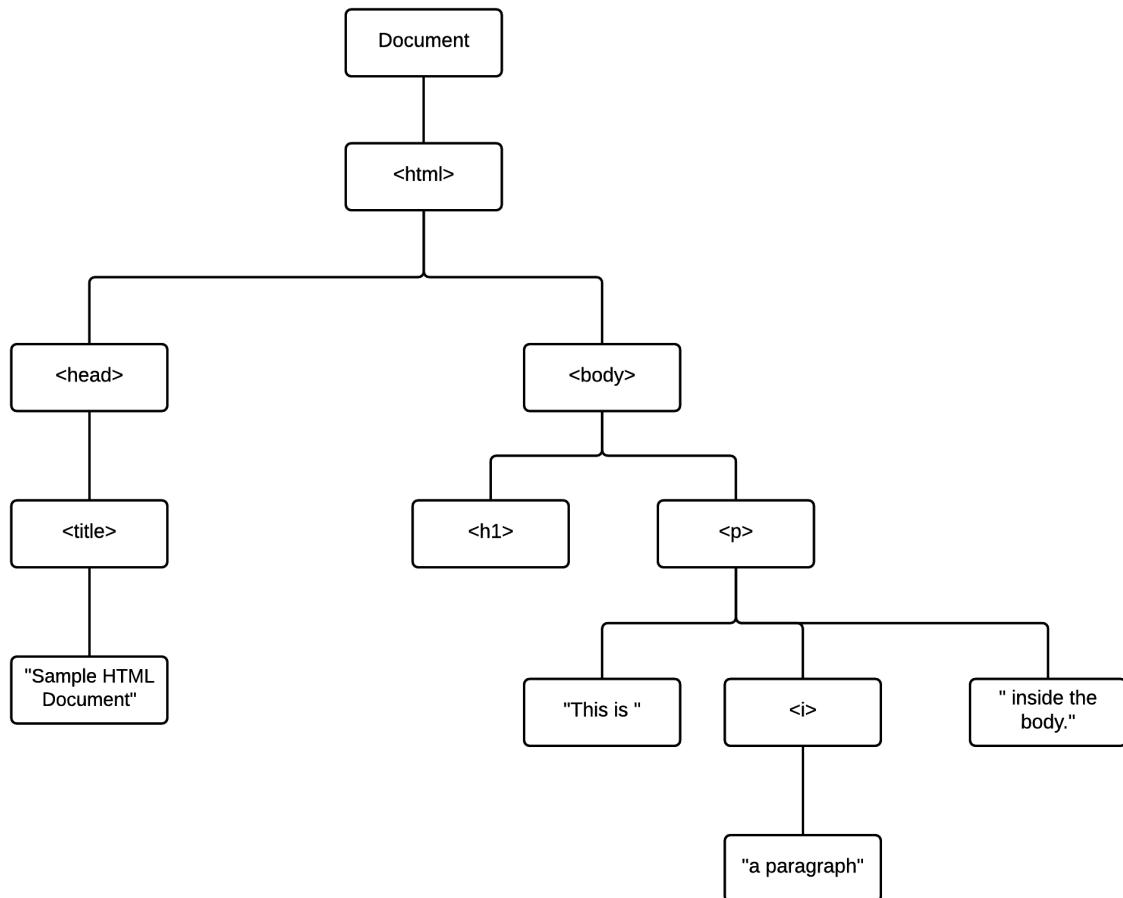
1. Consider declaring an array that would hold information about the windows. e.g. the URL/address and the window handle returned by `open()`. You will need that in order to close the window later on.
2. You might want to think recursively. Maybe a function that opens a window and calls itself to open the next one might be a good approach. But, this is just an idea. Try to work your ideas too.
3. In any case, try to break the big problem into smaller problems. **Divide and conquer**. This will help you reach a total solution at the end.

**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

## 10 - Introduction to DOM (Document Object Model)

### Summary

This chapter is an introduction to DOM, the document object model. It is very important to have a first understanding of how the DOM is built according to the HTML content.



DOM Representation of a Sample HTML Page

Then, we are going to learn the necessary JavaScript code that will allow us to locate HTML elements of the document. Locating the elements will have the chance to dynamically alter the HTML content. This is how we leave the static aspect of our Web page and we add dynamics to the Web page that would make it more attractive for the users to use.

Note that the techniques that we learn here are very basic. We will not delve into advanced DOM manipulation techniques. Advanced DOM manipulation will be covered later on in the jQuery chapters.

### Learning Goals

1. Learn about the DOM (Document Object Model).

2. Learn about the variable `document`.
3. Learn to select an HTML element by id.
4. Learn to select elements by tag.
5. Learn to select elements by class.
6. Learn to select elements by any CSS Selector.
7. Learn to access and update the HTML content of an HTML element.

## DOM

The DOM (Document Object Model) is a JavaScript object that is used to represent an HTML document.

*Note:* Actually, it is used to represent an XML document too. But, for the purpose of this course, we will stay to its HTML document aspect.

It is the variable `document` that actually holds the whole HTML structure in DOM representation.

The DOM represents an HTML document in a similar way to the way the HTML document is structured.

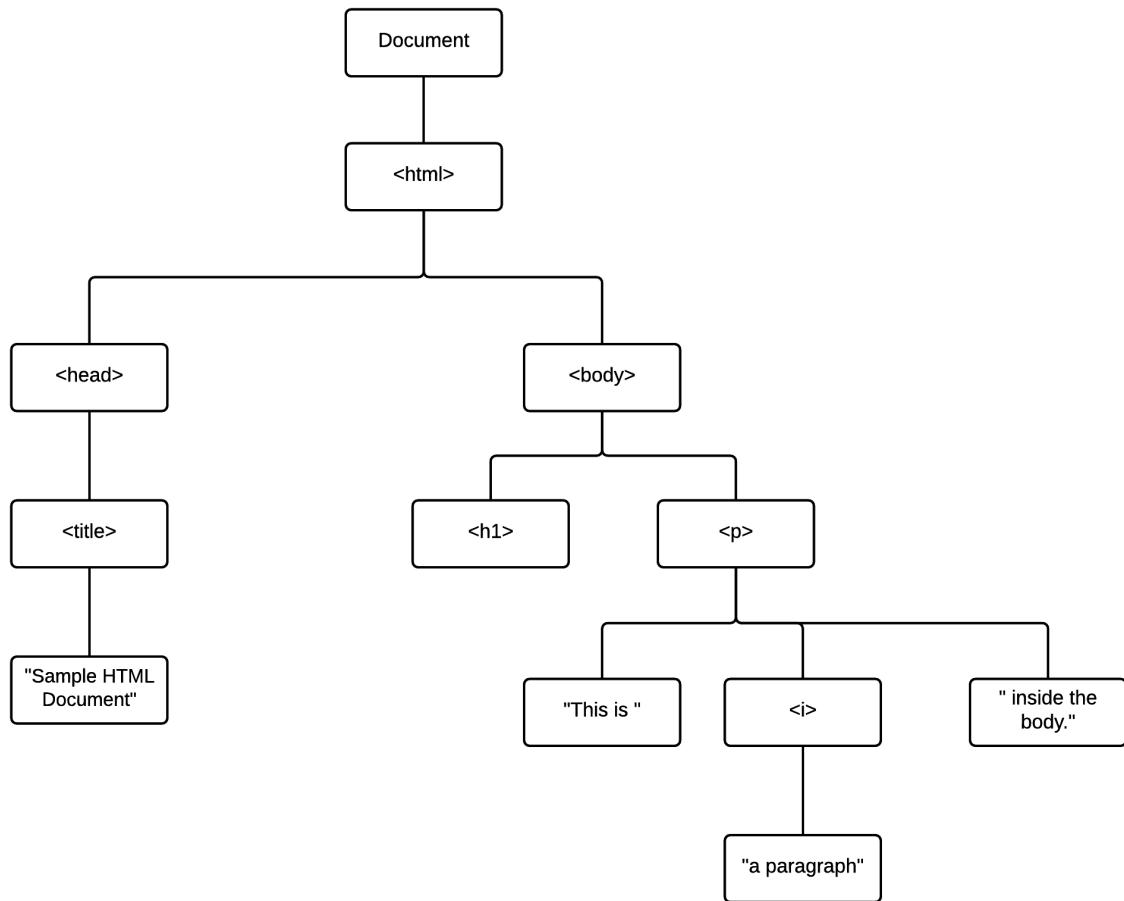
Lets assume that we have the following HTML document:

```
1 <html>
2   <head>
3     <title>Sample HTML Document</title>
4   </head>
5   <body>
6     <h1>Sample H1 content</h1>
7     <p>This is <i>a paragraph</i> inside the body.</p>
8   </body>
9 </html>
```

(the above code snippet online)

As you already know, the HTML document has a tree-like structure organization of its elements. For example, everything belongs to the `<html>` element. So, the `<html>` element can be considered the parent of all other elements. `<html>` element has 2 children. 1 `<head>` element and 1 `<body>` element. And so on.

We can visualize the tree structure of the HTML content of the above document as follows:



DOM Representation of a Sample HTML Page

## Selecting Elements of The DOM

Let's write the following HTML page.

```

1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Sample HTML Document</title>
5      <link href="assets/stylesheets/main.css" rel="stylesheet" type="text/css">
6      <script src="assets/javascripts/main.js"></script>
7    </head>
8    <body>
9      <h1 id="header" class="bold">Sample H1 content</h1>
10     <p id="paragraph-1" class="bold">This is <i>a paragraph</i> inside the body.< \
11   /p>
12
13     <p id="paragraph-2" class="light">This is the second paragraph</p>
14
  
```

```
15     <input type="text" value="Hello" class="light"/>
16 </body>
17 </html>
```

(the above code snippet online)

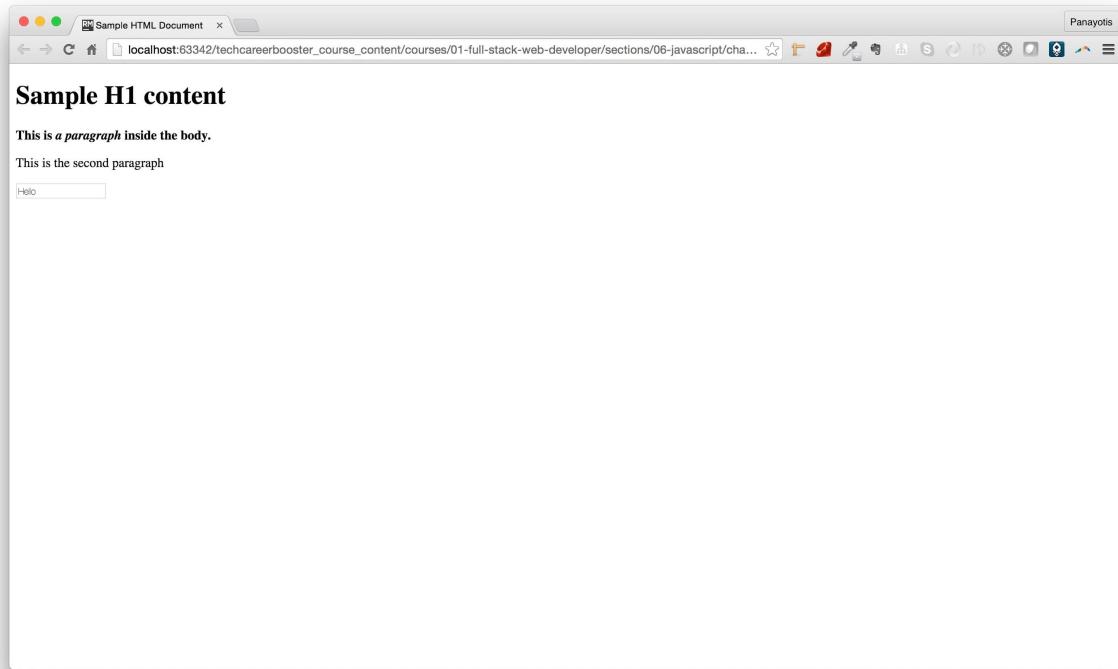
with the assets/stylesheets/main.css being:

```
1 .bold {
2     font-weight: bolder;
3 }
4
5 .light {
6     font-weight: lighter;
7 }
```

(the above code snippet online)

*Note:* Initially, the assets/javascripts/main.js file is empty.

If you save the above files and load the page on your browser, you will see this:



Sample HTML Page

## Selecting Elements by Id

document offers a way for you to select an element by its id. It is the method `getElementById()` which takes as input the id of the element you want to select.

While you have the page loaded, open the Chrome developer tools, on tab Console, and give the following commands:

```
1 var h1 = document.getElementById("header");
2 h1
```

(the above code snippet online)

The screenshot shows the Chrome Developer Tools interface with the 'Console' tab selected. The console output displays the following:

```
> var h1 = document.getElementById("header");
< undefined
> h1
<- <h1 id="header" class="bold">Sample H1 content</h1>
>
```

The page content includes an 

# element with the text "Sample H1 content". A text input field contains the text "Hello".

Selecting h1 by id

The JavaScript command `var h1 = document.getElementById("header");` locates the HTML element that has id "header" and returns that and saves it inside the variable `h1`.

## Selecting Elements by Tag

Another way to select elements from the DOM is by tag. For example, you can select all the paragraph elements. Try the following command on Chrome developer tools console for the page already loaded:

```
1 var paragraphs = document.getElementsByTagName("p");
2 paragraphs
```

(the above code snippet online)

If you do that, you will get something like this:

## Sample H1 content

This is a *paragraph* inside the body.

This is the second paragraph

```
Hello
```

A screenshot of a browser's developer tools, specifically the Console tab. The tab bar includes Elements, Console, Sources, Network, Timeline, Profiles, Resources, Security, and Audits. The Console tab is active. At the top left of the console area, there are icons for back, forward, and refresh, followed by a dropdown menu set to 'top' and a checkbox for 'Preserve log'. Below this, the JavaScript prompt starts with '> var paragraphs = document.getElementsByTagName("p");'. The response shows 'undefined' on the next line, followed by the variable name 'paragraphs' on the third line, and then an array containing two paragraph elements on the fourth line. The array is shown with a small triangle icon before it, indicating it's expandable. The first element is a bolded paragraph with id 'paragraph-1', and the second is a light-colored paragraph with id 'paragraph-2'.

```
> var paragraphs = document.getElementsByTagName("p");
< undefined
> paragraphs
< [ ><p id="paragraph-1" class="bold">...</p>, <p id="paragraph-2" class="light">This is the second paragraph</p>]
>
```

### Selecting Elements by Tag Name

As you can see above, the result is an Array of paragraphs. The `getElementsByTagName()` method, always returns an array, because the tag names are not unique within an HTML document.

## Selecting Elements by Class

Another way you can select elements is by a CSS class name. Let's try the following:

```
1 var boldElements = document.getElementsByClassName("bold");
2 boldElements
```

(the above code snippet online)

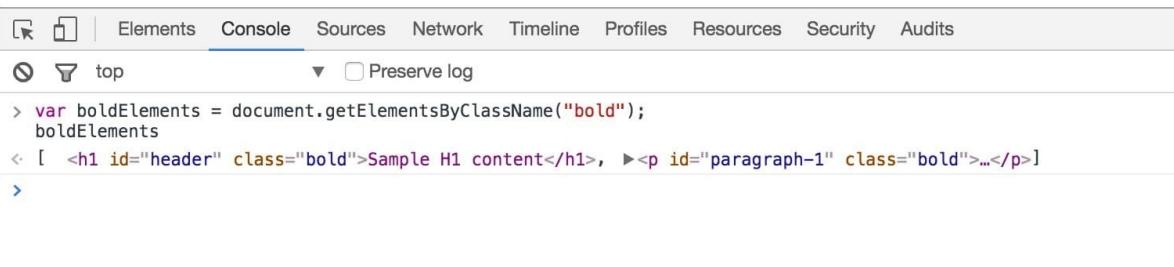
If you run the above on the console tab, you will get something like this:

# Sample H1 content

This is a *paragraph* inside the body.

This is the second paragraph

```
Hello
```



```
Elements Console Sources Network Timeline Profiles Resources Security Audits
top ▾ □ Preserve log
> var boldElements = document.getElementsByClassName("bold");
boldElements
< [ <h1 id="header" class="bold">Sample H1 content</h1>, ><p id="paragraph-1" class="bold">...</p> ]
>
```

Selecting Elements by CSS Class Name

Again, the result of `getElementsByClassName()` is an array, because there might be many elements in the HTML document having the same class.

## Selecting By any CSS Selector

Finally, we will see how we can select an element by any CSS selector. The method that we can use is `querySelectorAll()`. It takes as argument any CSS selector.

Let's run the following command on console:

```
1 var elementsWithValue = document.querySelectorAll("[value=Hello]");
2 elementsWithValue
```

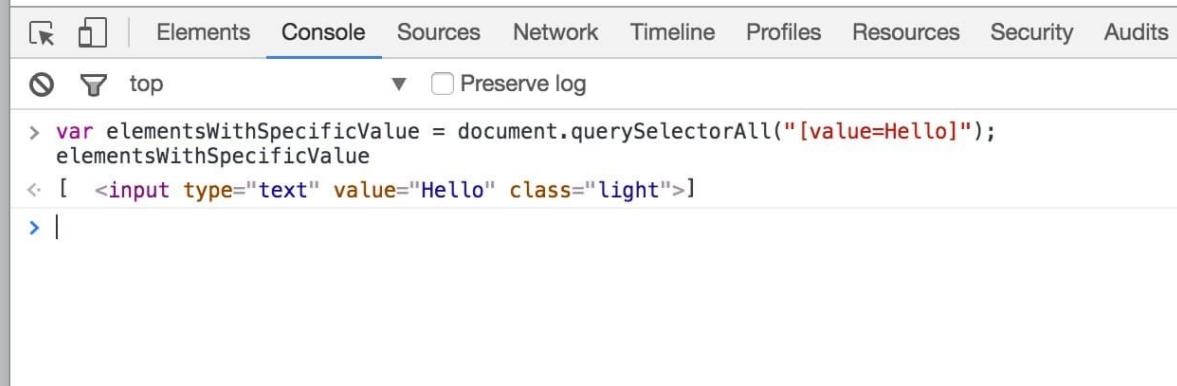
(the above code snippet online)

# Sample H1 content

This is a *paragraph* inside the body.

This is the second paragraph

```
Hello
```



The screenshot shows the Chrome DevTools interface with the 'Console' tab selected. The console output displays the following code and its execution results:

```
var elementsWithValue = document.querySelectorAll("[value=Hello]");
elementsWithValue
< [ <input type="text" value="Hello" class="light"> ]
```

## Selecting Elements Using CSS Selector

As you can see above, the `querySelectorAll("[value=Hello]");` returns all the elements that have the attribute `value` with the value "Hello". It returns an array with one element, because it is only one element with that attribute and that value.

## Element Content As HTML

You can dynamically set the content of an HTML element using the `innerHTML` property. The `innerHTML` can be assigned any valid HTML content. It will be parsed and the result will be presented by the browser.

Let's see the following example. Assume that we have following HTML page:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Using innerHTML - Example</title>
5     <link href="assets/stylesheets/main.css" rel="stylesheet" type="text/css">
6   </head>
7
8   <body>
9     <form>
10    <div id="form-input-container">
11
12    </div>
13  </form>
14
15  <script src="assets/javascripts/using-inner-html.js"></script>
16 </body>
17 </html>
```

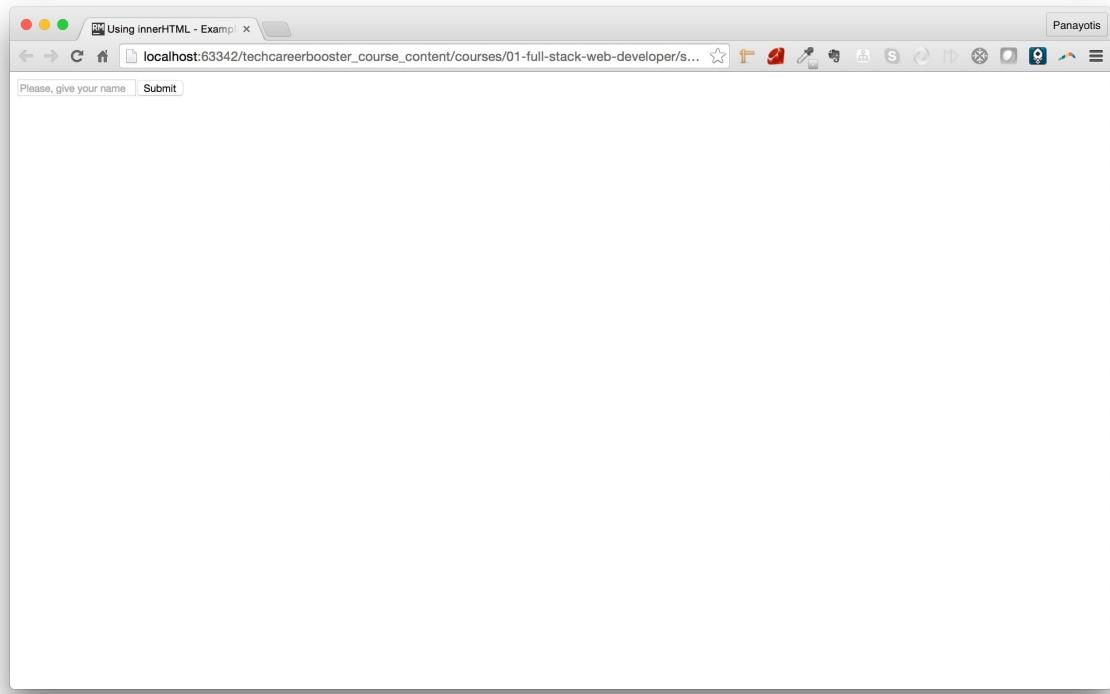
(the above code snippet online)

with the accompanying assets/javascripts/using-inner-html.js file being:

```
1 var formInputContainer = document.getElementById('form-input-container');
2 formInputContainer.innerHTML = '<input type="text" name="name" placeholder="Please give your name"/><button type="submit">Submit</button>';
```

(the above code snippet online)

If you load the above page on your browser, you will see this:



### Using innerHTML to dynamically build the content of the page

Although you can see the input and the button controls, these are not there in the source code of the HTML page. They are being added dynamically, at the end of rendering the HTML page content, with the use of the JavaScript command:

```
1 formInputContainer.innerHTML = '<input type="text" name="name" placeholder="Please,\n2 e, give your name"/><button type="submit">Submit</button>';
```

(the above code snippet online)

A HTML string literal ('<input type="text" name="name" placeholder="Please, give your name"/><button type="submit">Submit</button>') is assigned as value to the property of the element formInputContainer. This element has been located with the help of the getElementById() method.

## Tasks and Quizzes

**Before you continue, you may want to know that:** You can sign up to [Tech Career Booster](#) and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

## Quiz

Please, answer the following questions:

**Question #1 - What does DOM stand for?***Multiple Choice*

1. Data On the Motherboard
2. Document Object Model
3. Disk On Module
4. Date Of Manufacture

**Question #2 - Which is the variable that allows you to have access to DOM programmatically?***Fill In The Blanks*

---

**Question #3 - Which is the method that we use to select an element by id?***Multiple Choice*

1. document.getElementsByIds()
2. document.getFirstElementById()
3. getElementsById()
4. document.getElementById()

**Question #4 - Which is the method that we use to select elements by tag?***Multiple Choice*

1. document.getElementsByTagName()
2. document.getElementByTagNames()
3. document.getElementByTagsName()
4. document.getElementByTagsNames()

**Question #5 - Which is the method that we use to select elements by class name?***Multiple Choice*

1. document.getElementByClassName()
2. document.getElementsByClassName()
3. document.getElementsByClassNames()
4. document.getElementByClassNames()

**Question #6 - Which is the method that we use to select elements by a CSS selector?**

1. document.querySelectByCSS()
2. document.getSelectorAll()
3. document.querySelectorAll()
4. document.queryByCSS()

**Question #7 - The method `document.getElementById()` returns...***Multiple Choice*

1. An array of elements matching the id given as argument.
2. A specific element matching the id given as argument.
3. An array of elements matching the ids given as argument.

**Question #8 - The method `document.getElementsByTagName()` returns...***Multiple Choice*

1. An array of elements matching the tag name given as argument.
2. A specific element matching the tag name given as argument.
3. An array of elements matching the tag names given as argument.

**Question #9 - The method `document.getElementsByClassName()` returns...***Multiple Choice*

1. An array of elements matching the class name given as argument.
2. A specific element matching the class name given as argument.
3. An array of elements matching the class names given as argument.

**Question #10 - The method `document.querySelectorAll()` returns...***Multiple Choice*

1. An array of elements matching the CSS selector given as argument.
2. A specific element matching the CSS selector given as argument.
3. An array of elements matching the CSS selector given as argument.

## Task

Write a page the source code of which is the following:

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Task HTML Document</title>
5          <link href="assets/stylesheets/main.css" rel="stylesheet" type="text/css">
6      </head>
7      <body>
8          <form id="form-with-countries-select-box">
9
10         </form>
11
12         <script src="assets/javascripts/task.js"></script>
13     </body>
14 </html>
```

You need to fill in the content of the `assets/javascripts/task.js` file with JavaScript code that will fill in the HTML content of the `form` with id `form-with-countries-select-box`. The HTML content of this form should be a single select box with 4 countries to select from. Also, it should have a submit button with label `Save`.

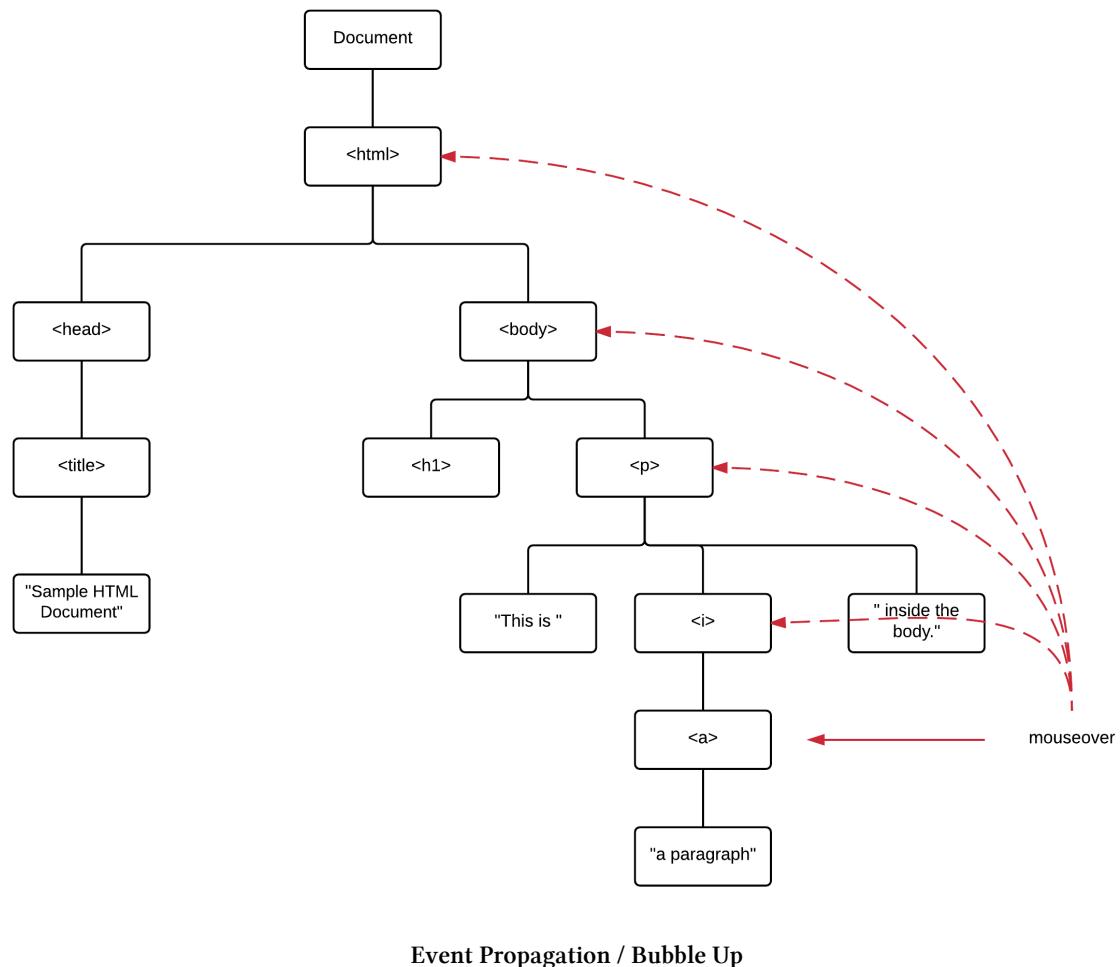
**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

# 11 - Introduction to Event Handling

## Summary

In this chapter, dynamics of our page are getting more interesting. We will teach you how you can respond to various events. For example, how you can write JavaScript code that will respond to user click events.

Also, we are going to teach you how events propagate or bubble up:



How you can cancel an event and how you can handle it on multiple DOM levels.

You will create interesting applications that respond to mouse over or blur and focus events.

Applications like this:

[How Application Responds to Various Events](#)

## Learning Goals

1. Learn basic definitions around events and event handling.

1. Event-driven Programming.
  2. Event Type.
  3. Event Target.
  4. Event Handler or Event Listener.
  5. Event Object.
  6. Event Propagation.
  7. Default Actions.
  8. Cancelling an Event.
2. Learn how you can register an event handler.
    1. Setting event handling property.
    2. Attaching multiple event handlers on an event target.
  3. Learn about Event Propagation and Bubbling.
  4. Learn how you can stop the Event Propagation.

Initially, let's go through some definitions around events and event handling.

## Event-driven Programming

Client-side JavaScript programs use an asynchronous event-driven programming model. In this style of programming, the web browser generates an event whenever something interesting happens to the document or browser or to some element or object associated with it. For example, the web browser generates an event when it finishes loading a document, when the user moves the mouse over a hyperlink, or when the user strikes a key on the keyboard. If a JavaScript application cares about a particular type of event, it can register one or more functions to be invoked when events of that type occur. These functions are called event handlers.

## Event Type

The *event type* is a string that specifies the kind of event that occurred. The type “mousemove” for example, indicates that the user has moved their mouse pointer. The type “keydown” means that a key on the keyboard has been pushed down. The event type is sometimes called *event name*.

## Event Target

The *event target* is the object on which the event occurred or with which the event is associated. When we speak of an event, we must specify both the type and the target. A load event on a Window, for example, or a click event on a <button> Element. Window, Document, and Element objects are the most common event targets in client-side JavaScript applications, but some events are triggered on other kinds of objects.

## Event Handler or Event Listener

The *event handler* or *event listener* is a function that handles or responds to an event. Applications register their event handler functions with the web browser, specifying an event type and an event target. When an event of the specified type occurs on the specified target, the browser invokes the handler. When event handlers are invoked for an object, we sometimes say that the browser has *fired*, *triggered*, or *dispatched* the event. There are a number of ways to register event handlers.

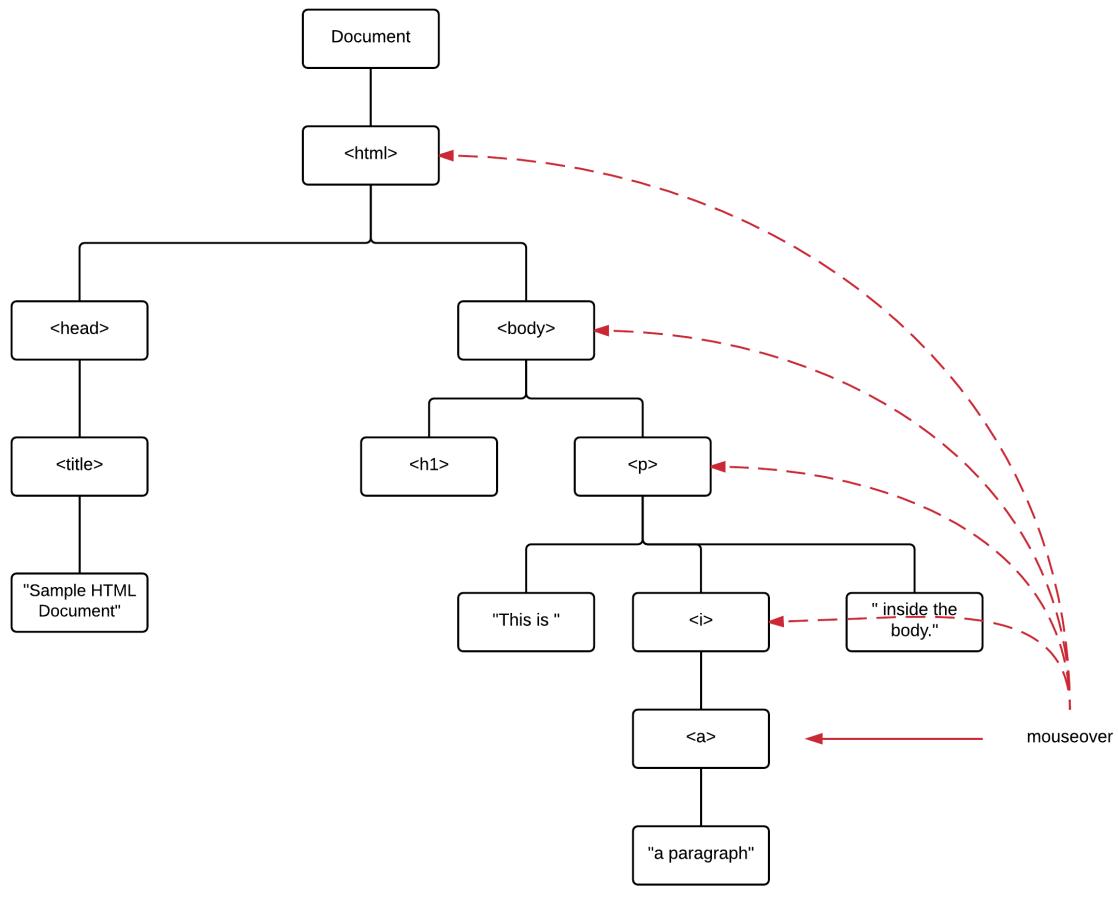
## Event Object

An *event object* is an object that is associated with a particular event and contains details about that event. Event objects are passed as an argument to the event handler function. All event objects have a *type* property that specifies the event type and a *target* property that specifies the event target. Each event type defines a set of properties for its associated event object. The object associated with a mouse event, for example, includes the coordinates of the mouse pointer. Or the object associated with a keyboard event contains details about the key that was pressed and the modifier keys that were held down. Many event types define only a few standard properties — such as *type* and *target* — and do not carry much other useful information. For those events it is the simple occurrence of the event, not the event details, that matter.

## Event Propagation

*Event propagation* is the process by which the browser decides which objects to trigger event handlers on. For events that are specific to a single object (such as the load event on the Window object), no propagation is required. When certain kinds of events occur on document elements, however, they propagate or “bubble” up the document tree. If the user moves the mouse over a hyperlink, the `mousemove` event is first fired on the `<a>` element that defines that link. Then it is fired on the containing elements: perhaps a `<p>` element, a `<div>` element, and the Document object itself.

It is sometimes more convenient to register a single event handler on a Document or other container element than to register handlers on each individual element you’re interested in. An event handler can stop the propagation of an event, so that it will not continue to bubble and will not trigger handlers on containing elements.



#### Event Propagation / Bubble Up

As you can see on the above example, when a `mouseover` event occurs on a link `<a>`, then this will be handled by any event handler registered for that event type on that event target. If there is no event handler, or if there is but event handler lets the event bubble up, then event will bubble up to the parent element. And so on. We will see and practice this event bubbling later in this chapter.

## Default Actions and Cancelling an Event

Some events have default actions associated with them. When a `click` event occurs on a hyperlink, for example, the default action is for the browser to follow the link and load a new page. Event handlers can prevent this default action by returning an appropriate value, or invoking a method of the event object, or by setting a property of the event object. This is sometimes called *cancelling the event*.

Having done this introduction to core definitions, we now proceed in learning how to register and implement an event handler. Note that this chapter is only an introduction to event handling with core JavaScript. In jQuery chapter, we will learn how to handle events using jQuery methods.

## Registering Event Handlers

There are basically two methods to register an event handler.

1. By setting the event function handler as a value for a property of the event target. The name of the property you set the value for starts with on and is followed by the event name. For example, if you want to register an event handler for the `mouseover` event, you need to set the event handler as value of the property `onmouseover`.
2. By calling the standard method `addEventListener()`.

## Setting an event handler as value of event handling property

Let's see the first method. Let's try the following example:

The HTML page:

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Attaching Event Handlers On Properties</title>
7     <script src="assets/javascripts/main.js"></script>
8   </head>
9   <body>
10    <form>
11      <input type="text" name="firstName"/>
12      <button type="submit" id="save-button">Save</button>
13    </form>
14  </body>
15 </html>
```

(the above code snippet online)

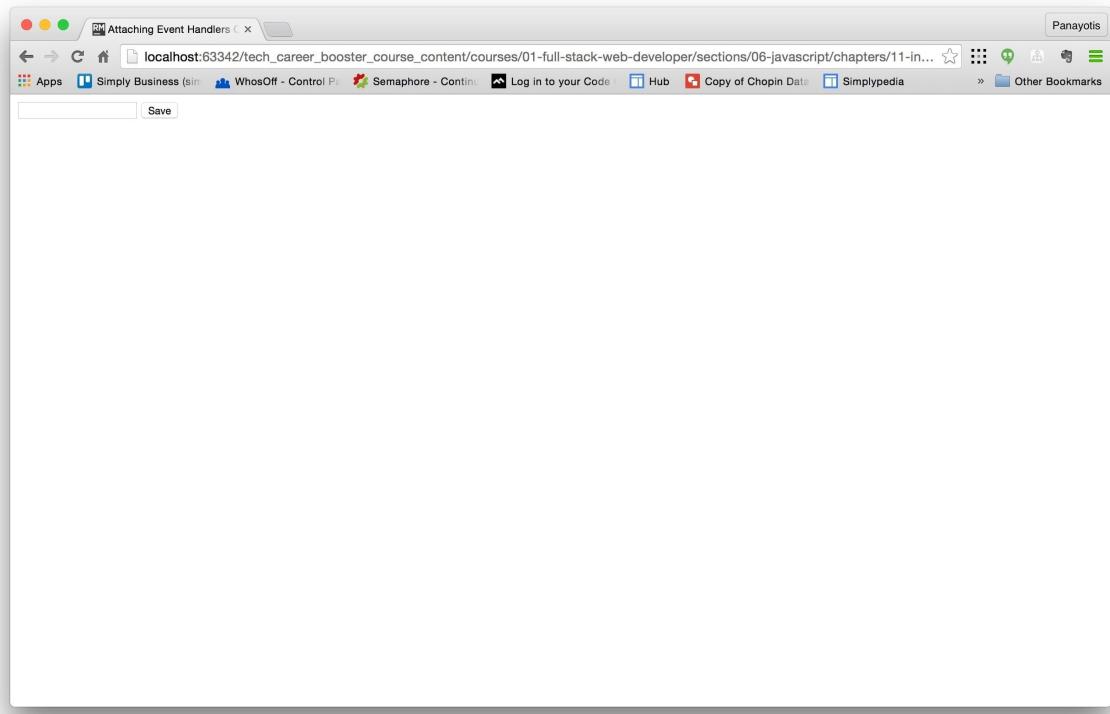
and the corresponding `assets/javascripts/main.js` with the following content:

```

1 window.onload = function() {
2
3   var formButton = document.getElementById('save-button');
4
5   formButton.onclick = function() {
6     var answer = confirm('Are you sure that you want to save this?');
7     if (answer) {
8       formButton.innerHTML = "Saved";
9       formButton.disabled = true;
10    }
11    return false; // Does let the event bubble up and does not let run the de\
12 fault handler.
13  };
14
15};
```

(the above code snippet online)

If you load the HTML page on your browser, you will see this:



#### Form With an Input Text And a Button

You can see the input text and the button.

The JavaScript code in the `assets/javascript/main.js` file does the following:

1. Registers an event handler for the event `load` on event target `window`. This basically means that this handler is going to fire after the page content has been loaded by the browser.
2. When the ‘`load`’ event handler fires, it tries to locate the form button with the following piece of code `var formButton = document.getElementById('save-button');` which is pretty straightforward, if one sees that the `<button>` element on the page has the id `save-button`.
3. Then it registers a handler on the `click` event for the button target. The handler asks the user whether he wants to save the data or not. If the user answers “Yes/Ok”, then we change the label of the button from “Save” to “Saved” and we disable the button by setting its `disabled` property to `true`.
4. In any case (with answer “Yes” or with answer “No”), we return `false` from the `click` handler indicating to the browser that it should not invoke the default action, and that we handled the event. In other words, the event does not need further handling.

If you load the page and click on the button, you will experience something like this:

[Attaching Events on Event Handling Properties](#)

**Important:** It is very important to code so that your event handler registers when the page has loaded. That's why, we attach the event handler to button from inside another event handler that fires when the "load" event on "window" target takes place. Otherwise, the element `save-button` might not have been loaded yet. Attaching the code on `load` event of `window`, you make sure that the elements of your document exist at the time the code is executed.

## Using `addEventListener()`

The problem with the above technique to attach event handlers is that each target can only have at most one handler for the same event type. If you want to have multiple event handlers for the same event type and target, you can use the `addEventListener()` method.

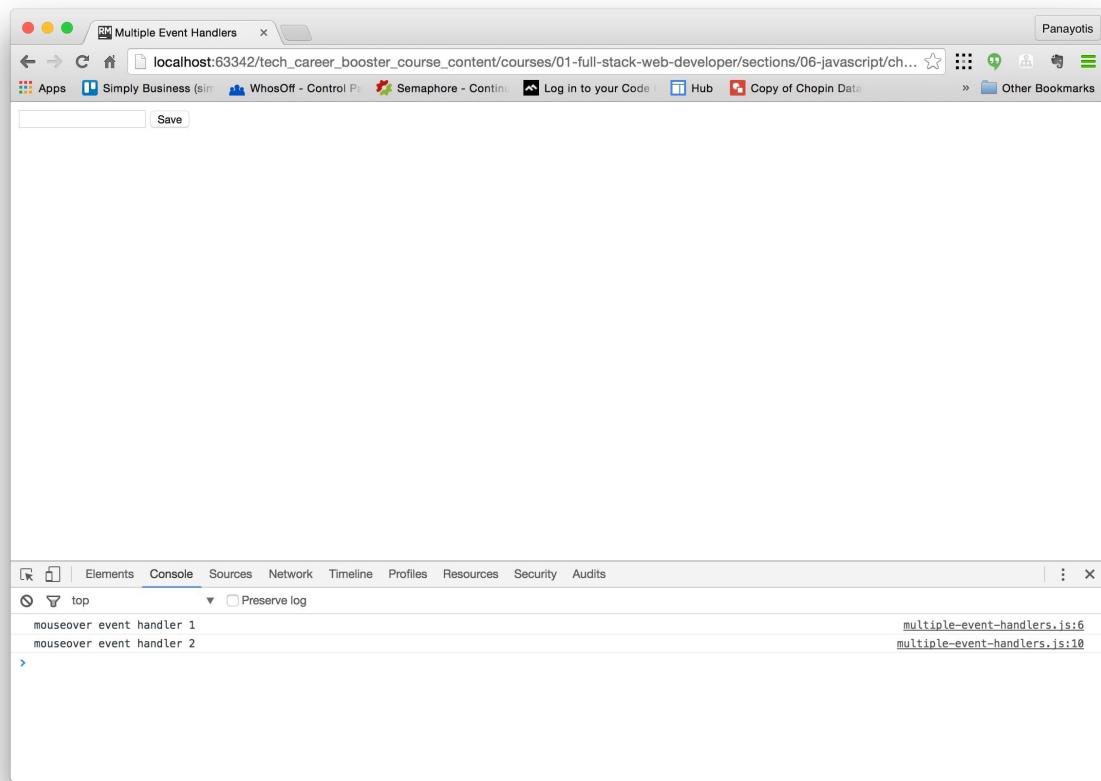
Let's try an example:

Assume the same HTML page as before, but we now link to file `assets/javascripts/multiple-event-handlers.js`:

```
1 window.onload = function () {  
2  
3     var formButton = document.getElementById('save-button');  
4  
5     formButton.addEventListener('mouseover', function () {  
6         console.log('mouseover event handler 1');  
7     });  
8  
9     formButton.addEventListener('mouseover', function () {  
10        console.log('mouseover event handler 2');  
11    });  
12};  
13};
```

(the above code snippet online)

If you load the HTML page on your browser, and then you move your mouse over the button, you will see two messages printed on the console:



### Two Event Handlers Firing

As you can see, the two event handlers are firing, each one printing their own message on the console. The two event handlers have been registered using the `addEventListener()` method.

The advantage of this technique is that you can attach as many handlers as you like. You are not limited to just one.

Note that if you have both an event handler attached to an event property for a particular event target, and you have also added an event listener for the particular event type and event target combination, then, browser is going to fire, first, the event handler on the event property and then the event handler (or handlers) attached using the `addEventListener()` method.

## Event Propagation

When the target of an event is the `window` object, the browser responds to an event simply by invoking the appropriate handlers on that object. When the event target is a document or document element, however, the situation is more complicated.

After the event handlers registered on the target element are invoked, most events “bubble” up the DOM tree. The event handlers of the target’s parent are invoked. Then the handlers registered on the target’s grandparent are invoked. This continues up to the `document` object, and then beyond to the `window` object. Event bubbling provides an alternative to registering handlers on lots of individual document elements: instead you can register a single handler on a common ancestor

element and handle events there. For example, you might register an `change` handler on a `form` element, instead of registering a `change` handler for every element in the form.

Most events that occur on document elements bubble. Notable exceptions are the `focus`, `blur`, and `scroll` events. The `load` event on document elements bubbles, but it stops bubbling at the `document` object and does not propagate on to the `window` object. The `load` event of the `window` object is triggered only when the entire document has been loaded.

Let's do an example in order to see how the bubbling works. Let's suppose that we have the following page:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Bubble Demo</title>
5     <script src="assets/javascripts/bubble.js"></script>
6   </head>
7   <body>
8     <div>
9       <form>
10      <button type="submit">Do Something</button>
11    </form>
12  </div>
13 </body>
14 </html>
```

(the above code snippet online)

It is a very simple page that is loading the JavaScript file `assets/javascripts/bubble.js`. Its content is a `div` that has a child a `form` element, which, in turn, has as child a `button` element.

The content of the JavaScript file is the following:

```
1 window.onload = function() {
2   var button = document.getElementsByTagName("button")[0];
3   button.addEventListener('click', function() {
4     alert('I am the button and I received a click event');
5   });
6
7   var form = document.getElementsByTagName("form")[0];
8   form.addEventListener('click', function() {
9     alert('I am the form and I received a click event');
10  });
11
12  var div = document.getElementsByTagName("div")[0];
13  div.addEventListener('click', function() {
14    alert('I am the div and I received a click event');
15  });
}
```

```

16
17   document.body.addEventListener('click', function() {
18     alert('I am the body and I received a click event');
19   });
20
21   document.addEventListener('click', function() {
22     alert('I am the document and I received a click event');
23   });
24
25   this.addEventListener('click', function() {
26     alert('I am the window and I received a click event');
27   });
28 }

```

(the above code snippet online)

It may look complex, but, if you read the code carefully, you will see:

1. We initially attach a handler on the `window` target for events of type `load`. So, we make sure that our function handler is being called when all the elements of the HTML document have been loaded and are part of the DOM tree.
2. We are attaching an event handler for event `click` (which means clicking once the mouse button), onto the event target button. The handler is basically an alert. The alert message only confirms that it is an alert triggered when the button attached `click`-handler has been fired.
3. We attach a similar `click` event handler to each one of the ancestors of the `button` element. Hence
  1. One handler for the `form` element.
  2. One handler for the `div` element.
  3. One handler for the `body` element. See here how we access `body` through `document`.
  4. One handler for the `document` element.
  5. One handler for the `window` element. See how we are using `this` keyword to access the current target of the `load` event, and use it inside the function handler. We could equally have used `window.addEventListener(...)` instead of `this.addEventListener(...)`, but we did that with `this` to demonstrate the fact that `this`, inside an event handler function, references the actual target of the event.

So, how do we expect this page to behave, if we loaded it on our browser? If the user clicked the button, then we would expect an alert telling us that the button has been clicked, but then all the rest of the event handlers attached to `button` ancestors, for the same event type, would fire too.

This is how it would work:

### Demo of Event Bubbling

*Try this one:* What would happen if you clicked on the document itself, instead of on the button? Try, in other words, to click on the blank white area of the browser window. Does any event fire? How many?

## Stopping the Event Propagation

Now, let's assume the same example as before, but let's change the handler of the `div` element as follows:

```
1 // ... previous code remains as it was ...
2
3 var div = document.getElementsByTagName("div")[0];
4 div.addEventListener('click', function(event) {
5     alert('I am the div and I received a click event');
6     event.stopPropagation();
7 });
8
9 // ... next code remains as it was ...
```

(the above code snippet online)

If you reload the page (in order for the update JavaScript code to be in effect), and then you click on the button, you will see the alerts going on up the ancestor tree being fired, but only up to, and including, the `div` element. The `body`, `document` and `window` event handlers do not fire.

## Tasks and Quizzes

**Before you continue, you may want to know that:** You can sign up to [Tech Career Booster](#) and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

### Tasks

(1) You need to create an HTML page that has two buttons. Button 1 and Button 2. Whenever user clicks on Button 1, then Button 1 becomes a big button. Whenever user clicks on Button 2, then Button 2 becomes the big button. Watch the following video in order to understand what the page should look like and how the button should be responding to events:

#### Task For Event Handling - Buttons

As you can see, the buttons, initially have same width. It is 110px wide. When they go large, their width is 300px.

**Hint:** When you want to change the width of an element using JavaScript, you can set a value to `style` property of the element. The value can be any valid style property value. For example: "width: 300px;".

**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

(2) Task with ‘blur’ and ‘focus’ events. You need to create a page in which there are two text input fields and a button. When the user clicks on any input in order to start typing a value, then the border of that input should become blue. When the user leaves an input to either go to the other input or to the button, the input border should become either red or green. It becomes red if the user has not given any value. It becomes green if the user has given some value. Watch the following video in order to understand what we mean.

### Task 2 - Demo

Here are some hints:

1. The event that is triggered when user clicks on an input control is the `focus` event.
2. The event that is triggered when user leaves an input control to move out of it, it is called `blur` event.
3. To set the border of the input control, just set a value to its `style` property, like `input.style = "border: 1px solid red;"`.

**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

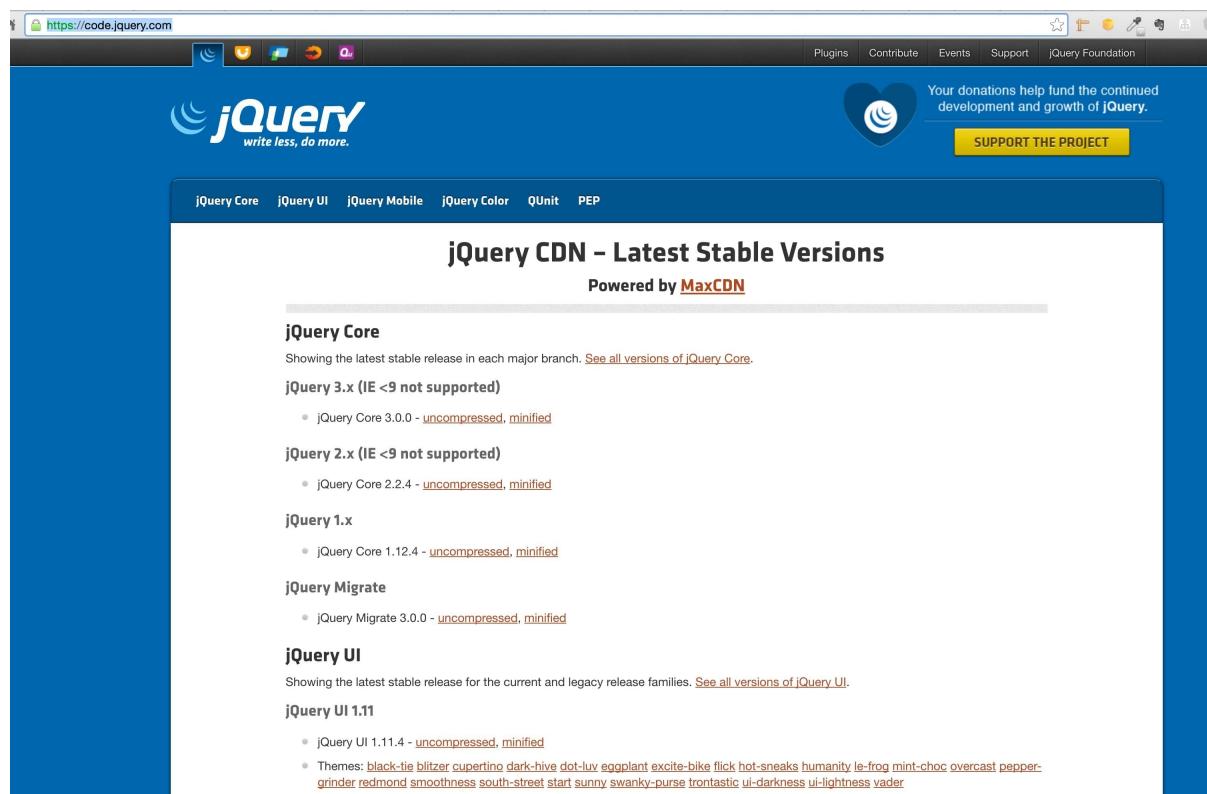
## 12 - jQuery Basics

### Summary

jQuery is one of the most popular JavaScript libraries used today. Its main purpose is to make the front-end developers life much easier, by hiding the complexities related to browser incompatibilities.

This chapter is an introduction to jQuery. However, you are going to learn lots of new and interesting stuff.

First we will learn how to download or reference jQuery.



### Download or Reference jQuery

You will then learn how to integrate that into your HTML page.

jQuery will give you lots of power to create dynamic HTML pages that offer rich functionality to end-user.

For example, you will be able to attach validations to an HTML form.

#### HTML Page With Form Validations

You will learn how to dynamically change the properties of the elements of your HTML document, like here:

#### App To Change Background Color

or like here:

## Adding and Removing Classes

You will also learn how to add and remove elements dynamically like here:

### Adding and Removing Elements Dynamically

Finally, you will be requested to create a UFO shooting game like this:

#### Task - Game: Shooting a UFO

## Learning Goals

1. Learn how to include jQuery into your HTML document.
2. Learn how to reference jQuery using a CDN (Content Delivery Network).
3. Learn to reference jQuery global function with `jQuery` or `$`.
4. Learn how to attach a handler on the document ready event.
5. Learn how to attach a handler for the `click` event that occurs on a button.
6. Learn how you can select elements using jQuery.
7. Learn how you can get the value of an input control using `.val()` method.
8. Learn how to prevent the default behaviour of a button when you attach your own handler.
9. Learn about the `on()` method.
10. Learn how the jQuery collections respond to `length`.
11. Learn how you can get or set the HTML content of an HTML element.
12. Learn how to get and set HTML attributes and their values.
13. Learn how the jQuery collections refer to a snapshot of the HTML document.
14. Learn about how you can remove HTML elements from your HTML document.
15. Learn about how you can insert HTML elements dynamically.
16. Learn how to get and set CSS attributes.
17. Learn how to get and set CSS classes.
18. Learn how to get and set HTML form values.
19. Learn how you can execute a function on every matching element.
20. Learn how you can fade in and fade out an particular element.
21. Learn how to execute some code at the end of the fade out process.
22. Learn about the strict mode.

## Introduction

JavaScript has a very simple core API. On the other hand, the client-side API, the one used to write applications executed by the browser, is very complicated, and the most important problem, is that there are a lot of incompatibilities between browser brands.

Hence, client-side JavaScript developers prefer to use a framework or JavaScript libraries that have been developed to hide the browser incompatibilities and they offer enriched functionality.

Such a JavaScript library is [jQuery](#).

jQuery makes it easy to find the elements of a document that you care about and then manipulate those elements by adding content, editing HTML attributes and CSS properties, defining event

handlers, and performing animations. It also has Ajax utilities for dynamically making HTTP requests and general-purpose utility functions for working with objects and arrays.

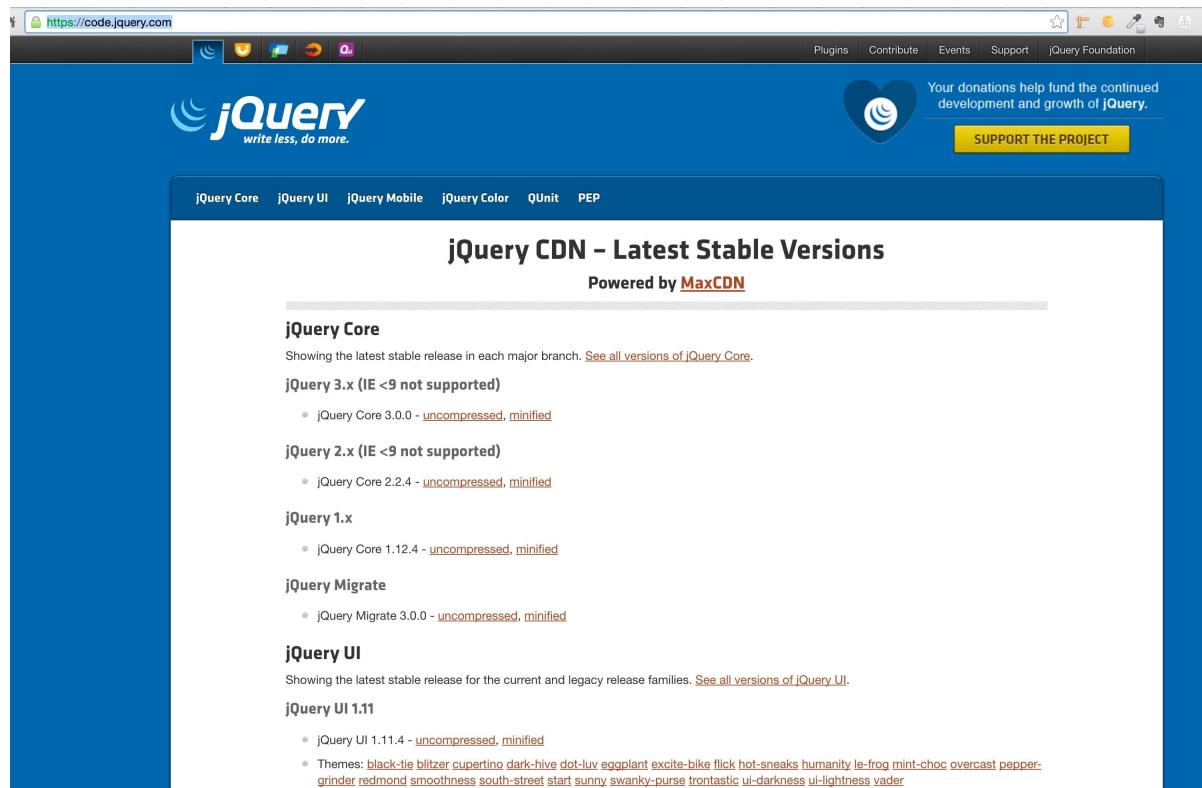
As its name implies, jQuery is based on queries. It allows you to define query criteria, using CSS selectors, that return a specific set of HTML elements from your document. Then, on the selected HTML elements, you can call various methods in order to manipulate the particular elements.

## Including jQuery to your HTML document

jQuery is a set of JavaScript functions. Like any JavaScript library, in order for you to use it, you have to include it inside your HTML document. This is usually done at the `<head>` section of the document using the `<script>` element.

You can download jQuery locally at your machine and then include the local files. Or you can have the `<script>` element reference the jQuery over the Internet, using a CDN (Content Delivery Network) reference.

On this page here, you can read how you can reference jQuery via a CDN. There, you will read that you have to go here: <https://code.jquery.com/> in order to get the URL for the jQuery version that you are interested in.



jQuery CDN Page

We will pick up the following URL to reference jQuery:

## jQuery 2.x (IE <9 not supported)

- jQuery Core 2.2.4 - [uncompressed](#), [minified](#)

## jQuery 1.x

- jQuery Core 1.12.4 - [uncompressed](#), [minified](#)
- [jQuery 2 Minified Link](#)

It is jQuery version 2.X which is the most mature and widely used as of today. If you click on this link you will see this:



Modal Dialog With Code For jQuery Integration

As you can read on the above modal, you need to copy the code that you are being presented with. Copy to your clipboard and than paste into your HTML page. Here is how HTML page should be:

```

1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="utf-8">
5          <meta name="viewport" content="width=device-width, initial-scale=1.0">
6          <title>jQuery Integration</title>
7          <script src="https://code.jquery.com/jquery-2.2.4.min.js"
8                  integrity="sha256-Bbhd1vQf/xTY9gja0Dq3HiwQF8LaCRTxZKRutelT44="
9                  crossorigin="anonymous"></script>
10     </head>
11     <body>
12
13     </body>
14 </html>
```

(the above code snippet online)

And now, you are ready to use jQuery library methods and functions. Let's see how.

## jQuery and \$

The jQuery library defines a single global function with name `jQuery`. Also, there is a shortcut name `$` that you can use instead.

## Selecting And Actioning on HTML Elements

Let's see how we can use the `$` function to select HTML elements on our HTML page and act upon them. Let's start with the following HTML page content:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>jQuery Integration</title>
7     <script src="https://code.jquery.com/jquery-2.2.4.min.js"
8           integrity="sha256-BbhdlvQf/xTY9gja0Dq3HiwQF8LaCRTXxZKRute1T44="
9           crossorigin="anonymous"></script>
10    <script src="assets/javascripts/main.js"></script>
11  </head>
12  <body>
13
14  </body>
15 </html>
```

(the above code snippet online)

As you can see above, we have included the `<script>` line that references our own JavaScript code, stored inside `assets/javascripts/main.js` file. This file is currently empty, but we will fill that in with JavaScript code later on.

**Important:** You need to first reference and load jQuery libraries and then your own custom JavaScript code that uses jQuery.

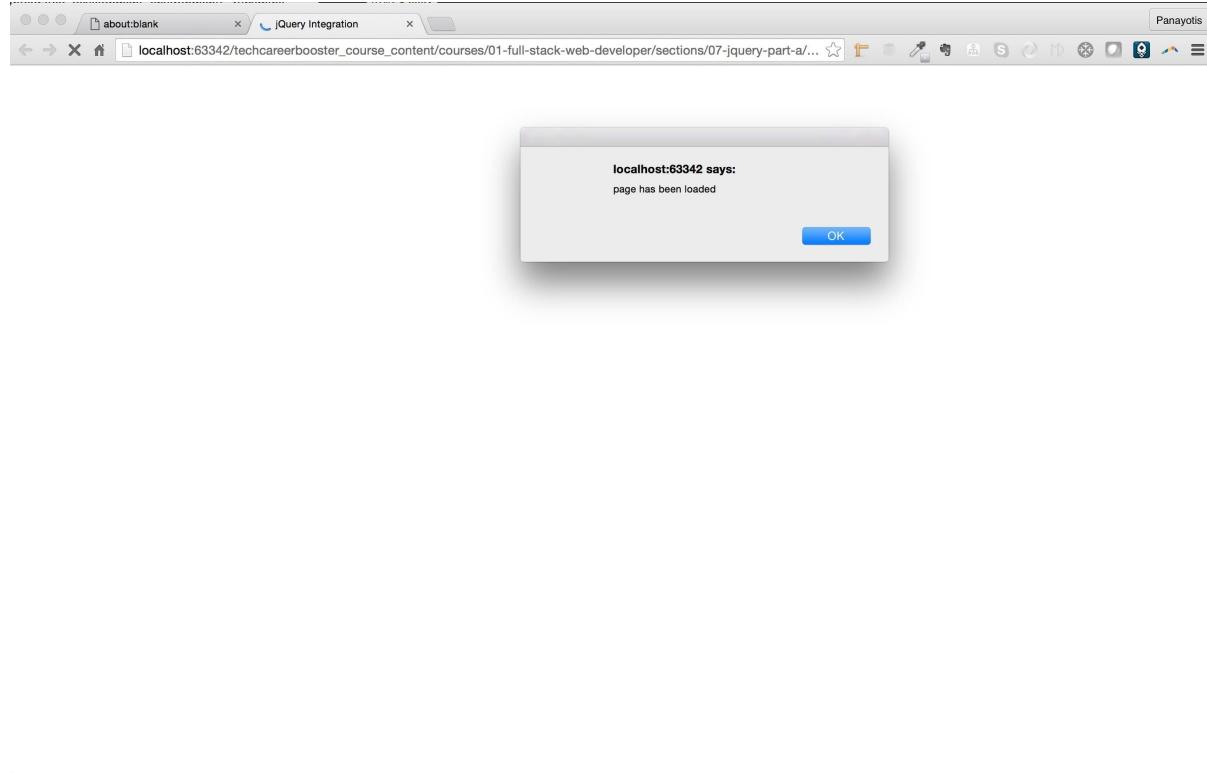
Now, inside `assets/javascripts/main.js` write the following JavaScript code:

```
1 $(document).ready(function() {
2   alert('page has been loaded');
3 });
```

(the above code snippet online)

This calls the function `$` giving as argument the global variable `document`, hence selecting the document to act upon. And then registers a function handler for the event `ready`. Hence, inside the function handler we write the JavaScript code that we want to be executed when the document will be ready, a.k.a. all the HTML page elements have been loaded by the browser.

If you save the above and you load the page on your browser, you will see this:



#### Alert Displayed When Document Ready, using jQuery

Let's see another example. We enhance our HTML page to have a form with a text input and a button. When the user clicks on the button, we will check the content of the input box. If it is empty, we will display an alert to the user and we will not submit the form. If the input box value is not blank, then we will let the form be submitted.

Here is the HTML code with the form:

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="utf-8">
5          <meta name="viewport" content="width=device-width, initial-scale=1.0">
6          <title>jQuery Integration</title>
7          <script src="https://code.jquery.com/jquery-2.2.4.min.js"
8                  integrity="sha256-Bbhd1vQf/xTY9gja0Dq3HiwQF8LaCRTXxZKRutelT44="
9                  crossorigin="anonymous"></script>
10         <script src="assets/javascripts/main.js"></script>
11     </head>
12     <body>
13         <form>
14             <label for="first-name">First Name:</label><br/>
15             <input type="text" name="first-name" id="first-name"/><br/>
16             <button type="submit" id="submit-form-button">Save</button>
```

```

17      </form>
18  </body>
19 </html>
```

(the above code snippet online)

And here is the updated assets/javascripts/main.js file:

```

1 $(document).ready(function() {
2     // Install a handler for the click event on the
3     // submit form button.
4     $('#submit-form-button').on('click', function(){
5         // get access to the input control
6         var $inputControl = $('#first-name');
7         // get the value of the input control
8         var value = $inputControl.val();
9         if (value.trim() === "") {
10             alert("You need to provide your first name");
11             return false; // this "return false" here will prevent the default bu\
12 tton "click" handler from being fired
13         }
14     });
15 });
```

(the above code snippet online)

If you save the above and load the page on your browser, you will experience this:

### jQuery Example of Form Validation

Here is the list of things you need to be aware of in regards to the jQuery code above:

1. Again, all the code is inside the function handler for `ready` on `document` target.
2. With `$('#submit-form-button')` we locate the button element on our form. The selection is done by id, but we can generally use any selector that would select the element(s) we want.
3. The `$(...)` function call returns a jQuery object and hence we can call jQuery methods on it. Generally, all the jQuery API documentation can be found [here](#).
4. On the jQuery object returned by `$('#submit-form-button')` we call the method `on(...)`. The documentation of this method can be found [here](#). You need to start reading the documentation little-by-little. The more you read jQuery documentation the more you will become familiar with its structure and, after lots of practice, it will be easier for you to deal with it.
5. The `on(...)` method call that we use above, installs an event handler on the matching elements the `on(...)` is called on. In other words, the `$('#submit-form-button')` returns a collection of matching elements (1 in our case) and the `on('click', function() { ... })` installs an event handler for `click` event on all those matching elements. Hence, we basically install a `click` handler on the button of the form.

## 6. What does the click handler do?

1. It locates the input control of the form. This is done with another call to `$(....)` that would take as input the CSS selector locating the input control box. It saves the result of this selection to the variable `$inputControl`. It is a common practice to name the variables that store jQuery objects with a name that starts with the character `$`. Hence, the developer knows that he can call jQuery methods on that particular variable.
2. It, then, takes the value of the input control by calling the method `val()` on the jQuery object `$inputControl`. The method `val()` is documented [here](#).
3. The value is saved in the variable with name `value`.
4. Then it calls the method `trim()` which is a core JavaScript string function. It removes any blanks left and right to the payload of the value.
5. It compares the trimmed value against blank string. If the value is blank, it displays an alert message and returns false. The return of false is important, because it will prevent browser from calling the default functionality for form submit buttons, which is to submit the form.

## Queries and Query Results

We have done some selections, queries, in the previous examples. Let's see some more here.

Write the following HTML page:

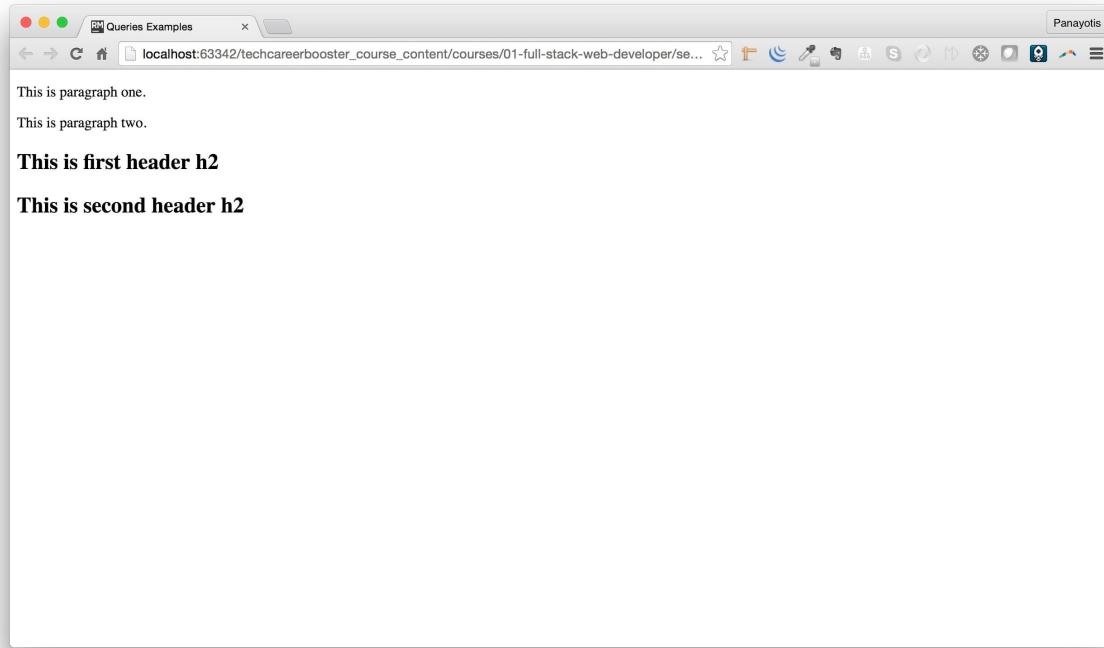
```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Queries Examples</title>
7     <script src="https://code.jquery.com/jquery-2.2.4.min.js"
8           integrity="sha256-BbhdlvQf/xTY9gja0Dq3HiwQF8LaCRTXxZKRutelT44="
9           crossorigin="anonymous"></script>
10    <script src="assets/javascripts/queries.js"></script>
11  </head>
12  <body>
13    <p>
14      This is paragraph one.
15    </p>
16    <p>
17      This is paragraph two.
18    </p>
19    <h2>
20      This is first header h2
21    </h2>
22    <h2>
23      This is second header h2
24    </h2>
```

```
25    </body>
26 </html>
```

(the above code snippet online)

with initially empty assets/javascripts/queries.js file.

If you save the files and load the page on your browser, you will see this:



#### An HTML page to Use with jQuery Queries

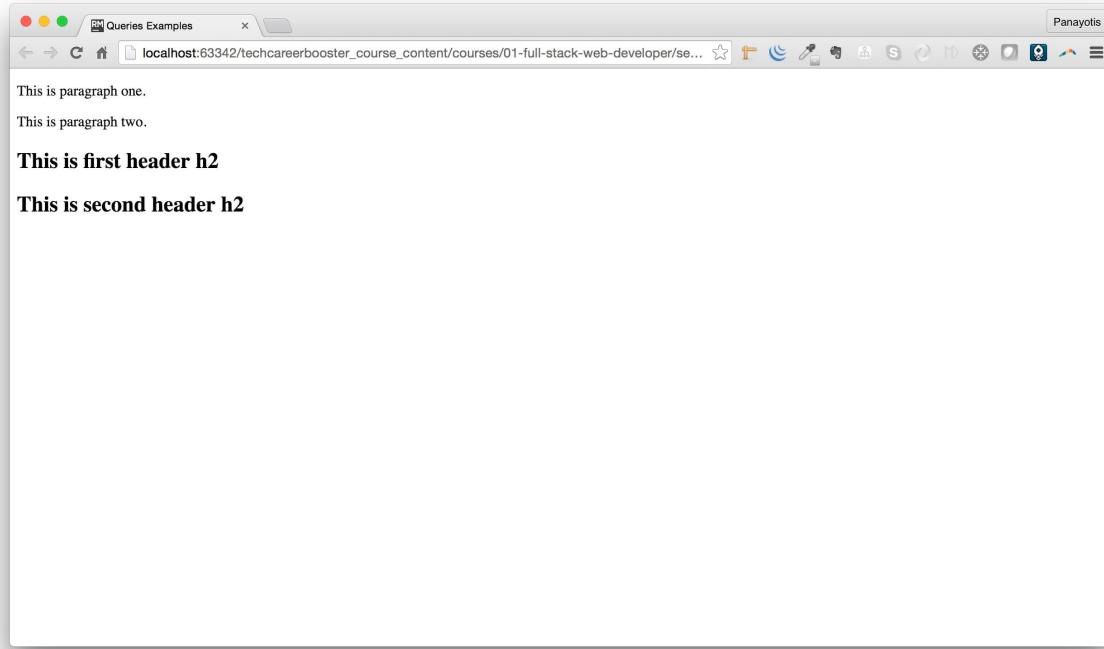
Now, let's write some JavaScript code that would demonstrate selecting elements with jQuery. Update your assets/javascripts/queries.js file as follows:

```
1 $(function() {
2     var $pElements = $('p');
3     console.log("$pElements: > ");
4     console.log($pElements);
5     console.log("$pElements.length: " + $pElements.length);
6     console.log("$pElements[0]: > ");
7     console.log($pElements[0]);
8     console.log("$pElements[1]: > ");
9     console.log($pElements[1]);
10});
```

(the above code snippet online)

If you save the above files and load the page on your browser, you will see this:

*Note:* Make sure that when you load the page you have the developer tools open on the console tab.



### Selecting Multiple Elements

Let me explain to you what we have done and what you see above.

1. The `$(function() { . . . });` is another way to attach a function handler to the `ready` event of the document. So, it is equivalent to  
`$(document).ready(function() { . . . });`
2. We select all the `p` elements with the CSS selector `p`. The jQuery call is `$( 'p' )`. We save the selected or matched elements inside the variable `$pElements`.
3. The `$pElements` is a jQuery collection and responds to `length`. This prints the number of matched elements.
4. The `[index]` operator returns the matched element at `index` position. First one being at position `0`. Second one being at position `1` e.t.c.

**Important:** The result of `[index]` operator on a jQuery collection object is not a jQuery object and you can not call jQuery methods on it. You need to wrap the result to a jQuery object by calling the `$( . . . )`. Let's continue the above example in order to see what I mean:

Let's add the following JavaScript lines inside the function handler:

```
1 var $h2Elements = $('h2');
2 var firstElement = $h2Elements[0];
3 console.log(firstElement.html());
```

(the above code snippet online)

so that the whole assets/javascripts/queries.js file to be:

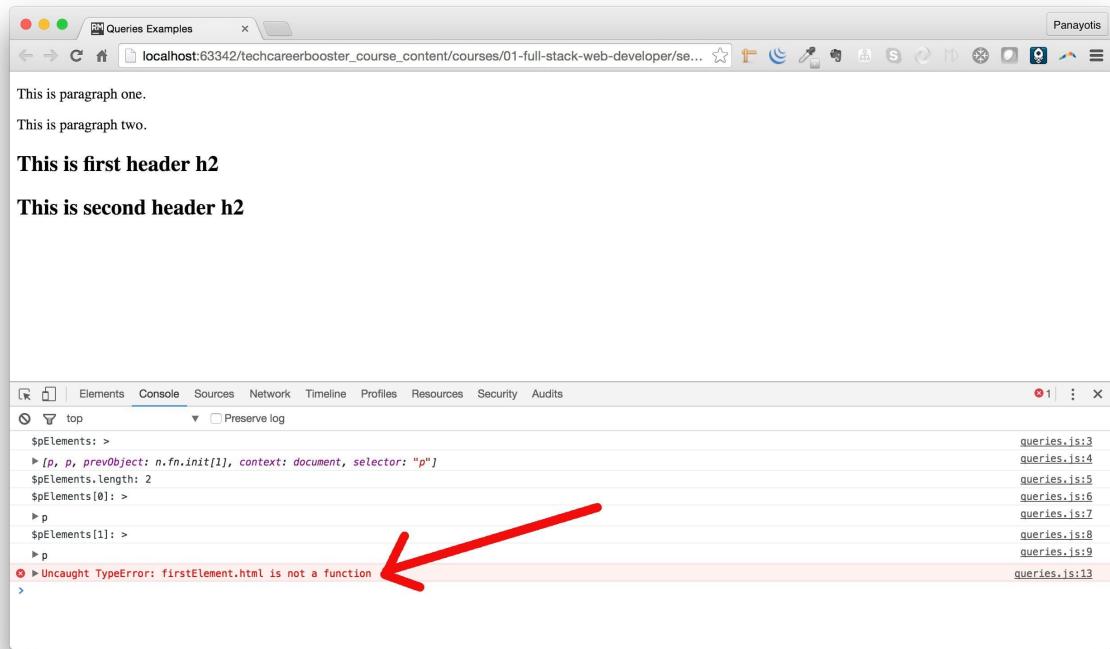
```
1 $(function() {
2     var $pElements = $('p');
3     console.log("$pElements: > ");
4     console.log($pElements);
5     console.log("$pElements.length: " + $pElements.length);
6     console.log("$pElements[0]: > ");
7     console.log($pElements[0]);
8     console.log("$pElements[1]: > ");
9     console.log($pElements[1]);
10
11    var $h2Elements = $('h2');
12    var firstElement = $h2Elements[0];
13    console.log(firstElement.html());
14});
```

(the above code snippet online)

With the last 3 lines,

1. we get the jQuery collection of the h2 elements: `var $h2Elements = $('h2');`
2. then we get access to the first element of the collection: `var firstElement = $h2Elements[0];`
3. then we try to get the h2 element HTML content by calling the jQuery method [.html\(\)](#).

If you save the above and reload the page on your browser (have the Console tab on developer tools open), you will see that an error is printed in the console.



#### Error When Calling jQuery method on non jQuery Object

Although \$h2Elements is a jQuery object (collection), the \$h2Elements[0] is not. Hence, calling the method .html() on that fails with the error that you see on the screenshot above. The \$h2Elements[0] is a normal HTML DOM element. And you can not call jQuery methods on this. You can, however, call the function \$(....) giving as argument the DOM element and get back a jQuery object, on which you can call jQuery methods.

Let's correct the JavaScript content as follows:

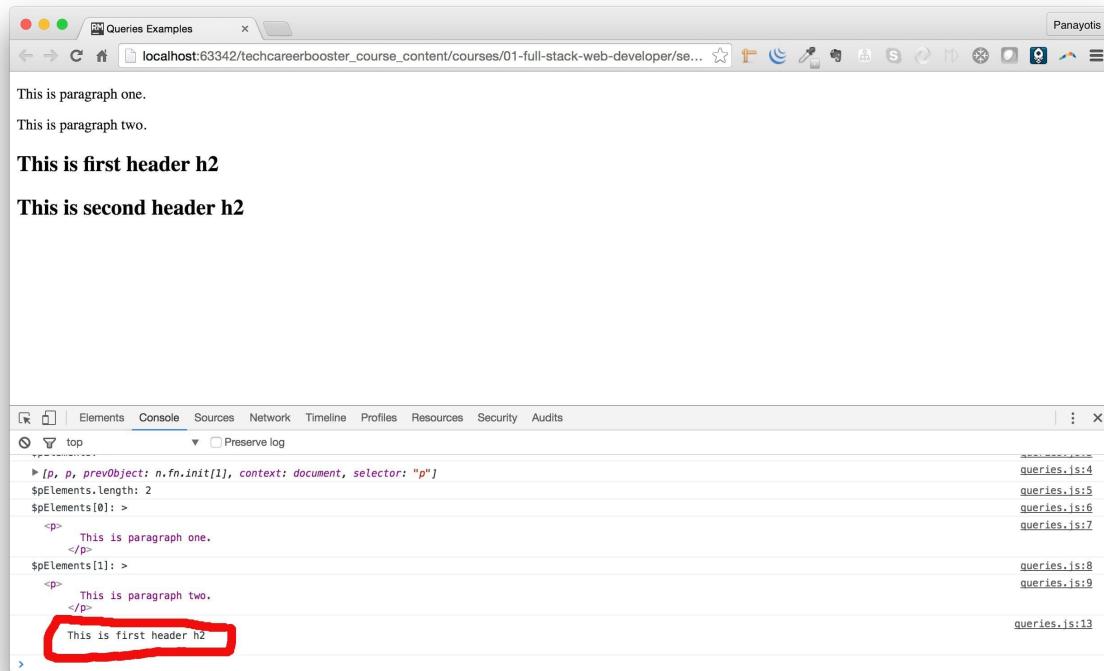
```

1  $(function() {
2      var $pelements = $('p');
3      console.log("$pelements: > ");
4      console.log($pelements);
5      console.log("$pelements.length: " + $pelements.length);
6      console.log("$pelements[0]: > ");
7      console.log($pelements[0]);
8      console.log("$pelements[1]: > ");
9      console.log($pelements[1]);
10
11     var $h2elements = $('h2');
12     var firstElement = $h2elements[0];
13     console.log($(firstElement).html());
14 });

```

(the above code snippet online)

Do you see the \$(firstElement).html() ? This is it. Let's save the files and reload the page on your browser.



### No Errors - Printing the HTML content of First h2

You will not see any errors. Instead, you will see the HTML content of the first h2 element printed in the console.

## Getting and Setting HTML Attributes

The method `.attr()` is used to set a value for a particular attribute on an HTML element. Let's see that in action.

Assume that we have a new HTML page as follows:

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>attr Method Demo</title>
7     <script src="https://code.jquery.com/jquery-2.2.4.min.js"
8           integrity="sha256-BbhdlvQf/xTY9gja0Dq3HiwQF8LaCRTXxZKRute1T44="
9           crossorigin="anonymous"></script>
10    <script src="assets/javascripts/attr.js"></script>
11  </head>
12  <body>
13    <form>
14      <input type="text" placeholder="Give your name" value="" id="name"/><br/>
15      <button id="set-value">Set Value</button><br/>
```

```
16      <button id="clear-value">Clear Value</button>
17  </form>
18 </body>
19 </html>
```

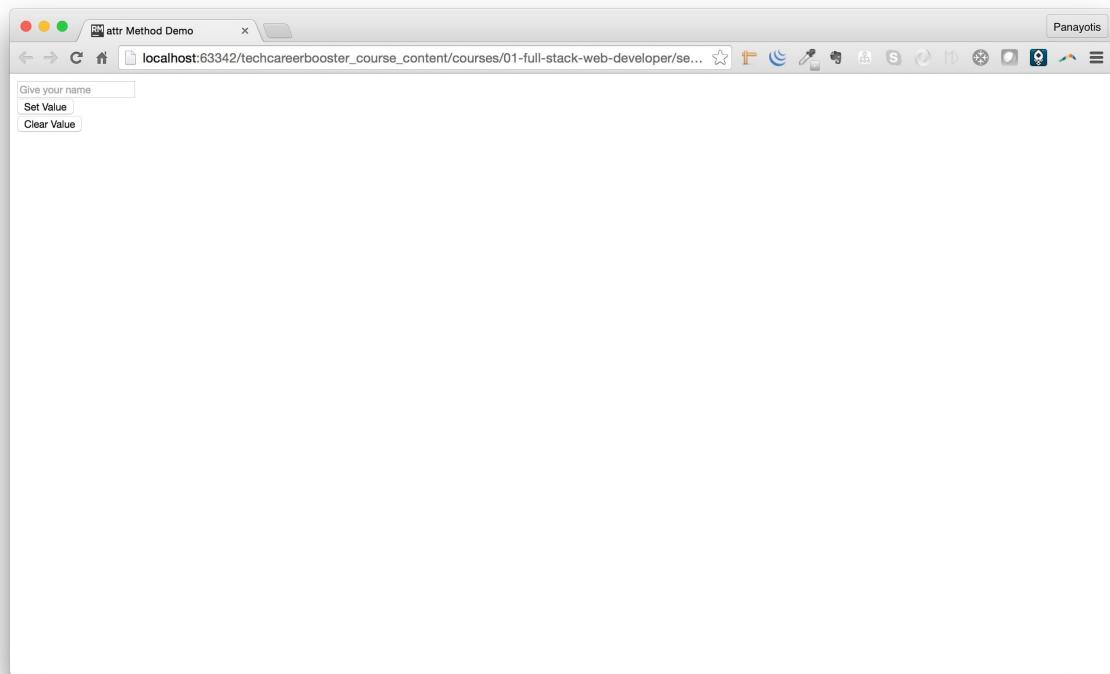
(the above code snippet online)

with the following JavaScript content inside assets/javascripts/attr.js file:

```
1 $(document).ready(function() {
2     $('#set-value').on('click', function() {
3         var $inputBox = $('#name');
4         $inputBox.attr('value', 'George');
5         return false;
6     });
7     $('#clear-value').on('click', function() {
8         var $inputBox = $('#name');
9         $inputBox.attr('value', '');
10        return false;
11    });
12});
```

(the above code snippet online)

If you save the above and load the page on your browser, you will see this page:



### Setting Attribute Values

This page works as follows. If one clicks on button `Set Value` it will display the name George in the input box. If one clicks on button `Clear Value` it will set the value of input box to the blank string. Either is achieved by setting the HTML attribute `value` of the input control.

The `$inputBox.attr('value', 'George');` calls the method `.attr()` on the jQuery referencing the input box and sets the value George to the attribute with name `value`.

The `$inputBox.attr('value', ''');` calls the same method, but sets the value '' to the attribute with name `value`.

*Note:* Again, the `$inputBox` is a jQuery collection of matching elements. What do they match to? They match to the selector that we have used here: `$( '#name' )`. Hence, they are all the elements that have id `name`. Usually, an HTML page has a single element matching a particular id. In other words, the id is (should be) unique. This means that the collection `$inputBox` contains only one element, a.k.a its `length` property should return 0.

The `.attr()` method can also be used to read the value of an HTML attribute. You only have to call that with a single argument, the name of the attribute the value of which you want to read from.

Let's do another example:

```

1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="utf-8">
5          <meta name="viewport" content="width=device-width, initial-scale=1.0">
6          <title>attr Method Demo</title>
7          <script src="https://code.jquery.com/jquery-2.2.4.min.js"
8                  integrity="sha256-BbhdlvQf/xTY9gja0Dq3HiwQF8LaCRTXxZKRutelT44="
9                  crossorigin="anonymous"></script>
10         <script src="assets/javascripts/attr.js"></script>
11     </head>
12     <body>
13         <form>
14             <input type="text" placeholder="Give your name" value="" id="name"/><br/>
15             <div id="value-container"></div>
16             <button id="set-value">Set Value</button><br/>
17             <button id="clear-value">Clear Value</button><br/>
18             <button id="read-value">Read Value</button>
19         </form>
20     </body>
21 </html>
```

(the above code snippet online)

This page is almost the same as before. We have added a `div` element with empty HTML content and id `value-container`. We have also added a new button with id `read-value`. We are going to

attach a handler on that last button, that would copy the value of the attribute `value` from the input box and would set it as an HTML content inside the `div` element. This is the new version of our JavaScript code:

```

1 $(document).ready(function() {
2     $('#set-value').on('click', function() {
3         var $inputBox = $('#name');
4         $inputBox.attr('value', 'George');
5         return false;
6     });
7     $('#clear-value').on('click', function() {
8         var $inputBox = $('#name');
9         $inputBox.attr('value', '');
10        var $valueContainer = $('#value-container');
11        $valueContainer.html('');
12        return false;
13    });
14    $('#read-value').on('click', function() {
15        var $inputBox = $('#name');
16        var value = $inputBox.attr('value');
17        var $valueContainer = $('#value-container');
18        $valueContainer.html(value);
19        return false;
20    })
21 });

```

(the above code snippet online)

We have updated the JavaScript handlers:

1. The `#clear-value` handler to clear the content of the `div #value-container`.
2. The `#read-value` handler to take the value of the attribute with name `value` and set it as HTML content of the `#value-container`.

Note that the above JavaScript code has a lot of repetition. We can polish that and keep it a little bit DRY (Don't Repeat Yourself).

```

1 $(document).ready(function() {
2     var $inputBox = $('#name');
3     var $valueContainer = $('#value-container');
4
5     $('#set-value').on('click', function() {
6         $inputBox.attr('value', 'George');
7         return false;
8     });
9     $('#clear-value').on('click', function() {

```

```

10     $inputBox.attr('value', '');
11     $valueContainer.html('');
12     return false;
13 });
14 $('#read-value').on('click', function() {
15     var value = $inputBox.attr('value');
16     $valueContainer.html(value);
17     return false;
18 })
19 );

```

(the above code snippet online)

As you can see above, at the start of our handler function for document ready, we save two references. One for the input box and one for the value container. Then we can reuse those inside the handlers for click events on the buttons.

## Removing And Adding Content

**Important:** Although the above removes some repetition, it does have a caveat. When the function handlers for the click events on the buttons are invoked, they assume that the \$inputBox and \$valueContainer they still point to the input box and div container. But, this may not be true, if, before the handler code invocation, the input box or the value container is removed from the page.

We will further expand on the previous important note above, by giving an example. We will add one more button on our page that, on click, will remove the input box. We will then click on that button, and we will see the input box removed. This will make the other buttons (Set Value, Clear Value, Read Value) not work, obviously, in the absence of the input box. But, even if we add the input box back (we do that with the introduction of one more button Add Input Box), these three buttons (Set Value, Clear Value, Read Value) will still not be working. Because, their functionality was bound to the collection of elements saved in \$inputBox, collection built on previous instance of the button.

This is easier explained with actual code and run, rather than words. Let's see that.

The new HTML page content, with the new buttons has as follows:

```

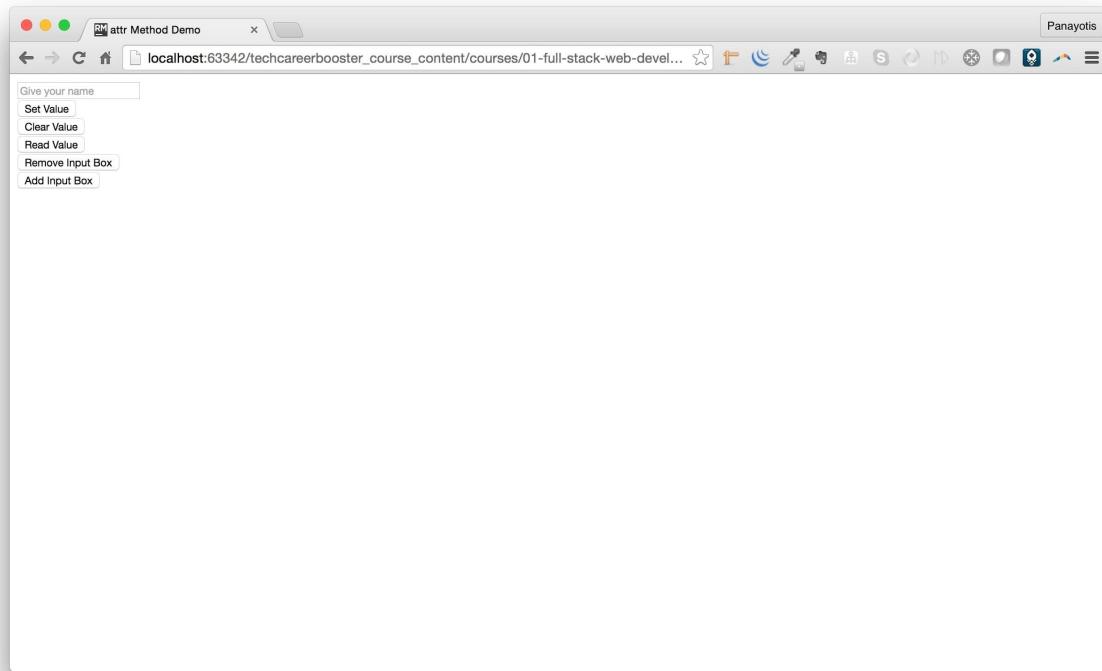
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>attr Method Demo</title>
7     <script src="https://code.jquery.com/jquery-2.2.4.min.js"
8           integrity="sha256-BbhdlvQf/xTY9gja0Dq3HiwQF8LaCRTXxZKRute1T44="
9           crossorigin="anonymous"></script>

```

```
10    <script src="assets/javascripts/attr.js"></script>
11  </head>
12  <body>
13    <form>
14      <input type="text" placeholder="Give your name" value="" id="name"/><br/>
15      <div id="value-container"></div>
16      <button id="set-value">Set Value</button><br/>
17      <button id="clear-value">Clear Value</button><br/>
18      <button id="read-value">Read Value</button><br/>
19      <button id="remove-input-box-button">Remove Input Box</button><br/>
20      <button id="add-input-box-button">Add Input Box</button>
21    </form>
22  </body>
23 </html>
```

(the above code snippet online)

If you load the page on your browser, you will see this:



#### Two Buttons Added To Remove And Add Button Back

And we amend the assets/javascripts/attr.js file as follows:

```

1 $(document).ready(function() {
2     var $inputBox = $('#name');
3     var $valueContainer = $('#value-container');
4
5     $('#set-value').on('click', function() {
6         $inputBox.attr('value', 'George');
7         return false;
8     });
9     $('#clear-value').on('click', function() {
10        $inputBox.attr('value', '');
11        $valueContainer.html('');
12        return false;
13    });
14    $('#read-value').on('click', function() {
15        var value = $inputBox.attr('value');
16        $valueContainer.html(value);
17        return false;
18    });
19    $('#remove-input-box-button').on('click', function() {
20        $inputBox.remove();
21        return false;
22    });
23    $('#add-input-box-button').on('click', function(){
24        var $form = $('form');
25        $form.prepend('<input type="text" id="name" placeholder="Give your name" \
26 value="" />');
27        return false;
28    });
29 });

```

(the above code snippet online)

Basically, we have added the following two blocks of code:

```

1 $('#remove-input-box-button').on('click', function() {
2     $inputBox.remove();
3     return false;
4 });
5 $('#add-input-box-button').on('click', function(){
6     var $form = $('form');
7     $form.prepend('<input type="text" id="name" placeholder="Give your name" valu\
8 e="" />');
9     return false;
10 });

```

(the above code snippet online)

The first block of code uses the `.remove()` jQuery method. This removes the selected elements from the HTML document. On our particular case, it will remove the input box, since it is called on `$inputBox`.

The second block of code uses the `.prepend()` jQuery method. This method adds, dynamically, HTML content to the document, to your page. The HTML content is given as first argument and it is added as first child to the selected elements. Since we call `prepend` as `$form.prepend( . . . )`, the content given as first argument will be added as first child of the `form` element. In other words, this second block of code adds the input button back. The important detail here is that it adds it with the same `id`, `name`.

We will save all the above content and reload the page on our browser.

1. We will use the first three buttons (`Set Value`, `Clear Value`, `Read Value`) that work with the input box and the div container.
2. Then we will remove the button by clicking on the `Remove Input Box` button.
3. Then we will use the first three buttons again and we will see that they will not be doing anything.
4. Then we will add the input box back again, by clicking the `Add Input Box` button.
5. Finally, we will use the first three buttons again, and we will still see that these buttons don't do anything. Again, that being because their code refers to the old instance of the button, as it was captured in the variable `$inputBox`, and not to the new button added dynamically by the `Add Input Box` handler.

Watch the video to see this page in action:

### Removing And Adding Content Dynamically

As you can see, the newly added content (the input box in our case) is not referred by jQuery collections that have been constructed before their introduction. How can we fix that?

Obviously, the solution is to remove the collection references from the top of the document ready handler and put them back, again, inside the button handlers:

```
1 $(document).ready(function() {  
2     $('#set-value').on('click', function() {  
3         var $inputBox = $('#name');  
4         $inputBox.attr('value', 'George');  
5         return false;  
6     });  
7     $('#clear-value').on('click', function() {  
8         var $inputBox = $('#name');  
9         $inputBox.attr('value', '');  
10        var $valueContainer = $('#value-container');  
11        $valueContainer.html('');  
12        return false;  
13    });  
14    $('#read-value').on('click', function() {  
15        var $inputBox = $('#name');
```

```

16     var value = $inputBox.attr('value');
17     var $valueContainer = $('#value-container');
18     $valueContainer.html(value);
19     return false;
20 });
21 $('#remove-input-box-button').on('click', function() {
22     var $inputBox = $('#name');
23     $inputBox.remove();
24     return false;
25 });
26 $('#add-input-box-button').on('click', function(){
27     var $form = $('form');
28     $form.prepend('<input type="text" id="name" placeholder="Give your name" \
29 value="" />');
30     return false;
31 });
32 });

```

(the above code snippet online)

If you save the above and try to work the page buttons, remove and add dynamically the input box, you will see that the three buttons (Set Value, Clear Value, Read Value) work as expected, even after the input box is added back dynamically.

## Getting and Setting CSS Attributes

Earlier on, we have learnt about the `.attr()` method that can be used to get or set the value of an HTML element attribute. There is an equivalent method, `.css()` which can be used to get and set CSS properties for selected elements.

Let's do an example. Here is the HTML page:

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Using css() method to set background color</title>
7     <!-- Bootstrap CSS files -->
8     <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/\ 
9      css/bootstrap.min.css"
10        integrity="sha384-1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1\ \
11 fgjWPGmkzs7" crossorigin="anonymous">
12     <!-- Custom CSS -->
13     <link rel="stylesheet" href="assets/stylesheets/main.css" type="text/css">
14
15     <!-- jQuery -->

```

```

16   <script src="https://code.jquery.com/jquery-2.2.4.min.js"
17     integrity="sha256-Bbhd1vQf/xTY9gja0Dq3HiwQF8LaCRTXxZKRute1T44="
18     crossorigin="anonymous"></script>
19
20   <!-- Custom JavaScript -->
21   <script src="assets/javascripts/bg-color.js"></script>
22 </head>
23 <body>
24   <div id="buttons-container">
25     <button class="btn btn-lg bg-yellow" data-class="change-bg-color-button">Ye\
26 llow</button>
27     <button class="btn btn-lg bg-blue color-white" data-class="change-bg-color-\
28 button">Blue</button>
29     <button class="btn btn-lg bg-green color-white" data-class="change-bg-color\
30 -button">Green</button>
31   </div>
32
33   <div id="color-container" class="page-header">
34   </div>
35 </body>
36 </html>

```

(the above code snippet online)

It references, besides Bootstrap and jQuery, a local CSS and a local JavaScript file. The local CSS file is:

```

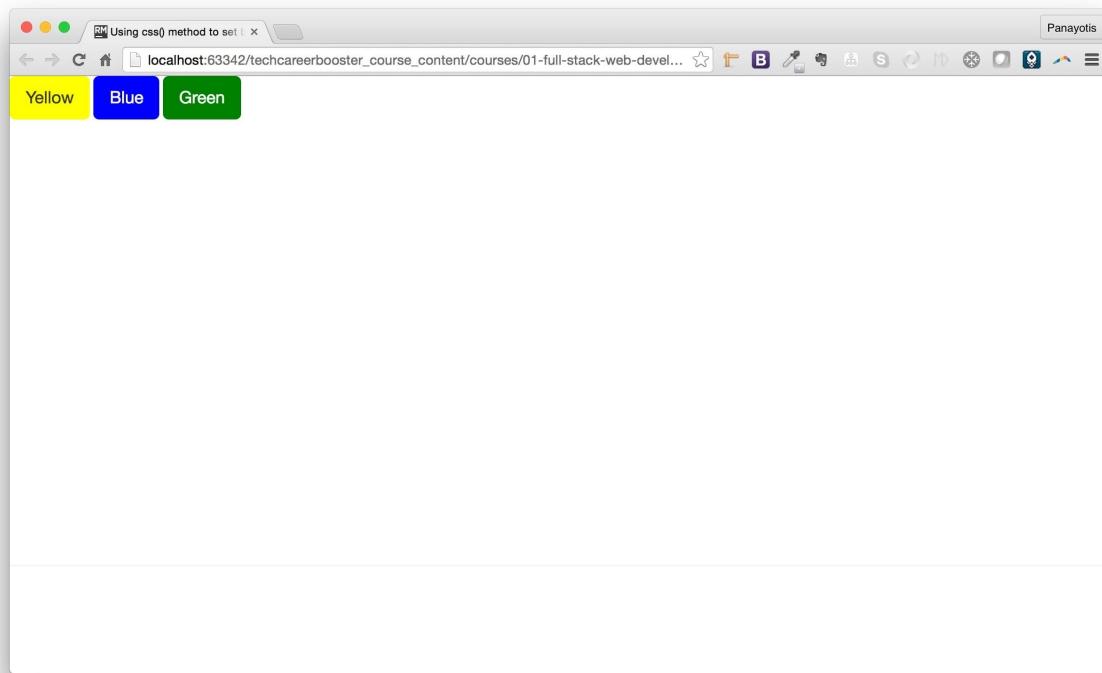
1  * {
2    box-sizing: border-box;
3  }
4
5  body, html {
6    height: 100%;
7  }
8
9  #buttons-container {
10   min-height: 30px;
11 }
12
13 #color-container {
14   margin-top: 30px;
15   margin-bottom: 30px;
16   height: 70%;
17 }
18
19 .bg-yellow {
20   background-color: Yellow;

```

```
21 }
22
23 .bg-blue {
24     background-color: Blue;
25 }
26
27 .bg-green {
28     background-color: Green;
29 }
30
31 .color-white {
32     color: White;
33 }
```

(the above code snippet online)

Assume, initially, that the local JavaScript file (`assets/javascripts/bg-color.js`) is empty. If you save the files and load the page on your browser you will see this:



#### HTML Page with 3 Action Buttons

What we want the page to do is the following. We want to click on any of the buttons and have the `div` with id `color-container` to be filled in with the corresponding background color. Like that:

#### App To Change Background Color

How would we implement such an application?

Reading the HTML code above, pay attention to the following:

1. We reference Bootstrap CSS. This will allow us to use the classes `btn` and `btn-lg`, as well as the class `page-header`.
2. We reference jQuery. This will allow us to build the dynamic behaviour of our page.
3. We have three buttons that they do not have their own id. But they have the same `data-class`. HTML element attributes that start with `data-` are usually used to declare custom HTML attributes (vs standard HTML attributes). We will use the value of `data-class` to select all the buttons on this page in order to attach to all of them the same functionality: When a button is clicked, we will be setting the background color of the `color-container` div. The bg color will be set to the value that is inside the HTML content of the corresponding button.

Let's implement this using JavaScript. Put the following JavaScript content inside the `assets/javascripts/bg-color.js` file:

```
1 $(document).ready(function() {  
2     $('[data-class=change-bg-color-button]').on('click', function() {  
3         var color = $(this).html();  
4         var $colorContainer = $('#color-container');  
5         $colorContainer.css('background-color', color);  
6         return false;  
7     });  
8});
```

(the above code snippet online)

If you save the above and you reload the page on your browser, then you are ready to enjoy the dynamic behaviour of your page. Every time you click on a button, the corresponding color is set as background color of the div below.

If you read the JavaScript code above carefully, you will see that it is all very easy and simple.

1. We select all the buttons: `$("[data-class=change-bg-color-button]").` We could have equally done `($('button')`, but for educational reasons, we decided to use the technique to select based on the value of an attribute.
2. On the matched button elements, we attach an event handler for the `click` event.
3. Inside the event handler, the `this` keyword points to the target of the event. The target of the event is one of the buttons.
4. We wrap the target of the event, the button, into a jQuery object (`$(this)`) and then we call the method `.html()` on that in order to get the value of its HTML content. We save that into a variable `color`.
5. Then, we locate the `color-container` div(`var $colorContainer = $('#color-container');`) and
6. We set its `background-color` to have the value stored inside `color` variable. The setting is done with the help of the method `.css()`.

7. Finally, we return `false` to make sure that the default handler for the button is not triggered.

Easy, isn't it?

With `.css()` method you can set any CSS attribute that you can usually set via CSS properties.

## Getting and Setting CSS Classes

We can definitely use the `.attr()` method to set a value for the `class` attribute of an HTML element. However, jQuery provides a series of convenience method that deal with this very popular attribute:

1. `.addClass()`. It is used to add a class to an HTML element.
2. `.removeClass()`. It is used to remove a class from an HTML element.
3. `.toggleClass()`. It is used to add a class that does not exist and remove a class that exists.

These methods are very useful. Adding and removing classes dynamically can change many aspects of the appearance of your HTML page at once.

Let's try an example. Here is the HTML page:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Using css() method to set background color</title>
7     <!-- Bootstrap CSS files -->
8     <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/\css/bootstrap.min.css"
9          integrity="sha384-1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1\fgjWPGmkzs7" crossorigin="anonymous">
10    <!-- Custom CSS -->
11    <link rel="stylesheet" href="assets/stylesheets/setting-class.css" type="text\css">
12
13
14
15
16    <!-- jQuery -->
17    <script src="https://code.jquery.com/jquery-2.2.4.min.js"
18        integrity="sha256-BbhdlvQf/xTY9gja0Dq3HiwQF8LaCRTXxZKRutelT44="
19        crossorigin="anonymous"></script>
20
21    <!-- Custom JavaScript -->
22    <script src="assets/javascripts/setting-class.js"></script>
23
24
25   <div id="buttons-container">
```

```

26      <button class="btn btn-lg">monospace</button>
27      <button class="btn btn-lg">serif</button>
28      <button class="btn btn-lg">sans-serif</button>
29  </div>
30
31  <div id="text-container" class="page-header text-center">
32      I love JavaScript!
33  </div>
34  </body>
35 </html>

```

(the above code snippet online)

This HTML page is very simple. It has three buttons and a div with id text-container below them. We will make the three buttons apply the class with the name equal to their HTML content, to the div with id text-container.

The assets/javascripts/setting-class.js file has the following, very simple, content:

```

1 $(document).ready(function() {
2     $('button').on('click', function() {
3         var $textContainer = $('#text-container');
4         var classToApply = $(this).html();
5         var classToRemove = $textContainer.attr('data-class-to-remove');
6         if ( classToRemove !== '' && classToRemove !== undefined) {
7             $textContainer.removeClass(classToRemove);
8         }
9         $textContainer.addClass(classToApply);
10        $textContainer.attr('data-class-to-remove', classToApply);
11        return false;
12    });
13 });

```

(the above code snippet online)

On all the buttons (`($('button')`) it attaches the same click event handler. The event handler takes the HTML content of the clicked button and saves it to the variable `classToApply`. Then, with the `$textContainer.addClass(classToApply);` will apply the class. Hence, the buttons potentially will apply the classes monospace, serif and sans-serif. These classes are defined inside the assets/stylesheets/setting-class.css file as follows:

```
1  * {
2      box-sizing: border-box;
3  }
4
5  body, html {
6      height: 100%;
7  }
8
9  #buttons-container {
10     min-height: 30px;
11 }
12
13 #text-container {
14     margin: 30px;
15     height: 70%;
16     padding: 30px;
17 }
18
19 .monospace {
20     font-family: monospace;
21     font-size: 32px;
22     color: Blue;
23     background-color: Gray;
24 }
25
26 .serif {
27     font-family: "Times New Roman", serif;
28     font-size: 38px;
29     color: Green;
30     background-color: Maroon;
31 }
32
33 .sans-serif {
34     font-family: Verdana, Arial, sans-serif;
35     font-size: 48px;
36     color: White;
37     background-color: DarkBlue;
38 }
```

(the above code snippet online)

There is an issue that we need to take care of. Before applying the class that corresponds to the button clicked, we need to remove any previously applied class. In order to remove the previously applied class, whenever we apply a new class we attach its name to the `data-class-to-remove` attribute. Hence, before applying the new class, we know which class we have to remove. The removal of the class is done with `$textContainer.removeClass(classToRemove);`.

Let's see this page in action:

## Adding and Removing Classes

Applying and removing a class dynamically using jQuery is a very powerful tool that will definitely prove useful on the field.

## Getting and Setting HTML Form Values

Whenever you deal with form values, the `.val()` method is used to read the values of the input controls of the form.

Let's create an HTML page that would ask the user to fill in their profile details. On submission of the form, we would check that all the values have been provided. If mandatory values are missing, we will highlight the fields on error.

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Form Validation</title>
7
8     <!-- Bootstrap CSS files -->
9     <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/\css/bootstrap.min.css"
10    integrity="sha384-1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1\fgjWPGmkzs7" crossorigin="anonymous">
11
12     <!-- Custom CSS -->
13     <link rel="stylesheet" href="assets/stylesheets/profile.css" type="text/css">
14
15     <!-- jQuery -->
16     <script src="https://code.jquery.com/jquery-2.2.4.min.js"
17            integrity="sha256-BbhdlvQf/xTY9gja0Dq3HiwQF8LaCRTXxZKRutelT44="
18            crossorigin="anonymous"></script>
19
20
21     <!-- Custom JavaScript -->
22     <script src="assets/javascripts/profile.js"></script>
23   </head>
24
25   <body>
26
27     <div class="container">
28       <h1 class="page-header">Profile Details</h1>
29
30       <form id="profile-details-form">
31         <div class="form-group">
32           <label for="first-name">First Name *:</label>
33           <input type="text" class="form-control mandatory" id="first-name" place\
```

```
34 holder="First Name" autofocus="autofocus"/>
35     </div>
36
37     <div class="form-group">
38         <label for="last-name">Last Name *:</label>
39         <input type="text" class="form-control mandatory" id="last-name" placeh\
40 older="Last Name"/>
41     </div>
42
43     <div class="form-group">
44         <label for="contact-phone">Contact Phone *:</label>
45         <input type="text" class="form-control mandatory" id="contact-phone" pl\
46 aceholder="Contact Phone"/>
47     </div>
48
49     <div class="form-group">
50         <label for="gender">Gender *:</label>
51         <select id="gender" class="form-control mandatory">
52             <option></option>
53             <option value="male">Male</option>
54             <option value="female">Female</option>
55         </select>
56     </div>
57
58     <div class="text-right">
59         <button type="submit" class="btn btn-primary btn-lg">Save</button>
60     </div>
61     </form>
62 </div>
63 </body>
64 </html>
```

(the above code snippet online)

This is an HTML page with a profile details form. It references Bootstrap and jQuery. Also it references two local files, the CSS file assets/stylesheets/profile.css and the JavaScript file assets/javascripts/profile.js.

The CSS file is very simple:

```
1 * {
2     box-sizing: border-box;
3 }
4
5 .has-error {
6     border: 1px solid red;
7 }
```

(the above code snippet online)

It basically defines the class `has-error`. This class will dynamically be added to elements that are mandatory but the user has not provided any info.

If you save the above and load the page on your browser, you will see a form like this:

The screenshot shows a web browser window titled "Form Validation". The URL in the address bar is "localhost:63342/techcareerbooster\_course\_content/courses/01-full-stack-web-developer/sections/07...". The main content is a form titled "Profile Details" with the following fields:

- First Name \*: An input field containing "First Name".
- Last Name \*: An input field containing "Last Name".
- Contact Phone \*: An input field containing "Contact Phone".
- Gender \*: A dropdown menu currently empty.

A blue "Save" button is located at the bottom right of the form area.

#### An HTML Form - Profile Details

Let's see how we are going to implement the validation on the mandatory fields. As you can see from the HTML code above, we have added the class `mandatory` to the fields in order to indicate that the user needs to key in some info before submitting the form. The form could equally had non mandatory fields, case in which those fields wouldn't had that class added.

The JavaScript code that does the whole validation is the following.

```

1 $(function(){
2     $('#profile-details-form button[type=submit]').on('click', function() {
3         var invalidForm = false;
4         var $firstInvalidField = null;
5         $('.mandatory').each(function() {
6             var $input = $(this);
7             var value = $input.val();
8             if (value.trim() === '') {
9                 if (!$input.hasClass('has-error')) {
10                     $input.addClass('has-error');
11                 }
12 
13             // We save the first invalid field, so that we can move the focus\
14             on it.
15             if ($firstInvalidField === null) {
16                 $firstInvalidField = $input;
17             }
18 
19             // We flag the form as invalid, so that we do not submit it.
20             invalidForm = true;
21         }
22         else {
23             $input.removeClass('has-error');
24         }
25     });
26 
27     if (invalidForm) {
28         alert('Please, fill in the mandatory form fields');
29 
30         // give the focus on the first invalid field
31         if ($firstInvalidField !== null) {
32             $firstInvalidField.focus();
33         }
34         return false; // do not submit the form, since we prevent default han\
35 dler from running
36     }
37 });
38 });

```

(the above code snippet online)

If you save all the files and load the page on your browser, you will see an HTML form page behaving like this:

### HTML Page With Form Validations

Let's give some more explanation on the JavaScript code above:

(1).We install an event handler for the event click on the button of the form:

```
1  $('#profile-details-form button[type=submit]').on('click', function() { ... });
```

(the above code snippet online)

(2).We use a variable named `invalidForm` to flag whether the form has validation errors or not. If it has, we will display an `alert()` and we will return `false`. Returning `false` will prevent the button default handler from firing, hence, the form will not be submitted.

(3).We also use another variable, named `$firstInvalidField`, that will be holding the first field that is not valid. Hence, we will know where to put the focus on after displaying the alert.

```
1  if ($firstInvalidField !== null) {
2    $firstInvalidField.focus();
3 }
```

(the above code snippet online)

(4).We use the `.each()` jQuery method that allows us to execute/call a function on each one of the matching elements.

```
1  $('.mandatory').each(function() { ... });
```

(the above code snippet online)

As you can see above, we select all the elements that have the class `mandatory` and we execute a function on each one of these elements.

(5).The function that is executed for each one of the mandatory input fields

(5.1).saves a jQuery reference to the input control itself: `var $input = $(this);`.

(5.2).saves the value of the input control in the `value` variable: `var value = $input.val();`.

(5.3).if the value is blank, we basically flag the form as invalid (`invalidForm = true;`) and we add the class `has-error`. Note that the class is added only if it is not present. This check is done with the method `.hasClass()`.

## Inserting and Replacing Elements

jQuery offers some methods that will allow you to add HTML content / elements dynamically. These are the ones most frequently used:

- **`append()`**: Insert content, specified by the parameter, as the last child of each element in the set of matched elements.
- **`prepend()`**: Insert content, specified by the parameter, as the first child of each element in the set of matched elements.
- **`after()`**: Insert content, specified by the parameter, after each element in the set of matched elements.
- **`before()`**: Insert content, specified by the parameter, before each element in the set of matched elements.

- **replaceWith()**: Replace each element in the set of matched elements with the provided new content and return the set of elements that was removed.

We will use an example to demonstrate the use of these functions.

Let's start with the HTML content:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Form Validation</title>
7
8     <!-- Bootstrap CSS files -->
9     <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/\css/bootstrap.min.css"
10    integrity="sha384-1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1\fgjWPgmks7" crossorigin="anonymous">
11
12    <!-- Custom CSS -->
13    <link rel="stylesheet" href="assets/stylesheets/inserting-and-replacing.css" \type="text/css">
14
15
16    <!-- jQuery -->
17    <script src="https://code.jquery.com/jquery-2.2.4.min.js"
18      integrity="sha256-BbhdlvQf/xTY9gja0Dq3HiwQF8LaCRTXxZKRutelT44="
19      crossorigin="anonymous"></script>
20
21
22    <!-- Custom JavaScript -->
23    <script src="assets/javascripts/inserting-and-replacing.js"></script>
24 </head>
25
26 <body>
27   <div class="container">
28     <div class="row">
29
30       <div class="col-xs-2 image-container">
31         
33
34       <div class="add-right-link-container">
35         <a href="#">
36           <span class="glyphicon glyphicon-plus"></span>
37         </a>
38       </div>
39
40       <div class="remove-left-link-container">
```

```
41         <a href="#">
42             <span class="glyphicon glyphicon-minus"></span>
43         </a>
44     </div>
45 </div>
46
47     </div>
48 </div>
49 </body>
50 </html>
```

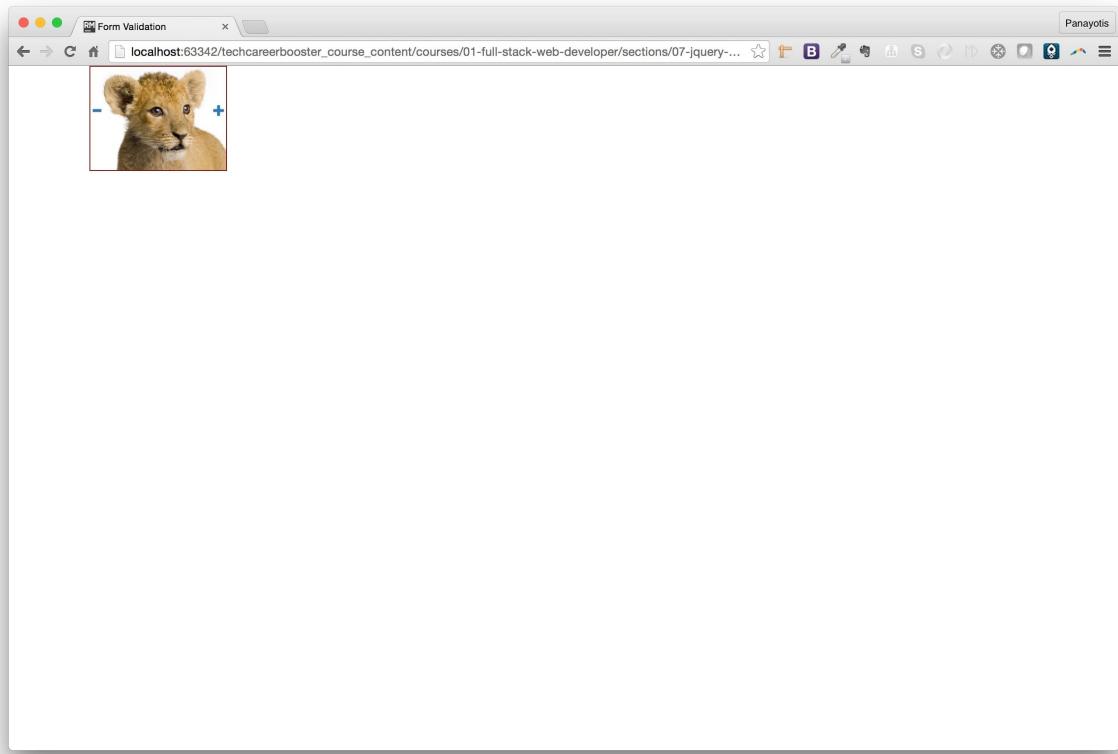
(the above code snippet online)

The above page references Twitter Bootstrap, jQuery and some custom CSS and JavaScript files. The CSS has as follows:

```
1  * {
2      box-sizing: border-box;
3  }
4
5  img {
6      border: 1px solid maroon;
7  }
8
9 .image-container {
10    position: relative;
11 }
12
13 .add-right-link-container {
14    position: absolute;
15    top: 35%;
16    right: 17px;
17 }
18
19 .remove-left-link-container {
20    position: absolute;
21    top: 35%;
22    left: 17px;
23 }
```

(the above code snippet online)

Before looking into the actual JavaScript code, let's save the HTML and CSS and load the page on your browser. You will see this:



Page with Small Lion Image

It is very easy to build this page, as long as you have studied the HTML/CSS material on previous chapters.

What we want to do now is the following:

1. Whenever the user clicks on the plus link, on the right hand side of the lion image, a new lion should be added after the lion clicked.
2. Whenever the user clicks on the minus link, on the left hand side of the lion image, the lion should be removed.
3. The lion image that is added it is being faded in.
4. The lion image that is removed it is being faded out.

The above behaviour is depicted on the following video:

#### How Final Page Will Behaving

How will we achieve this functionality? With the help of JavaScript of course. Let's start little by little.

```
1 $(document).ready(function(){
2     $('.add-right-link-container a').on('click', function() {
3         var $image_container = $(this).parents('.image-container');
4
5         var image_container_content = $image_container[0].outerHTML;
6
7         $image_container.after(image_container_content);
8     });
9 });
```

(the above code snippet online)

Let's save the above content inside the assets/javascripts/inserting-and-replacing.js file. What does the above JavaScript do?

1. It attaches a click handler on all anchor a elements that are inside a .add-right-link-container. In other words on all the plus icons on the right hand side of all the small lion images.
2. The click handler will use the `.parents()` method to locate the parent of the link that matches the CSS selector `.image-container`. We are doing that because we want to get all the HTML fragment that corresponds to the link clicked. We will take the HTML content and we will replicate it as a new sibling next to the original one.
3. Having the image container div that contains the link clicked, we take its outer HTML using `...[0].outerHTML` property on the HTML element itself. Note that `$image_container[0]` returns the DOM element representation of the first matching element, hence, the DOM element object that corresponds to the image container div of the link clicked.
4. We then use the `.after()` method to append the content that constitutes the image container next to the original image container (`$image_container.after(image_container_content);`).

If you save the above JavaScript content and you reload the page on your browser, you will see the page behaving like this: You will be able to add only one image next to the existing one. Why is that?

This is because the click handler has been added to the existing `.add-right-link-container a` elements when the page was loaded. When we click and add a new lion image, this new lion image was not there when the page was loaded for first time, and hence, no click handler was attached to it.

How do we attach the click handler dynamically, to both existing elements and new ones? In order to achieve that, instead of attaching the handler while on `$(document).ready(function() { ... })`, we attach the handler on `$(document)` like this:

```

1 $(document).on('click', '.add-right-link-container a', function() {
2     var $image_container = $(this).parents('.image-container');
3
4     var image_container_content = $image_container[0].outerHTML;
5
6     $image_container.after(image_container_content);
7 });

```

(the above code snippet online)

Replace the content of the assets/javascripts/inserting-and-replacing.js file with the one above. This tells to jQuery to attach the event handler for click event on all .add-right-link-container a elements, even if they are created dynamically after the page has been loaded.

Having the new JavaScript content above, load the page again and try to add new lions. Does it work?

It does! As you can see, the small lion images are added after you click on any of the existing small lion images, both on the existing small lion image and on the new ones added.

Now, we need to make the UI respond to the clicks on the minus icon. This will have to remove the corresponding image container. Let's amend the JavaScript file assets/javascripts/inserting-and-replacing.js as follows:

```

1 $(document).on('click', '.add-right-link-container a', function() {
2     var $image_container = $(this).parents('.image-container');
3
4     var image_container_content = $image_container[0].outerHTML;
5
6     $image_container.after(image_container_content);
7 }).on('click', '.remove-left-link-container a', function() {
8     var $image_container = $(this).parents('.image-container');
9     $image_container.remove();
10 });

```

(the above code snippet online)

It's very simple. We have added a click handler for the .remove-left-link-container a anchors. Also, the handler is attached so that it is triggered for all the elements, either the ones that exist on page load, or the ones that are created dynamically after the page has been loaded.

What does it do? it just removes the container.

Save the above JavaScript file and reload the page on your browser. Then try to click on pluses and minuses. It should be adding and removing the small lion images.

### Clicking on Pluses and Minuses

What's left to be done? It is the fade in and fade out effects. Let's start with the fade out effect for the small lion images that we remove.

There is a method called `.fadeOut()` that can be used here. What this method basically does, is to set the `style` attribute to the value `display: none;` and, effectively, make the element invisible. Also, it does not do it immediately, when we call it. It does it with a smoothing delay of 400 milliseconds. Let's change the click handler for the `.remove-left-link-container a` to actually fade out the image container, instead of removing it.

Make sure that the file `assets/javascripts/inserting-and-replacing.js` has the following content:

```
1 $(document).on('click', '.add-right-link-container a', function() {  
2     var $image_container = $(this).parents('.image-container');  
3  
4     var image_container_content = $image_container[0].outerHTML;  
5  
6     $image_container.after(image_container_content);  
7 }).on('click', '.remove-left-link-container a', function() {  
8     var $image_container = $(this).parents('.image-container');  
9     $image_container.fadeOut();  
10});
```

(the above code snippet online)

If you save the above content and reload the page on your browser, you will see that the click on a minus will fade out the lion image. The problem, however, is that the actual image container is not removed any more. It is only faded out. See how we remove the lion images, but the corresponding divs still remain as part of the HTML elements tree.

### Fading Out Does Not Remove The Element

How can we both fade out and remove the element? Somebody might say, let's write something like this:

```
1 $image_container.fadeOut().remove();
```

(the above code snippet online)

Although syntactically correct, it removes the element before letting the browser fade that out.

The solution is to call the `.remove()` method at the end of the fading out process. This can be achieved if we pass as argument to the `fadeOut()` method a function that will do the removal.

Make sure that the JavaScript file `assets/javascripts/inserting-and-replacing.js` has the following content:

```

1 $(document).on('click', '.add-right-link-container a', function() {
2     var $image_container = $(this).parents('.image-container');
3
4     var image_container_content = $image_container[0].outerHTML;
5
6     $image_container.after(image_container_content);
7 }).on('click', '.remove-left-link-container a', function() {
8     var $image_container = $(this).parents('.image-container');
9     $image_container.fadeOut(function() {
10        $(this).remove();
11    });
12 });

```

(the above code snippet online)

As you can see, when we call the `fadeOut()` method we also pass a function that will call `$(this).remove()` and will remove the image container.

Save the above and load the page on your browser again. Try adding and removing images. Have your developer tools open and see how the elements are now been removed, at the end of the fading out process.

Another thing that is left to be done, is the fading in of the images that are being added. We need to add some piece of code inside the handler that handles the additions. When a new image is added we need to set `display` property equal to `none`, making it invisible, and then we need to call the `.fadeIn()` method. This method will remove the `display:none;` property, slowly, finally making it visible.

Change the content of the `assets/javascripts/inserting-and-replacing.js` file as follows:

```

1 $(document).on('click', '.add-right-link-container a', function() {
2     var $image_container = $(this).parents('.image-container');
3
4     var image_container_content = $image_container[0].outerHTML;
5
6     $image_container.after(image_container_content);
7
8     var $last_image_container_added = $image_container.next('.image-container');
9
10    $last_image_container_added.css('display', 'none').fadeIn();
11 }.on('click', '.remove-left-link-container a', function() {
12     var $image_container = $(this).parents('.image-container');
13     $image_container.fadeOut(function() {
14        $(this).remove();
15    });
16 });

```

(the above code snippet online)

We have added the two lines:

```
1 var $last_image_container_added = $image_container.next(".image-container");  
2  
3 $last_image_container_added.css('display', 'none').fadeIn();
```

(the above code snippet online)

On first line, having the `$image_container` that we clicked on, and having already added another container next to it (on the previous lines), we now use the `.next()` method to get its next, newly added, div element.

On second line, we set the `display` property to `none` and then we call the `.fadeIn()` method in order to remove it gracefully and smoothly.

Save the above content and reload the page on your browser. You will see that the page now behaves as required.

## strict mode

Before we close the jQuery chapter, we will now introduce you to the *strict* mode. Strict mode is a mode that you can switch your JavaScript interpreter to and make it be more rigorous and *strict* with your code. It has been introduced in order to protect you from mistakes that could go unnoticed and make your life hard.

## Global variables

Let's see the following script:

```
1 'use strict';  
2  
3 message = 'Hello World';  
4  
5 console.log(message);
```

(the above code snippet online)

Try to run this in a JSFiddle. The strict mode does not allow you to create global variables accidentally. The above script will throw the error:

`ReferenceError: assignment to undeclared variable message.`

But if you try the same script without *strict* mode, you will not get the same error. I.e. the following script write 'Hello World' in the console:

```
1 message = 'Hello World';  
2  
3 console.log(message);
```

(the above code snippet online)

## Function formal arguments with same names

Now let's try the following example:

```
1 function foo(a, b, b) {  
2     console.log(a, b, b);  
3 }  
4  
5 foo(2, 3, 5);
```

(the above code snippet online)

If you try this script, you will get the following message printed in the console:

```
1 2 5 5
```

(the above code snippet online)

You can see that the function definition `foo` has two formal arguments with the same name. When we call it with `foo(2, 3, 5)`, then `b`, inside the function, takes the value 5 and not 3.

If you try the same example in strict mode:

```
1 'use strict';  
2  
3 function foo(a, b, b) {  
4     console.log(a, b, b);  
5 }  
6  
7 foo(2, 3, 5);
```

(the above code snippet online)

you will get the error: `SyntaxError: duplicate formal argument b.`

As you can see, strict mode requires formal arguments to be unique.

## Octal literals

Now let's try this script:

```
1 var number = 015;  
2  
3 if (number === 13) {  
4     console.log("number 015 is the octal representation of number 13");  
5 }
```

(the above code snippet online)

The literal `015`, as you can see above, it represents an octal number. In fact, it is the octal representation of the decimal number 13. Run the above script and you will see the console message `number 015 is the octal representation of number 13` being printed.

However, a lot of novice developers don't know that a literal number starting with `0` is actually an octal number. For that reason, the strict mode raises an error for such literals. Try this:

```
1 'use strict';
2
3 var number = 015;
4
5 if (number === 13) {
6   console.log("number 015 is the octal representation of number 13");
7 }
```

(the above code snippet online)

If you run the above script, you will see the following error being raised:

```
1 SyntaxError: "0"-prefixed octal literals and octal escape sequences are deprecate\
2 d; for octal literals use the "0o" prefix instead
```

(the above code snippet online)

If you want to use an octal literal while in strict mode, you will have to prefix the literal with 0o:

```
1 'use strict';
2
3 var number = 0o15;
4
5 if (number === 13) {
6   console.log("number 015 is the octal representation of number 13");
7 }
```

(the above code snippet online)

## Reserved words

The strict mode is preparing the way to ECMAScript 2015 (ES6). For that reason, when a script is running in strict mode, then the following words cannot be used as variable names:

- implements
- interface
- let
- package
- private
- protected
- public
- static
- yield
- const

There are some other rules that *strict* mode enforces. [You can read about them here](#)

## Tasks and Quizzes

**Before you continue, you may want to know that:** You can sign up to [Tech Career Booster](#) and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

### Tasks

You will need to implement a web page that will actually be a game.

1. The player needs to click on a UFO image while this is visible.
2. Whenever the user clicks on a UFO image it gets one more point.
3. The UFO images are displayed on the page on random positions. And they are visible for a random duration.
4. User has 1 minute to play the game. His objective is to click on as many UFO images as he can.

You can see this page in action here:

<https://player.vimeo.com/video/194307766>

Some hints that will help you build your web page:

1. You will need to have 3 files: `task.html`, `assets/javascripts/task.js` and `assets/stylesheets/task.css`.
2. The UFO image is already given to you. Look at the resources folder for this chapter.
3. You will need both Twitter Bootstrap and jQuery.
4. You will need a div that would contain the clickable image. This will generally be invisible (hint: class `hidden`).
5. You will need a div to display the current score.
6. You will need a div to display the current time left.
7. Make sure that you are using box sizing border-box for all the elements.
8. The div that would display the UFO will need to change position dynamically and randomly. Since its position will be on specific `top` and `left` coordinates, we are suggesting that you will use absolute position for it. But this also means that the containing element, `body` will need to have `relative` position.
9. On JavaScript layer:
  1. Break the problem into smaller functions. This is a suggested list of functions:
    1. a `displayImage` function would be responsible to display the image. This needs to calculate the new coordinates of the image (`top` and `left` properties) and then display the image on this position.
    2. a `hideImage` function would be responsible to hide the image.
    3. a `random` function would be responsible to calculate the number of milliseconds an image would be displayed.
    4. an `increaseScore` function would be responsible to increase the current score. You will also need a variable to keep track of the current score.

5. a `displayScore` function would be responsible to display the current score.
  6. a `handleImage` function would be responsible to
    1. hide the current image by calling `hideImage`.
    2. display the image on new position by calling `displayImage`.
    3. register a new timeout action to call `handleImage` again on a new random timeout.
  7. a `displayTimeLeft` function that would display the current time left. You will need a variable to hold the time left too.
  8. a `decreaseTimeLeft` function that would decrease the time left by 1.
  9. an `initializeTimer` function that would display the time left and register an interval to update it every second.
  10. a function that would be called on `click` event that would occur on the UFO image. This would
    1. first hide the current image, by calling the `hideImage()` function.
    2. increase the score, by calling the `increaseScore` function.
    3. display the new score, by calling the `displayScore` function.
2. Note that `.width()` method on `$(window)` returns the current window width.
  3. The `.height()` method on `$(window)` returns the current window height.
  4. Window width and window height are necessary in order to make sure that you calculate a new random position for the image that falls in between the window dimensions.
  5. The `Math.random()` returns a random number between 0 (including) and 1 (excluding).
  6. You will need to use the `.css()` method to set the `top` and the `left` properties of the image within the `displayImage` function.
  7. You will need to use the method `.removeClass()` to remove the `hidden` class when you want to make the image visible. Add this class back when you want to make the image invisible.
  8. The duration of the image being visible, needs to be between 0 and 2 seconds. A `Math.random() * 2000` will give you that in miliseconds.
  9. The `setTimeout()` needs to be used to handle the image at random intervals.
  10. The `setInterval()` needs to be used to set the time countdown.
  11. The `clearInterval()` needs to be used to clear the time countdown.

We understand that this might be a difficult task. It's your last task before we close the JavaScript section.

**Important:** Your code needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

## 13 - JavaScript Workers

### Summary



JavaScript Web Workers

This chapter is teaching you how you can write JavaScript code that will be executed in the background, without blocking/freezing the UI. This tool is adequate for long-running heavy tasks, because it does not affect the performance of the web page.

### Learning Goals

1. Learn about Web Workers
2. Learn how to send messages to a worker.
3. Learn how to get messages back from a worker.
4. Learn how to implement a primitive prime number calculator using a worker.
5. Learn how to implement an image manipulation program using a worker.
6. Learn how to handle errors raised by workers.

### Web Workers

Web Workers is an [HTML5 specification](#) that allows JavaScript to execute background tasks. These scripts are going to be executed independently from the UI handling JavaScript code.

Workers are supposed to be heavy-running tasks that require a lot of memory and CPU load. Hence, they are not meant to be spawned in slew numbers.

## Prime Numbers Example

Let's create a worker that will help us calculate prime numbers. Here is the main HTML content:

```

1 <html>
2   <head>
3     <meta charset="utf-8">
4     <meta name="viewport" content="width=device-width, initial-scale=1">
5     <title>Primer Numbers With Web Workers</title>
6     <script src="assets/javascripts/main.js"></script>
7   </head>
8
9   <body>
10    <h1>Prime Numbers with Web Workers</h1>
11    <p>
12      The highest prime number discovered so far is <span id="result"></span>
13    </p>
14  </body>
15 </html>
```

(the above code snippet online)

When you load this page (load it using a simple HTTP server, like `python -m SimpleHTTPServer`), it will start displaying the prime numbers. This is done with the help of two scripts:

1. The `assets/javascripts/main.js` script:

```

1 // assets/javascripts/main.js
2 //
3 window.onload = function() {
4   var worker = new Worker('assets/javascripts/worker.js');
5   worker.onmessage = function(event) => {
6     document.getElementById('result').textContent = event.data;
7   }
8 }
```

(the above code snippet online)

On line 4, we instantiate a new `Worker`. It needs to be given a script that will be the actual code that will be executed. We will see that next. The `worker` holds a reference to the worker and we can register a message handler. This is what we do on line 5. We install a message handler and hence we give `worker` a way to send back messages to the main script of execution. The message will arrive as `event.data`.

1. The `assets/javascripts/worker.js` script:

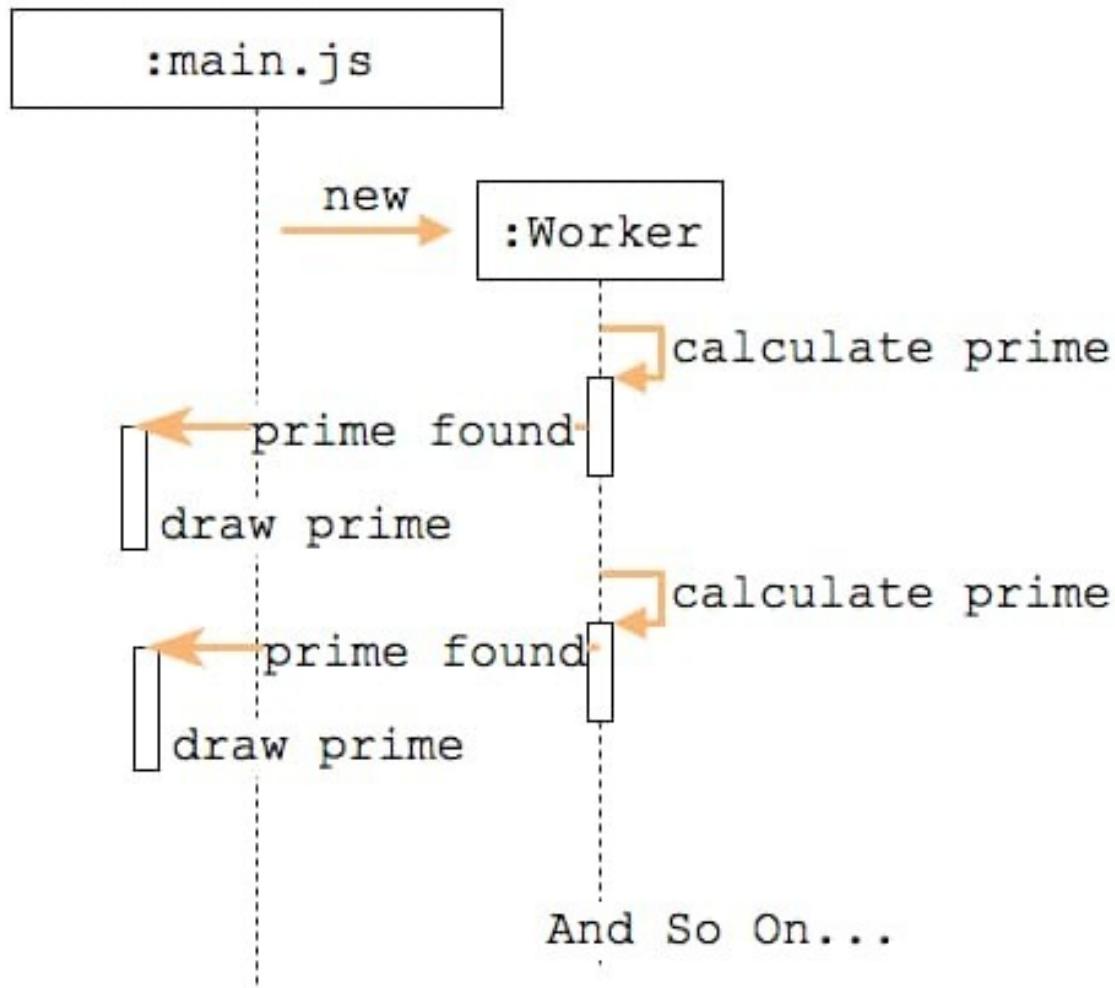
```
1 let n = 1;
2
3 while (true) {
4     n += 1;
5     let prime = true;
6     for (let i = 2; i <= Math.sqrt(n); i += 1) {
7         if (n % i === 0) {
8             prime = false
9         }
10    }
11    if (prime) {
12        postMessage(n);
13    }
14    if (n > 1000000) {
15        break;
16    }
17 }
```

(the above code snippet online)

The worker will run a loop until `n` becomes larger than a big number (line 14). For each `n` it will run a heavy unoptimized algorithm to find out whether the `n` is a prime number or not. If `n` is a prime number, it will call `postMessage(n)` (line 12) and will pass the prime number back to the main script (which will then print that number).

The following picture explains what happens to the above program:

## Web Worker running in the background



### Web Worker Running In the Background

The Worker runs in its own thread. And it is using `postMessage()` to send a message back to the main script every time it encounters a prime number. The main script does nothing but taking the prime number and drawing it in the corresponding part of the HTML page.

## Off-Main Thread Image Manipulation

Let's write another example in which we use a worker to manipulate images out of the main thread of execution. This is a good use case for workers, because manipulating images, usually, consumes a lot of resources and you wouldn't like to block the UI thread.

Here is the HTML part:

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="utf-8">
5          <meta name="viewport" content="width=device-width, initial-scale=1.0">
6          <title>Off-Main Thread Image Manipulation with Workers</title>
7          <script src="http://code.jquery.com/jquery-2.2.4.min.js"></script>
8          <script src="assets/javascripts/main.js"></script>
9      </head>
10     <body>
11
12         <p>
13             <label>
14                 Type an image URL to decode
15                 <input type="url" id="image-url" list="image-list"/>
16                 <datalist id="image-list">
17                     <option value="http://localhost:8000/assets/images/lion.jpg"></option>
18                     <option value="https://html.spec.whatwg.org/images/drawImage.png"><opt\
19 ion>
20                     <option value="https://html.spec.whatwg.org/images/robots.jpeg"><optio\
21 n>
22                     <option value="https://html.spec.whatwg.org/images/arcTo2.png"></option>
23                 </datalist>
24             </label>
25         </p>
26
27         <p>
28             <label>
29                 Type a filter to apply
30                 <select id="filter">
31                     <option value="none">none</option>
32                     <option value="grayscale">grayscale</option>
33                     <option value="brighten">brighten by 20%</option>
34                     <option value="darken">darken by 20%</option>
35                 </select>
36             </label>
37         </p>
38
39         <p>
40             <button id="apply-filter">Apply</button>
41         </p>
42
43         <canvas id="output">
44         </canvas>
45
46     </body>
```

47 </html>

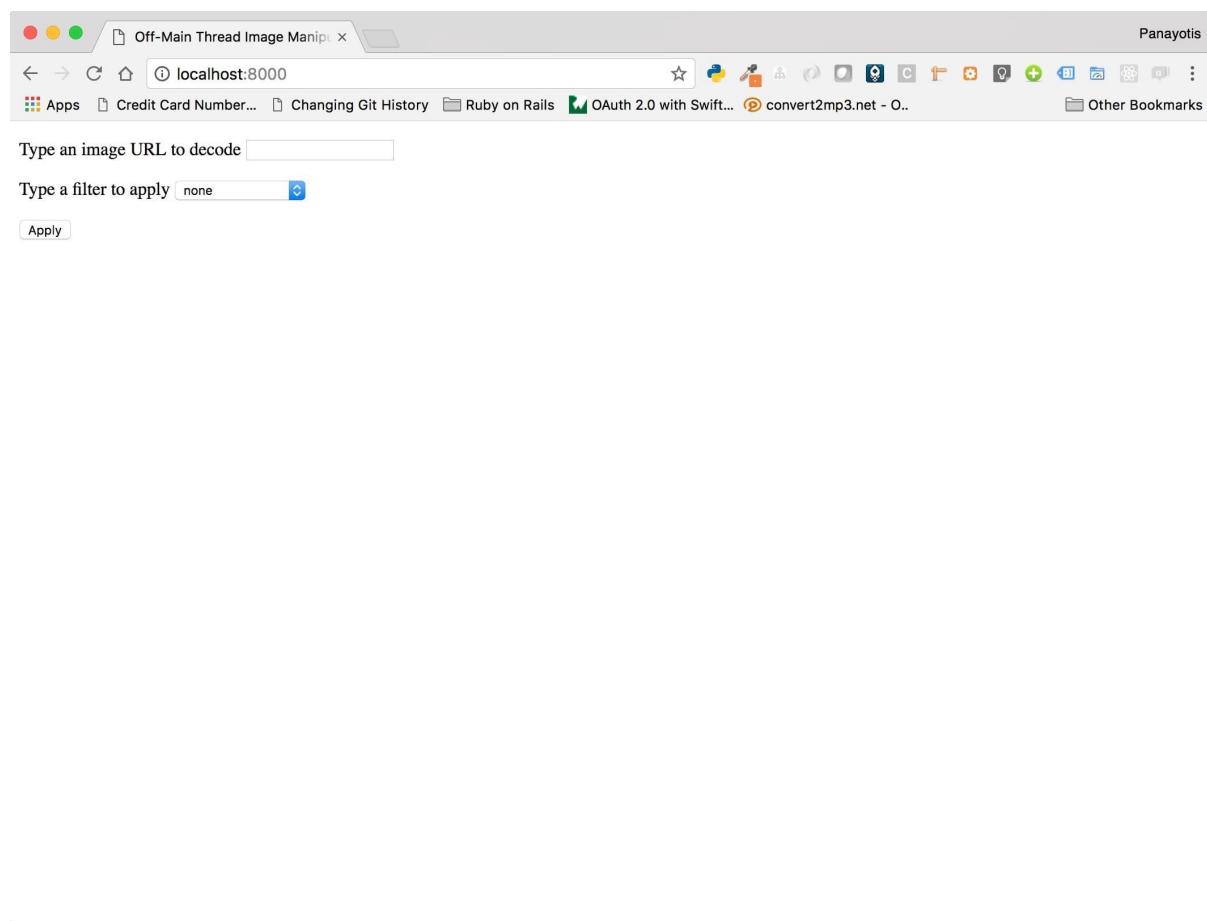
(the above code snippet online)

As you can see, we are also referencing jQuery from a CDN source (line 7).

The HTML page is composed of four elements:

- (1) An input field that the user gives a URL to an image.
- (2) A data list that is there to help user give their image URL input. It has a list of URLs that the user can choose from. However, even if there is a list, the user can give on (1) any URL that they like. Note that there are four images referenced. One is local to the project and you can find it in the resources of the chapter. The others are references to external resources.
- (3) A drop down with available filters to apply.
- (4) A button to Apply the filter.

Here is a screenshot of the page as soon as it starts:



### Web Worker Image Manipulation - Start

The idea behind this small project is that whenever the user clicks on `Apply`, the image specified is drawn in the canvas area. Also, if the image is already displayed/drawn in the canvas, the `Apply` button applies the filter selected.

Let's see the `main.js` script:

```
1 (function($) {
2     $(document).ready(function() {
3         var img;           // Holds reference to "img" element created.
4         var imageData;    // The imageData loaded. The ImageData of the currently lo\
5 aded Image.
6         var originalData; // The originalData. The ImageData of the original image. \
7 Used to reset back to original image, when user selects "none" for filter
8         var context;      // This will be the canvas context. The image is drawn wit\
9 hin a canvas.
10        var canvas = $('#output')[0]; // This is the output canvas
11
12        var worker = new Worker('assets/javascripts/worker.js')
13        worker.onmessage = function(event) {
14            // https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2\
15 D/putImageData
16            // When we get the new image data from the Worker, we just paint the image.
17            context.putImageData(event.data.imageData, 0, 0);
18            imageData = event.data.imageData;
19        }
20
21        $('#apply-filter').on('click', function() {
22            var src = $('#image-url').val();
23            var $filter = $('#filter');
24
25            if (img && img.src === src) {
26                // Tell worker to do work.
27                worker.postMessage({imageData: imageData, filter: $filter.val(), original\
28 Data: originalData});
29            }
30            else {
31                // I create an image element inside canvas
32                img = document.createElement('img');
33                img.crossOrigin = 'Anonymous';
34                img.src = src;
35
36                $(img).on('load', function() {
37                    var imageElement = this;
38                    var width = imageElement.width;
39                    var height = imageElement.height;
40                    if (width === 0 || height === 0) {
41                        return;
42                    }
43
44                    // We draw the image on the canvas
45                    canvas.width = width;
46                    canvas.height = height;
```

```
47
48     context = canvas.getContext('2d');
49     context.drawImage(imageElement, 0, 0);
50
51     originalData = context.getImageData(0, 0, width, height);
52
53     // Tell worker to do work.
54     worker.postMessage({imageData: originalData, filter: $filter.val(), ori\
55 ginalData: originalData});
56   });
57 }
58 );
59 });
60 );
61 });
62 })(jQuery);
```

(the above code snippet online) This is what this script does:

(1) On line 9, it creates the background Worker. The worker relies on script `worker.js` which we will see next.

(2) Then, on line 10, the main script installs a handler to deal with messages that will be coming from the worker back to the main script. Basically, the message will contain the new image data. Hence, the responsibility of the main script is to put that data on the canvas. This is done on line 13. Also, on line 14, we save the current set of data. We will need it to give it to the next Apply.

(3) Then we install a `click` handler for the `Apply` button. This has two branches. First one, deals with the `Apply` when there is already an image displayed. It just only asks the worker to calculate the new pixels and tells us when ready. Very simple. The second branch, is a little bit more complicated because it needs to create the `img` element (line 24), set its `src` attribute to the value that the user has specified in the input element (line 29) and also makes sure that the image can be loaded from third-party resources (by setting the `img.crossOrigin` to `Anonymous`). Then, this branch needs to wait for the image to load before doing further work. That is why it installs a `load` handler for the `img` element created (line 31). The `load` handler needs to take the dimensions of the image and adapt the dimensions of the canvas accordingly (lines 40 and 41). Then it draws the image on the canvas (line 43 and 44). Finally, it takes the pixels of the image (line 46) and asks worker to process it, according to the filter selected.

Before we see how the `worker.js` script has been implemented to manipulate the data / pixels of the image loaded, let's see the page in action:

### JavaScript Worker - Manipulating Images

Let's now see the `worker.js` script:

```
1 // Worker
2 //
3
4 // "filters" define a set of functions that
5 // do work on image data. For example, the 'brighten' takes the pixels
6 // and multiply their values by 1.2, so that the whole picture becomes
7 // brighter.
8 //
9 var filters = {
10     none: function(data, originalData) {
11         for(var i = 0; i < data.length; i++)
12             data[i] = originalData[i];
13     },
14
15     grayscale: function(data) {
16         for(var i = 0; i < data.length; i++) {
17             var r = data[i];
18             var g = data[i + 1];
19             var b = data[i + 2];
20
21             // CIE luminance for the RGB
22             // The human eye is bad at seeing red and blue, so we de-emphasize them.
23             // An algorithm to turn to gray. See more here: http://www.tannerhelland.co\
24             m/3643/grayscale-image-algorithm-vb6/
25             data[i] = data[i + 1] = data[i + 2] = 0.2126 * r + 0.7152 * g + 0.0722 * b;
26         }
27     },
28
29     brighten: function(data) {
30         for(var i = 0; i < data.length; i++) {
31             data[i] *= 1.2;
32         }
33     },
34
35     darken: function(data) {
36         for(var i = 0; i < data.length; i++) {
37             data[i] /= 1.2;
38         }
39     }
40 }
41
42 // This is what the worker will do when it gets a message from Main script
43 //
44 self.onmessage = function(event) {
45     var imageData = event.data.imageData;
```

```
47 var filter = event.data.filter;
48 var originalData = event.data.originalData;
49
50 filters[filter](imageData.data, originalData.data);
51
52 // after having filtered the image, it will send the data back to the main scri\
53 pt
54 self.postMessage({imageData: imageData});
55 }
```

(the above code snippet online)

On lines 44 to 53, the last ones, we specify what the worker will do when it receives a message from the main script. Basically, depending on the `filter` requested, worker calls the corresponding function from the function pool `filters`. For example, if the `filter` is `brighten`, then the `filters[brighten]()` function will be called.

The `filters` functions do nothing but changing the values of the pixels of the image at hand. Note that `originalData` is only important to the `none` function which resets the data to their original value.

The worker installs a message handler on line 44 and at the end of this handler it responds back to the main script by calling `postMessage()`. This is going to trigger the execution of the code on lines 10 to 15 from the `main.js` script.

## Features Available To Workers

You need to know that the following features are available to workers:

- The `navigator` object
- The `location` object
- `XMLHttpRequest` that would allow you Ajax requests.
- `setTimeout()` / `clearTimeout()` and `setInterval()` / `clearInterval()`.
- The *Application Cache*
- The `importScripts()` method that allows you to import external scripts.
- The ability to *spawn other web workers*.

However, the following features are not available:

- The DOM, because accessing the DOM is not thread-safe. This means that you cannot manipulate the DOM by adding / removing nodes, for example.
- The `window` object.
- The `document` object.
- The `parent` object.

## Handling Errors

It is important that you add error handling code when writing code with workers. This is very easy. All you have to do is to attach a handler on the `error` event on your worker.

Let's see an example:

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Error Handling</title>
7     <script src="assets/javascripts/main.js"></script>
8   </head>
9   <body>
10  </body>
11 </html>
```

(the above code snippet online)

The above is a very simple web page that is using a worker. Load it using a simple HTTP server like `python -m SimpleHTTPServer`.

The `main.js` script is the following:

```

1 window.onload = function() {
2
3   var worker = new Worker('assets/javascripts/worker.js');
4   worker.addEventListener('message', function(event) {
5     console.log("Worker sent message: ", event.data.message);
6   });
7   worker.addEventListener('error', function(event) {
8     console.log("Something went wrong: Line:", event.lineno, "filename:", event.f\
9 ilename, "message:", event.message);
10  });
11  worker.postMessage({message: 'Hello World'});
12};
```

(the above code snippet online)

Note: We are using `addEventListener()` to attach handlers in this example.

As earlier, we attach a handler for `message` event. This is the handler that will get a message from the worker. It will just print the message (line 5).

The new thing is the handler for the `error` event. This is the error handling function and this is the suggested way to handle errors related to workers. The event argument (line 7) has three very useful properties:

- lineno
- filename
- message

These can be used to troubleshoot errors related to your worker code.

Now see the actual worker script:

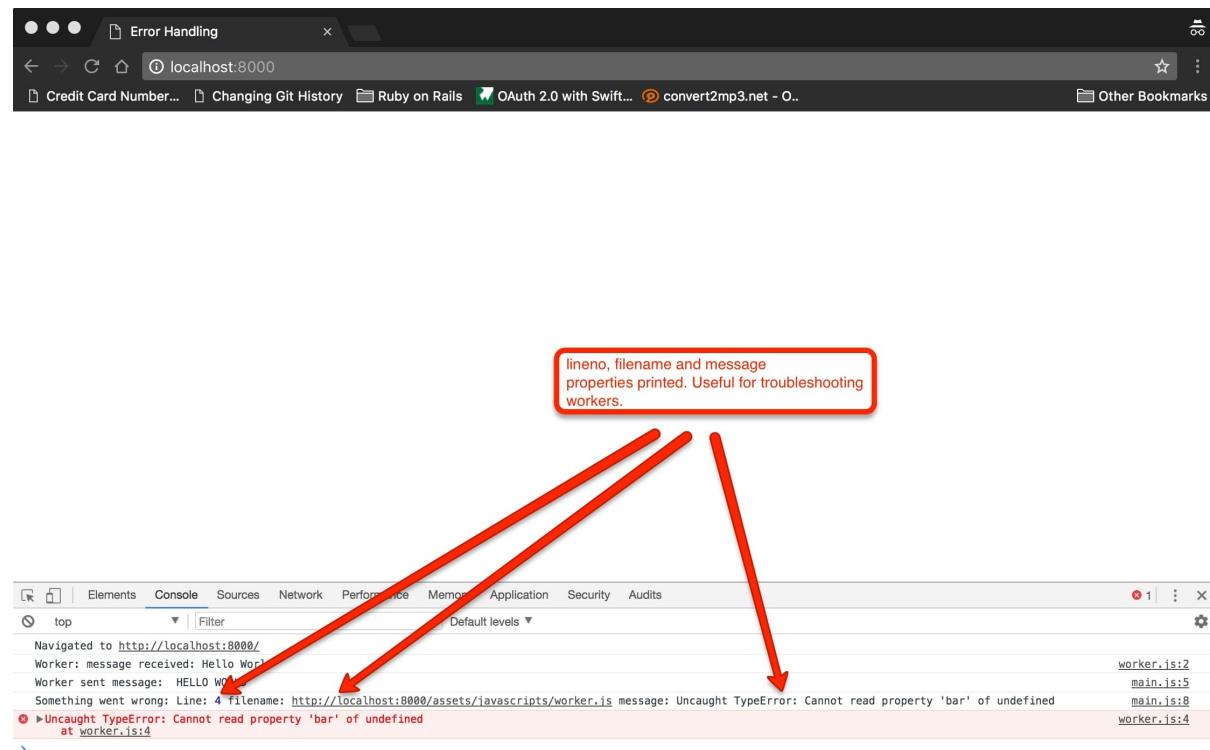
```

1 addEventListener('message', function(event) {
2   console.log("Worker: message received:", event.data.message);
3   postMessage({message: event.data.message.toUpperCase()});
4   postMessage({message: event.data.foo.bar});
5 });

```

(the above code snippet online)

On line 4, we do an error, on purpose. If you load the page on your browser and watch the console logs you will see this:



### Web Workers - Handling Errors

You can see how we take advantage of the three error properties and print useful information in the console.

## Shared Workers

The specification talks about *shared workers* too. We will not discuss them here.

## Closing Note

We have seen how workers can handle our heavy tasks in the background. This is not a feature that one employs very frequently, but it is very good to know because it shows you how JavaScript is progressing into becoming a language to deal with things other than only UI stuff on the browser.

## Tasks and Quizzes

**Before you continue, you may want to know that:** You can sign up to [Tech Career Booster](#) and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

### Task Details

This is a JavaScript snippet that reverses a string in a very inefficient way. It takes 1 second to reverse every letter of the string.

```
1 var input = '12345678';
2 var result = '';
3 var i = input.length - 1;
4 var interval = null;
5
6 function reverse() {
7   if (i < 0) {
8     clearInterval(interval);
9   }
10  else {
11    result += input[i--];
12    console.log("result:", result);
13  }
14}
15
16 interval = setInterval(reverse, 1000);
```

You can try that on a JSBin or on JSFiddle.

But, it can be used to simulate a long-running job that could be executed in the background.

Your task is the following:

Create a worker that reverses a string. Every time it reverses a character, it notifies back the main script, which prints the current result.

Here is a video that demonstrates the page you need to build.

[JavaScript Worker - Task Reversing a String](#)

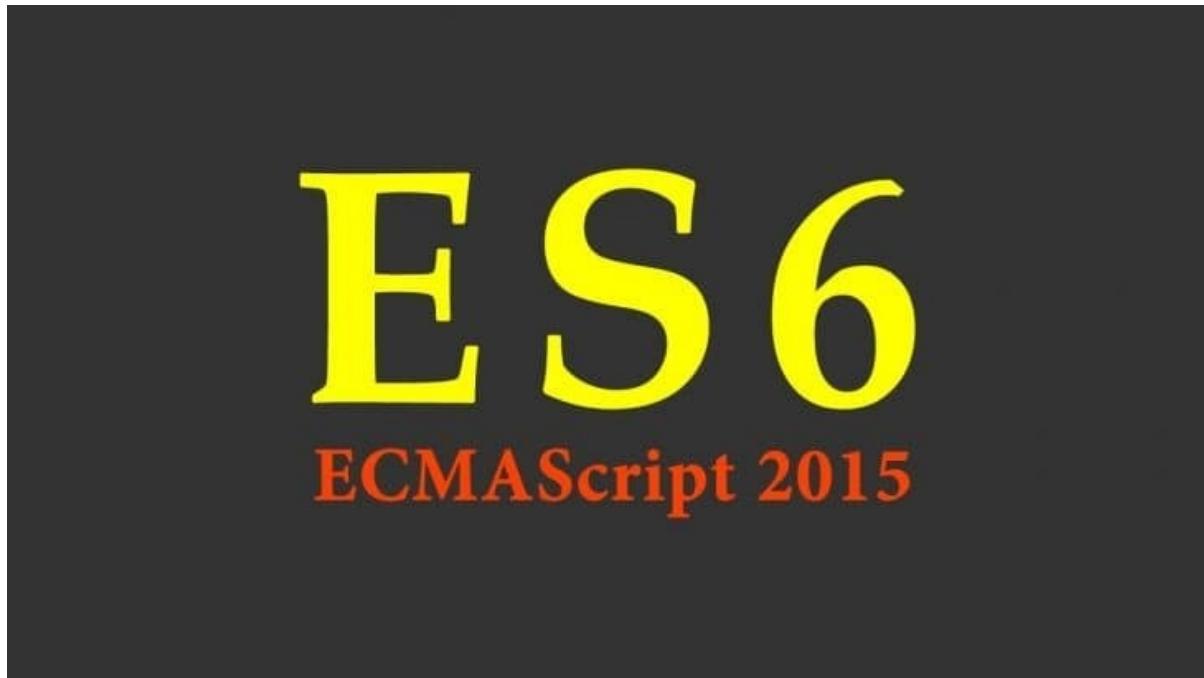
Note that the reverse implementation presented in the JavaScript code above, uses a 1 second delay between characters processing. Use a much smaller value in order to make the reverse process for long tests finish quickly. Like what you see in the video above.

Also, make sure that while the reversing process is taking place, the Reverse button should be disabled. When the process finishes, then Reverse button should become enabled again.

**Important:** upload your code to your Github account and let your mentor know about it.

## 14 - ES6 - ECMAScript 2015 - Advanced JavaScript

### Summary



ES6 - ECMAScript 2015

JavaScript is a programming language that is evolving very rapidly. It is trying to become more classic in terms of Object-Oriented Programming but still keep its huge power.

This chapter is a **difficult one**, especially because it has a lot of small tasks that you have to accomplish at the end.

But it is equally and **absolutely necessary** if you want to be up-to-date with the latest JavaScript language improvements.

### Learning Goals

1. Learn about the `let` keyword.
2. Learn about the `const` keyword.
3. Learn about property definition methods.
4. Learn how to make a property of an object constant/read-only.
5. Learn about the exponentiation operator.
6. Learn about the template/interpolated strings.
7. Learn about the arrow function shorthands.
8. Learn about the enhanced object literals.
9. Learn about array destructuring assignment.
10. Learn about object destructuring assignment.
11. Learn about the *rest* and *spread* operators.

12. Learn how you can define default values for function parameters.
13. Learn how to make copies of an object.
14. Learn about classes.
15. Learn how to create classes with declarations and expressions.
16. Learn about constructors.
17. Learn about static methods.
18. Learn how to extend a class.
19. Learn about accessor properties.
20. Learn about mix-ins.
21. Learn how you can use Sets and Maps in ES6
22. Learn about symbols.
23. Learn about iterators.
24. Learn about generators.
25. Learn about iterables.
26. Learn about the `for-of` statement.
27. Learn about modules.
28. Learn about promises.
29. Learn about the `fetch()` API.
30. Learn about the `async/await` construct.

## Introduction

JavaScript is an implementation of the ECMAScript (ES) standard. In 2015, the ES2015 or ES6 was released which included a lot of new features. Most of them are now supported by the modern browsers and runtime engines, like Node.js. The previous version of JavaScript that we have learned so far, was ES5.

Let's see some of the most frequently used new features of ES6.

### let Statement

The `let` allows you to declare variables that are limited in scope to the block, statement, or expression on which it is used. This is unlike the `var` keyword, which defines a variable globally, or locally to an entire function regardless of block scope.

Let's see the difference with an example:

```
1 for(var i = 0; i < 5; i++) {  
2     console.log(i);  
3 }  
4  
5 console.log("After for loop: ", i);
```

(the above code snippet online)

You can see that the variable `i` is declared at the `for` level. However, it is globally accessible, even after the scope of the loop. If you run the above in a JSBin, you will get something like this:

```

1 0
2 1
3 2
4 3
5 4
6 After for loop: 5

```

(the above code snippet online)

Now, let's turn the var to let:

```

1 for(let i = 0; i < 5; i++) {
2   console.log(i);
3 }
4
5 console.log("After for loop: ", i);

```

(the above code snippet online)

and try to run the program again. You will see that you get an error "ReferenceError: i is not defined", which means that i is not accessible outside of the scope of the for block.

So, variables declared with let are less susceptible to errors.

## const Statement

The const declare references to values and the identifier cannot ever be reassigned another value. Hence, they take their values at the initialization point:

```
1 const foo = 'bar';
```

(the above code snippet online)

The foo will always have the value bar. Also, its scope works like the scope of let. However, watch out for the trap. Having a reference const, does not make the properties of the object pointed to by the reference being constant too.

For example, the program

```

1 const student = {
2   firstName: 'John',
3   lastName: 'Paul'
4 }
5
6 student.lastName = 'Woo';
7 console.log(student.lastName); // => 'Woo'

```

(the above code snippet online)

works perfectly fine and finally prints Woo, without a problem.

How can we make the properties of a constant object constants too? One way you can do that is with Object.defineProperty() method. Try the following:

```

1 const student = {
2   firstName: 'John'
3 }
4
5 Object.defineProperty(student, 'lastName', {value: "Paul", writable: false});
6
7 student.lastName = 'Woo';
8 console.log(student.lastName); // => "Paul"

```

(the above code snippet online)

That made the property readonly. However, in ES6, you can just use a property definition method. Try this:

```

1 const student = {
2   firstName: 'John',
3   get lastName() {
4     return 'Paul';
5   }
6 }
7
8 student.lastName = 'Woo';
9 console.log(student.lastName); // => 'Paul'

```

(the above code snippet online)

The above makes sure that `lastName` remains `Paul` even if we try to change it to `Woo`. Also, if you try to print the properties of `student` object, you will see that `lastName` is part of the properties:

```

1 const student = {
2   firstName: 'John',
3   get lastName() {
4     return 'Paul';
5   }
6 }
7
8 student.lastName = 'Woo';
9 console.log(student.lastName); // => 'Paul'
10
11 console.log(Object.getOwnPropertyNames(student));

```

(the above code snippet online)

## Exponentiation Operator

We know that `Math.pow()` is the most common way to calculate the power of a number. For example:

```
1 console.log(Math.pow(5, 2)); // => 25
```

(the above code snippet online)

The above prints the 2nd power of 5 (`52`).

However, EcmaScript2016 introduced the *exponentiation operator*, a binary operator that does not oblige you to use a method call to calculate the power of a number, like it happens in many other languages:

```
1 console.log( 5 ** 2 ); // => 25
```

(the above code snippet online)

The exponentiation operator is `**` and the above snippet will print 25, like before.

Note that `**` operator has the highest priority among the binary operators.

```
1 console.log( 10 * 5 ** 5 ); // => 31250
```

(the above code snippet online)

The above prints 31250, because it first calculates `5 ** 5` (gives 3125), and then multiplies by 10.

## Template Strings

This is a new facility that allows us to create concatenated strings of dynamic content. Like the interpolated strings in Ruby. Let's see an example:

```
1 var firstName = "John";
2 var lastName = "Travolta";
3
4 console.log(` ${firstName} ${lastName}`);
```

(the above code snippet online)

If you run the above in JS Bin, you will get "John Travolta". The interpolation happens with the help of backticks `<code></code>` surrounding the string construction expression, and the `{}$`, which encloses a JavaScript expression that evaluates to something that can be turned into a string.

Also, the backticks can be used to create multiline strings:

```
1 var output = `This is a very long string that
2 spans multiple lines.`;
3
4 console.log(output);
```

(the above code snippet online)

## Arrows

Arrows are function shorthands that use the symbol sequence `=>`. They are not completely equivalent to the ES5 function expressions, which are still here. For example, they do not have their own `this`, `arguments`, `super` or `new.target`. The arrow functions are best suited for non-method functions and they cannot be used as constructors.

Below you can see an example of an arrow function:

```
1 (a) => a * a;
```

(the above code snippet online)

This is equivalent to the ES5 version:

```
1 (function(a) {  
2     return a * a;  
3 });
```

(the above code snippet online)

If you try the `(a) => a * a;` on a JSBin, you will not see anything happening. But you can try this:

```
1 var result = (a) => a * a;  
2 console.log(result(2));
```

(the above code snippet online)

This will print 4 on the console window.

And another very frequently encountered example is the following. Instead of writing this:

```
1 var numbers = [1, 2, 3, 4];  
2 var transformed = numbers.map(function(v, i) {  
3     return v * i;  
4 });
```

(the above code snippet online)

You can write this:

```
1 var numbers = [1, 2, 3, 4];  
2 var transformed = numbers.map((v, i) => v * i);
```

(the above code snippet online)

Do you see how we give the argument to the `map()` function call? We give an arrow function definition.

Note that if the function is defined as taking one argument, then you don't have to wrap it in rounded parenthesis.

```
1 const doubleIt = a => a * 2;  
2  
3 console.log(doubleIt(5));
```

(the above code snippet online)

Note that if the function body spans multiple lines, rather than just a return statement like above, then you need to be careful to enclose the body in curly braces and explicitly define the `return` statement (if it needs to return something):

```
1 const complexFunction = (a, b, c) => {  
2     const aa = a * 2;  
3     const bb = b * 18;  
4     const cc = c + 10;  
5     return aa + bb + cc;  
6 }
```

(the above code snippet online)

## Enhanced Object Literals

We have learned about object literals in the `Objects` chapter. ES6 has some new features that we can use with object literals:

1. prototype at construction
2. shorthand for `foo: foo` assignments
3. defining methods
4. making `super` calls
5. computing property names with expressions, i.e. dynamic property names

The following example demonstrates the use of each one of those:

```
1 class Animal {  
2     constructor(name) {  
3         this.name = name;  
4     }  
5  
6     makeNoise() {  
7         console.log("I am Animal " + this.name);  
8     }  
9 }  
10  
11 var belt = "brown";  
12  
13 var dog = {  
14     __proto__: new Animal('peter'),
```

```
15
16     makeNoise() {
17         super.makeNoise();
18         console.log("I am a dog!");
19     },
20
21     belt,
22
23     [5 + 6]: 'hello'
24 };
25
26
27 console.log(dog.name);
28 dog.makeNoise();
29 console.log(dog.belt);
30 console.log(dog[11]);
```

(the above code snippet online)

On line 13, you can see the object `dog` being defined literally. And then on lines 27 till 30, we call various properties and methods on it.

1. Line 14 is an example of the first feature, prototype at construction. On the example, we make the `dog` inherit from `Animal`.
2. Line 21 is an example of the second feature, shorthand for `foo: foo` assignments. In this particular case, with `belt` we basically say `belt: belt`.
3. Line 16 is an example of a method definition.
4. Line 17 is an example of making a call to `super` in order to call the functionality inherited from the super class.
5. Line 23 is an example of dynamically computing the name of the property. The name of the property ends to be `11`.

## Destructuring Assignment

The destructuring assignment allows us to assign specific elements of an array or specific properties of an object.

### Array Destructuring

Let's see an example with array:

```
1 var [a, ,b] = [1, 2, 3];
2
3 console.log(a);
4 console.log(b);
```

(the above code snippet online) The first line of the snippet above, tells that we are interested only on the first and third element of the array. Hence, if you run the above, you will get this:

```
1 1
2 3
```

(the above code snippet online)

You can also have default values. Here is another example:

```
1 var [a, , b, c = 7] = [1, 2, 3];
2
3 console.log(a);
4 console.log(b);
5 console.log(c);
```

(the above code snippet online)

Now, the c is going to have the value 7, because it is not assigned a value on line 1.

We can also assign the rest of an array to a variable. Look at this example:

```
1 var [a, ...b] = [1, 2, 3, 4, 5, 6];
2
3 console.log(a);
4 console.log(b);
```

(the above code snippet online)

If you run this in a JSBin, you will get the value [2, 3, 4, 5, 6] for the variable b.

## Object Destructuring

Except for array, destructuring works with objects too.

For example, we can quickly get into variables part of the object properties:

```

1 var order = {date: '2017-11-16', number: '123456', customer: 'John Woo'};
2 var {date, number} = order;
3
4 console.log(date);
5 console.log(number);

```

(the above code snippet online)

The second line is the new way one can get the values of the properties `date` and `number` from the object `order`.

## Object Rest/Spread Properties

ES6 allows us to use object *rest* and *spread* properties. Here is an example of using the *rest operator*:

```

1 const x = {a: 1, b: 2, c: 3, d: 4, e: 5};
2 const {a, b, ...rest} = x;
3
4 console.log("a = ", a); // => prints a = 1
5 console.log("b = ", b); // => prints b = 2
6 console.log("rest = ", rest); // => prints rest = {c: 3, d: 4, e: 5}

```

(the above code snippet online)

On line 2, we can get the `a` and `b` properties of `x` explicitly, and the rest of the properties of `x` are assigned to the `rest` constant. We do that by using the `...` rest operator as prefix to the `rest` identifier.

And here is a *spread* operator example:

```

1 const a = 10;
2 const b = 8;
3 const x = {foo: 'bar', mary: 'woo'};
4 const y = {a, b, ...x};
5
6 console.log("y = ", y); // prints y = {a: 10, b: 8, foo: "bar", mary: "woo"}

```

(the above code snippet online)

On line 4, we use the properties of `x` with the *spread* prefix. Hence, all of its own properties are being spread and given to `y`, as extra properties, after, `a` and `b`.

## Function Parameter Default Values

It goes along with destructuring that we can now set easier the default values for the parameters of a function.

```

1 const sayHello = ({name = 'Stranger'} = {}) => {
2   console.log(`Hello ${name}`);
3 }
4
5 sayHello();
6 sayHello({name: 'John'});

```

(the above code snippet online)

The `sayHello()` function above, can be called without any argument. The `Stranger` value will be used for the `name`.

## Object.assign()

The `Object.assign()` method is used to copy the values of all enumerable own properties of an object into another. Actually, the source object can be many objects. Let's see an example:

```

1 let object1 = {first_name: 'John', last_name: 'Woo'};
2 let object2 = Object.assign({}, object1);
3
4 console.log(object2);

```

(the above code snippet online)

If you run that on a JSBin or a JSFiddle you will see that it prints `{first_name: "John", last_name: "Woo"}`. The `object2` has the same own properties like the `object1`. Or, you can try another example that is using two source objects:

```

1 let object1 = {first_name: 'John', last_name: 'Woo'};
2 let object2 = {first_name: 'Maria', salary: 2000};
3 let object3 = Object.assign({}, object1, object2);
4
5 console.log(object3);

```

(the above code snippet online)

If you run the above, you will see that `object3` has the properties that both `object1` and `object2` have. Also, the `object2#first_name` was finally applied on top of the `object1#first_name`, because it was applied last.

## Classes

We have already seen how we can define objects in the `Objects` chapter. With *Classes*, ES6 is trying to make the syntax more familiar to those that they have been used in programming with more standard OOP tools. They make the inheritance in JavaScript much more clear and simple, if compared to the prototypical inheritance that one could use until ES5.

## Defining Classes

Classes are specialized functions, and as we do have function declarations and function expressions, we also have *class declarations* and the *class expressions*.

### Class Declarations

Here is an example of a class declaration, using the `class` keyword.

```
1 class Point {
2     constructor(x, y) {
3         this.x = x;
4         this.y = y;
5     }
6 }
7
8 var p = new P(5, 8);
9 console.log(p);
```

(the above code snippet online)

If you run this in your JSBin, you will get something like this:

```
1 [object Object] {
2     x: 5,
3     y: 8
4 }
```

(the above code snippet online)

### Class Expressions

There is another way you can define a class, by using a class expression. Look at the following example:

```
1 // unnamed
2 var Point = class {
3     constructor(x, y) {
4         this.x = x;
5         this.y = y;
6     }
7 };
8
9 // named
10 var Rectangle = class Rectangle {
11     constructor(height, width) {
12         this.height = height;
```

```
13     this.width = width;
14 }
15 };
16
17 var p = new Point(5, 8);
18 console.log(p);
19
20 var r = new Rectangle(20, 30);
21 console.log(r);
```

(the above code snippet online)

If you run the above in a JSBin, you will get something like this:

```
1 [object Object] {
2   x: 5,
3   y: 8
4 }
5
6 [object Object] {
7   height: 20,
8   width: 30
9 }
```

(the above code snippet online)

which is actually expected. In the following picture, you can see the difference in unnamed vs named case:

```
// unnamed
var Point = class {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
};

// named
var Rectangle = class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};

var p = new Point(5, 8);
console.log(p);

var r = new Rectangle(20, 30);
console.log(r);
```

#### Class Expressions Named and Unnamed

## constructors

Each class can have a special method called `constructor`. But there can only be one. The `constructor` is invoked when we call `new` to instantiate an object.

## static methods

A class may have a static method. These are class-level methods and they do not require an instance in order to be called. Actually, they cannot be called via an instance of a class. In order to declare a static method we use the keyword `static`.

```
1 class ConcatAndUpcase {
2   static doIt(s1, s2) {
3     return (s1 + s2).toUpperCase();
4   }
5 }
6
7 console.log(ConcatAndUpcase.doIt("foo", "bar"));
```

(the above code snippet online)

In the above example, you can see how we call the method `doIt` on the last line. We have defined it as `static` inside the body of the `class`. If you run this in a JSBin, you will get this "FOOBAR".

## Extending a Class

We can extend a class definition and inherit stuff from the super class. We use the keyword `extends` in order to do that. Let's see an example:

```
1 class Animal {
2     constructor(name) {
3         this.name = name;
4     }
5     run() {
6         console.log(`I am ${this.name} and I am running`);
7     }
8 }
9
10 class Dog extends Animal {
11     bark() {
12         console.log(`I am ${this.name} and I will bark and then run`);
13         this.run();
14     }
15 }
16
17 const animal = new Animal('Peter');
18 console.log("I will ask animal to run...");
19 animal.run();
20
21 const dog = new Dog('Wolf');
22 console.log("I will ask dog to run...");
23 dog.run();
24 console.log("I will ask dog to bark...");
25 dog.bark();
```

(the above code snippet online)

We have created the class `Animal` and then we have created the class `Dog` which inherits from `Animal`. You can also see that `Dog` instance has access to the method `run()` which is defined in `Animal` class.

If you run the above in a JSBin, you will get something like this:

```
1 I will ask animal to run...
2 I am Peter and I am running
3 I will ask dog to run...
4 I am Wolf and I am running
5 I will ask dog to bark...
6 I am Wolf and I will bark and then run
7 I am Wolf and I am running
```

(the above code snippet online)

## Accessor Properties

Classes allow you to prefix your functions with `get` or `set` and turn the function name into an accessor property identifier. Here is an example:

```
1 class Animal {
2     constructor(color, name) {
3         this.color = color;
4         this.name = name;
5     }
6
7     get fullDescription() {
8         return `${this.name} - ${this.color}`;
9     }
10
11    set fullDescription(value) {
12        [this.name, this.color] = value.split(' - ').map((element) => element.trim());
13    }
14 }
15 const dog = new Animal('white', 'max');
16
17 console.log('dog name =', dog.name, 'dog color =', dog.color);
18 console.log('dog fullDescription =', dog.fullDescription);
19
20 dog.fullDescription = 'flox - brown';
21 console.log('dog name =', dog.name, 'dog color =', dog.color);
22 console.log('dog fullDescription =', dog.fullDescription);
```

(the above code snippet online)

The `fullDescription` can now be used to get and set the property `fullDescription`, like you do for the other properties of the object, without using the `()` suffix. For example, on line 18, we use `dog.fullDescription` instead of `dog.fullDescription()`. And on line 20, we use `dog.fullDescription = 'flox - brown'` instead of `dog.fullDescription('flox - brown')`.

## Mix-ins

We have seen how we can extend a base class and create a sub-class. The `extends` keyword allows us to inherit from only one class. That means that only single-inheritance is allowed. But, we can use the technique of *mix-ins* if we want to incorporate into a class pieces of code from different modules.

Here is an example:

```

1 const calculatorMixin = Base => class extends Base {
2   calc() {
3     console.log("calc() in Base");
4   }
5 };
6
7 const randomizerMixin = Base => class extends Base {
8   randomizer() {
9     console.log("randomizer() in Base");
10  }
11 };
12
13 class Foo extends(calculatorMixin(randomizerMixin(class {}))) {
14   bar() {
15     console.log("bar() in Foo");
16   }
17 }
18
19 var f = new Foo();
20 f.calc();
21 f.randomizer();
22 f.bar();

```

(the above code snippet online)

On lines 1 to 5 and 7 to 11, we define two mix-ins. The first one defines the method `calc()` and the second one defines the method `randomizer()`. Then, we define a class, `Foo`, that extends from the mix-ins. We enlist the mix-ins one after the other in a nested scheme, and the inner most definition is `class {}`.

Then on lines 20, 21 and 22 we call methods on a `Foo` instance. These methods come either from the `Foo` class itself or from the mix-ins.

If you run the above in a JSBin, you will see this:

```

1 calc() in Base
2 randomizer() in Base
3 bar() in Foo

```

(the above code snippet online)

## Sets and Maps

Before ES6, sets and maps were implemented using special techniques. For example:

```
1 let setOfColors = Object.create(null);
2
3 setOfColors.yellow = true;
4 setOfColors.red = true;
5
6 // checking for existence
7 if (setOfColors.brown) {
8   console.log("Brown is there");
9 }
10 if (setOfColors.yellow) {
11   console.log("Yellow is there");
12 }
13 if (setOfColors.red) {
14   console.log("red is there");
15 }
```

(the above code snippet online)

The above example creates an object using `null` as prototype. Hence, no properties will be present except from the ones defined for the particular `setOfColors` object. Then we define the properties `yellow` and `red`. On top of that, setting them to have `true` value is a trick that allows us to check for the existence of a property.

Having done the above, we essentially mimic the `Set` type.

Similarly, in order to mimic a `Map` we did the same trick as for `Set`, but we stored any value for the property:

```
1 let mapOfCustomers = Object.create(null);
2
3 mapOfCustomers.peterWoo = {age: 65, height: 1.5};
4 mapOfCustomers.maríaFoo = {age: 30, height: 1.4};
5
6 console.log(mapOfCustomers.peterWoo.age);
7 console.log(mapOfCustomers.maríaFoo.height);
```

(the above code snippet online)

The `mapOfCustomers` works like a `Map` object. We can retrieve the value of a key like we do at the last two lines.

Using the above technique to work with sets and maps has some disadvantages. For example, all the keys are converted to strings. For example the `5` and the `"5"` will end up being the same property. That might not have been your intention.

```
1 let map = Object.create(null);
2
3 map[5] = 5 * 5;
4
5 if (map["5"] > 0) {
6   console.log("Value of '5' is greater than 0");
7 }
```

(the above code snippet online)

The above will print the message Value of '5' is greater than 0, although the key you added to the map was the integer 5 and not the string "5".

## ES6 Sets

ES6 adds the Set type which is an ordered list of values without duplicates. The set is constructed with the Set() constructor. Items are added to the set with the add() method. The .size property returns the number of elements in the set. Here is an example:

```
1 let setOfColors = new Set();
2
3 // add two colors
4 setOfColors.add("red");
5 setOfColors.add("yellow");
6
7 console.log(setOfColors.size);
8
9 // add existing color, will not change size of set
10 setOfColors.add("yellow");
11
12 console.log(setOfColors.size);
13 setOfColors.delete("yellow");
14
15 console.log(setOfColors.size);
```

(the above code snippet online)

The above is a very simple example of a set of colors. The add() method is used to add elements and the delete() method is used to remove elements from the set.

Note that the .has() method can tell you whether an item is in the set or not:

```

1 let setOfColors = new Set();
2
3 // add two colors
4 setOfColors.add("red");
5 setOfColors.add("yellow");
6
7 console.log(setOfColors.has("yellow"));
8 console.log(setOfColors.has("blue"));

```

(the above code snippet online)

The above will print `true` and then `false`.

The `clear()` method will remove all items from the set. And the `forEach()` method can be used to carry out an action on each one of the elements of the set.

```

1 let setOfColors = new Set();
2
3 setOfColors.add("red");
4 setOfColors.add("yellow");
5 setOfColors.add("blue");
6
7 setOfColors.forEach(value => console.log(value.toUpperCase()));

```

(the above code snippet online)

A set can be easily converted to an array. Hence, you will be able to access it by index:

```

1 let setOfColors = new Set();
2
3 setOfColors.add("red");
4 setOfColors.add("yellow");
5 setOfColors.add("blue");
6
7 let arrayOfColors = [...setOfColors];
8 for(let i = 0; i < arrayOfColors.length; i++) {
9   console.log(arrayOfColors[i]);
10 }

```

(the above code snippet online)

On line 7 above, we convert the set to an array. We are using the *spread operator* (...).

## ES6 Maps

Maps are constructed with the `Map()` constructor. The `set()` method is used to save an element into the map. The `get()` method is used to get the value of an element from the map. Here is an example:

```
1 let mapOfCustomers = new Map();
2
3 mapOfCustomers.set("peterWoo", {age: 65, height: 1.5});
4 mapOfCustomers.set("mariaFoo", {age: 55, height: 1.4});
5
6 console.log(mapOfCustomers.get("mariaFoo"));
```

(the above code snippet online)

Like set does, map has the method `has()` to tell whether a key exists in the map. Also, the `delete()` method removes a key from the map, and the `clear()` method removes all the keys. Also, the `size` property returns the number of keys in the map.

## Symbols

Symbols are a primitive type joining the existing primitive types: strings, numbers, booleans, null, and undefined.

### Creating a Symbol

You do that by using the global function `Symbol` like in this example here:

```
1 let firstName = Symbol("First name");
2 let person = {};
3
4 person(firstName) = "Panos";
5
6 console.log(person(firstName));
```

(the above code snippet online)

See how we use the `Symbol` function to create a Symbol. Also, see how we use it to create a property for the object `person`.

Note that the argument to `Symbol()` function is optional. It is a description that is only used for debugging purposes. It is recommended that you always use one.

Symbols are used whenever you can use a computed property name.

### Symbols Can Be Shared

It is very useful to keep track of the symbols that you create and JavaScript helps with that with a global symbol registry. But, in order to have access to it, you will have to use the function `Symbol.for("<a string identifier for the symbol>")`. Let's see the following example:

```
1 let firstName = Symbol.for("First Name");
2 let person = {}
3
4 person(firstName) = "Panos";
5 console.log(person(firstName));
```

(the above code snippet online)

The `Symbol.for("First Name")` call is searching into the global symbol registry for a symbol that is identified by the given string, i.e. by the string “First Name”. If so, the method returns the existing symbol. If the symbol does not exist, the new symbol is created, it is registered in the global symbol registry and then it is returned.

The method that returns the key of a symbol, if given the symbol itself, it is the `keyFor()` method. There is an example that you can run in a JSBin:

```
1 let firstName = Symbol.for("First Name");
2 console.log(Symbol.keyFor(firstName));
```

(the above code snippet online)

If you run the above, you will get the string “First Name” as output, which is the key for the symbol `firstName`.

Note that the global symbol registry is one for the whole JavaScript application instance. So, don’t make any assumptions about keys presence. It may be a good idea if you used a namespacing technique to name the keys of the symbols you save in the global registry. For example, instead of `let order = Symbol.for("Order");`, better do `let order = Symbol.for("mycompany.Order");`.

## Iterators, Generators and Iterables

### Iterators

Iterators are very popular in many programming languages. We have learnt about the Ruby `.each` method which can let you iterate over the elements of an Array:

```
1 [1, 2, 3].each do |element|
2   puts "element: #{element}"
3 end
```

(the above code snippet online)

The iterators shift the programming paradigm from `for` loops that require a tracking position variable to a technique that requires less typing and generates more clear code.

ES6 specifies the interface that an Iterator object should expose.

Let’s see a JavaScript loop that iterates over the elements of an array, written in the traditional ES5 style:

```

1 const students = ['John', 'Mary', 'Paul', 'Andrew'];
2
3 for (let i = 0; i < students.length; i++) {
4   console.log(`Student ${i}): ${students[i]}`);
5 }

```

(the above code snippet online)

Will now implement the above using iterators. Iterators are objects with specific interface designed for iteration. All iterators need to have a `.next()` method that returns a `result` object like this `{value: <the value fetched>, done: [true|false]}`. Hence, the result tells us whether there are more values to return (done property) and the actual value returned (value property). Note that when done is `false`, value does not contain the last item of the collection. It contains whatever we want the iterator to return as a general-iterator-return value, or `undefined` if we have designed our iterator to return nothing.

Having said the above, the following code creates an iterator but using ES5 and not ES6:

```

1 function createIterator(collection) {
2   return {
3     index: 0,
4     next: function() {
5       return this.index >= collection.length ?
6         {value: undefined, done: true} :
7         {value: collection[this.index++], done: false};
8     }
9   }
10 }
11
12 var iterator = createIterator([1, 2, 3]);
13 var iterator2 = createIterator([5, 6, 7]);
14 console.log(iterator.next());
15 console.log(iterator2.next());
16 console.log(iterator.next());
17 console.log(iterator2.next());
18 console.log(iterator.next());
19 console.log(iterator2.next());
20 console.log(iterator.next());
21 console.log(iterator2.next());

```

(the above code snippet online)

As you can see, it is not very difficult to write an iterator that complies with the interface requirements of ES6.

## Generators

However, ES6 allows us to use another tool, generators, in order to implement iterators easier.

A generator is a function that returns an iterator. The generator is indicated by the \* following the `function` keyword.

The following is a simple iterator created using a generator:

```

1 function *createIterator(collection) {
2   yield 1;
3   yield 2;
4   yield 3;
5 }
6
7 const iterator = createIterator();
8
9 console.log(iterator.next());
10 console.log(iterator.next());
11 console.log(iterator.next());
12 console.log(iterator.next());
```

(the above code snippet online)

The `yield` keyword tells which value should be returned every time the `.next()` method is called on the iterator. The generator should yield all the values in the order they should be returned by `.next()`.

Here is another example in which the generator takes as input a collection:

```

1 function *createIterator(collection) {
2   for(let i = 0; i < collection.length; i++) {
3     yield collection[i];
4   }
5 }
6
7 const iterator = createIterator([1, 2, 3]);
8
9 console.log(iterator.next());
10 console.log(iterator.next());
11 console.log(iterator.next());
12 console.log(iterator.next());
```

(the above code snippet online)

As you can see above, the implementation of the iterator using a generator is much simpler.

Note that `yield` can only be used inside of generators. Hence the following would return a syntax error:

```

1 function *createIterator(collection) {
2   collection.forEach((element) => yield element);
```

```
3  }
4
5 const iterator = createIterator([1, 2, 3]);
```

## Iterables

Iterables are closely related to iterators. An iterable is an object with a `Symbol.iterator` property. The well-known `Symbol.iterator` property should reference a function that returns an iterator for the given object, something like the create iterators we saw earlier. Note that, in ES6, all arrays, sets, maps and strings are iterables and they have a default iterator specified. For example:

```
1 const numbers = [1, 2, 3];
2 const iterator = numbers[Symbol.iterator]();
3
4 console.log(iterator.next()); // prints {value: 1, done: false}
5 console.log(iterator.next()); // prints {value: 2, done: false}
6 console.log(iterator.next()); // prints {value: 3, done: false}
7 console.log(iterator.next()); // prints {value: undefined, done: true}
```

(the above code snippet online)

The above demonstrates how, in ES6, the arrays have an iterator already defined in their `Symbol.iterator` property.

If a collection, like the arrays, has an iterator defined at the well-known property `Symbol.iterator`, then we can use the `for-of` statement to loop over the elements of the collection. Look at the following example:

```
1 const students = ['John Woo', 'Mary Foo', 'Paul Papas', 'Peter Pan']
2
3 for(let student of students) {
4   console.log(student)
5 }
```

(the above code snippet online)

This `for-of` loop works on iterables. It calls the `.next()` method on the iterator of the iterable and stores the `value` value into the variable used for the iteration (`student` in our example above). The repetition continues until `done` returned by the iterator is `true`.

If you run the above program, you will get the following in the console:

```
1 John Woo
2 Mary Foo
3 Paul Papas
4 Peter Pan
```

(the above code snippet online)

## Creating Iterables

Objects created by developers are not iterables by default. However, you can make them iterables, by setting their `Symbol.iterator` property to be a Generator.

Let's see an example:

```

1 let collection = {
2     items: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12],
3     *[Symbol.iterator]() {
4         for(let item of this.items) {
5             if(item % 2 === 0) {
6                 yield item;
7             }
8         }
9     }
10 };
11
12 for(let x of collection) {
13     console.log(x);
14 }
```

(the above code snippet online)

If you run the above, you will see even numbers between 1 and 12 to be printed in the console. The `Symbol.iterator` property has been set up to be a Generator, that yields the even elements of the `items` array. Then on line 12, we can use the `for-of` statement to print the even members of the collection.

Note: ES6 collections, like arrays, maps and sets, they come with some built-in iterators that are quite useful.

1. `entries()` - Returns an iterator whose values are key-value pairs.
2. `values()` - Returns an iterator whose values are the values of the collection.
3. `keys()` - Returns an iterator whose values are the keys contained in the collection.

## Modules

ES6 includes the features of modules which is another way of encapsulating code. Let's see in more detail what they are.

Modules are here to solve the problem of namespacing and global space pollution. Before ES6, all JavaScript code loaded by the browser was globally shared. When the applications started to grow, that was a problem of maintenance and debugging.

Modules are JavaScript files that are loaded in a different mode, a *module* mode. This is different to the way JavaScript loads the *scripts*, which is the *script* mode.

Hence, a JavaScript file can be loaded

1. as a Module, or

## 2. as a Script

These are the main properties of loading a file as a module:

- (1) The code of a module is executed in strict mode. So, you don't have to write "use strict"; at the beginning of your file.
- (2) Variables that are created at the top level of the file aren't added to the shared global scope. On the other hand, when a script is executed in *script* mode, any top level variables are global and they can be accessed from anywhere.

This global space pollution can be demonstrated with the following code:

1. The `index.html` file. Create the HTML file with name `index.html` with the following content:

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Global variable demo</title>
5      <script src="assets/javascripts/script1.js"></script>
6      <script src="assets/javascripts/script2.js"></script>
7    </head>
8    <body>
9
10   </body>
11
12 </html>
```

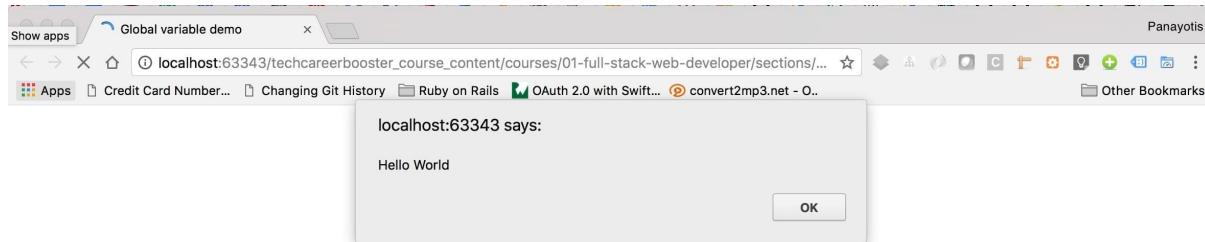
2. The `assets/javascripts/script1.js` file. Create the JavaScript file `script1.js` inside the folder `assets/javascripts`.

```
1  var globalVariable = "Hello World";
```

3. The `assets/javascripts/script2.js` file. Create the JavaScript file `script2.js` inside the folder '`assets/javascripts`'.

```
1  alert(globalVariable);
```

Save all the files and load the `index.html` page on your browser. You will see something like this?



### Global Variable Demo

This proves that your `script2.js` has access to the variable declared inside the `script1.js`. This will not be true with the modules and we are going to demonstrate that later on.

(3) The value of `this` at the top level is `undefined`.

With scripts the value of `this` is the object `Window`. On the other hand, the value of `this`, at the higher level, is `undefined` inside modules.

(4) Modules must export anything that should be available to code outside of the module.

Hence, if the module wants the variable `globalVariable` to be accessible from code outside of the module, it has to export that variable. We will see examples a little bit later on.

(5) Modules may import bindings from other modules.

The exported stuff from a module can be imported into another using a specific statement.

## Example of Modules

Let's now see an example of using modules. Create a new page `index.html` with the following content:

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <title>Modules Demo</title>
5     <script type="module" src="assets/javascripts/script1.js"></script>
6     <script type="module" src="assets/javascripts/script2.js"></script>
7   </head>
8   <body>
9
10  </body>
11 </html>
```

(the above code snippet online)

This is the same file with the earlier `index.html`, but the scripts are loaded as modules. This is done with the attribute `type` that has the value `module`, instead of `text/javascript`. The `type="module"` tells browser to load the scripts as modules.

Otherwise, keep the `scriptX.js` files as before:

The `assets/javascripts/script1.js` file. Create the JavaScript file `script1.js` inside the folder `assets/javascripts`.

```
1 var globalVariable = "Hello World";
```

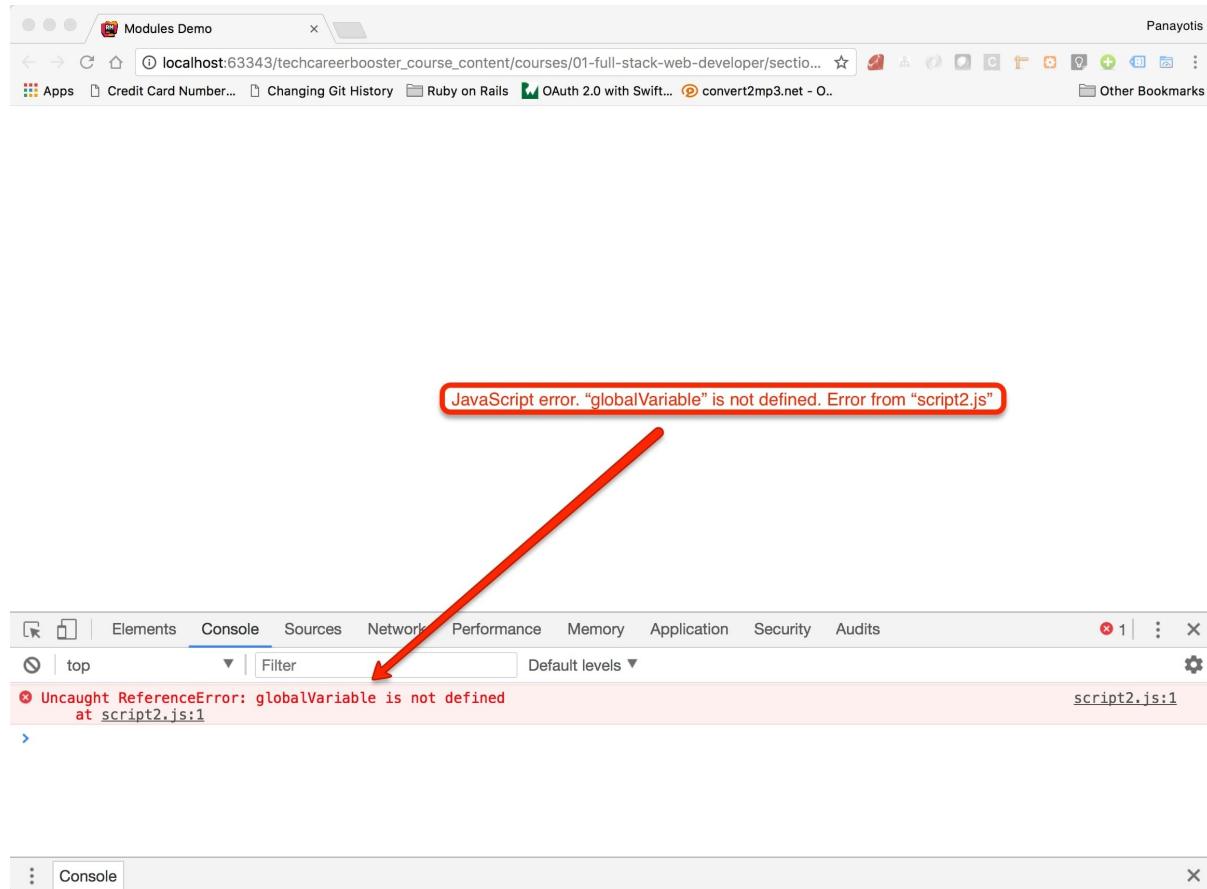
(the above code snippet online)

The `assets/javascripts/script2.js` file. Create the JavaScript file `script2.js` inside the folder ‘`assets/javascripts`’.

```
1 alert(globalVariable);
```

(the above code snippet online)

And then try to load the `index.html` page in your browser. You will not see anything. Actually, if you have your developer tools open, you will see a JavaScript error being logged:



### Global Variable Is Not Defined When Using Modules

As you can tell from the error, the variable `globalVariable` defined inside `script1.js` is not visible outside of it. `script2.js` is trying to access the `globalVariable`, but it fails, because it is not in its scope.

That is the idea of the modules. It encapsulates the code without polluting the global namespace. But, what if the `script1.js` wanted the variable `globalVariable` to be visible outside of it. It would have had to export it. Change the `assets/javascripts/script1.js` file to be as follows:

```
1 export var globalVariable = "Hello World";
```

(the above code snippet online)

We have added only an `export` prefix.

If you try to load the page `index.html` on your browser, once more, you will get the same error again. This is because it is not enough for the `script1.js` to export something. The script that wants to use it, it needs to import it too. Go and change the `assets/javascripts/script2.js` file according to this:

```

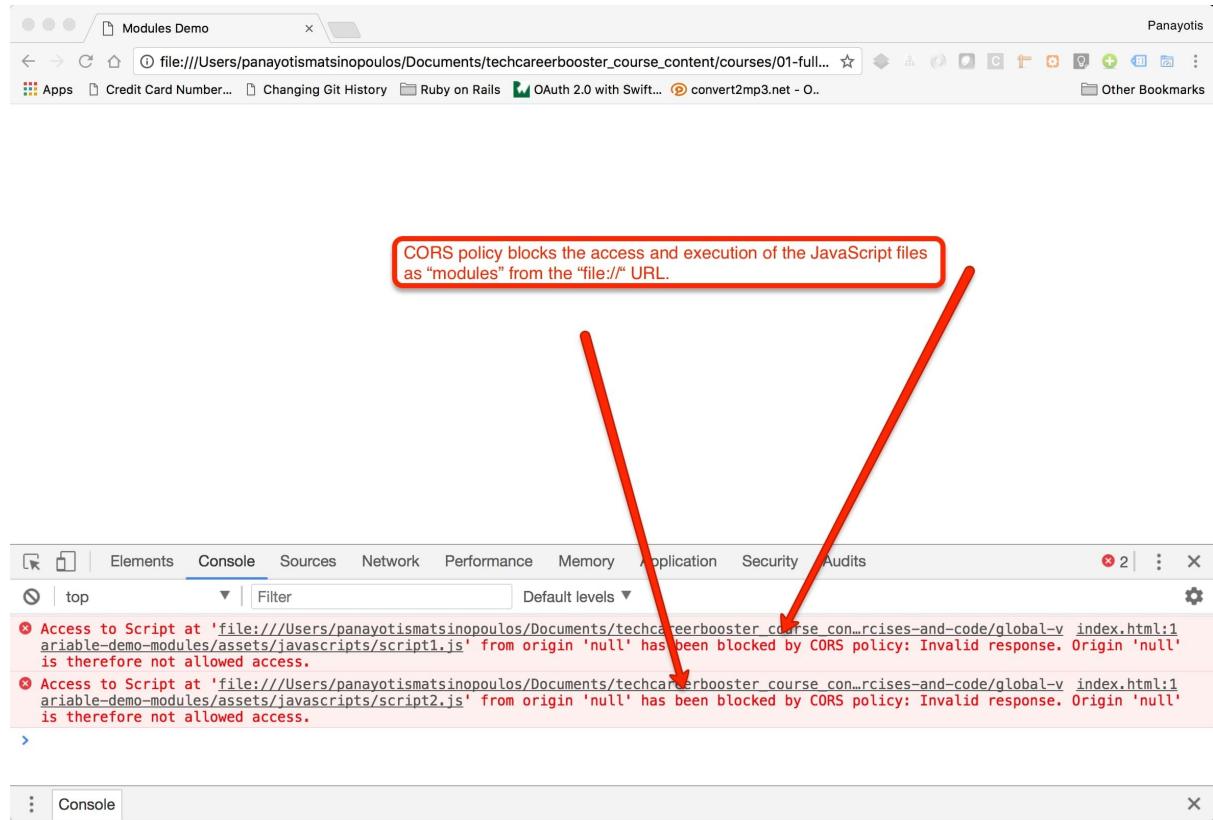
1 import { globalVariable } from './script1.js';
2
3 alert(globalVariable);

```

(the above code snippet online)

You can see that we are declaring an import. We specify which reference to import, the `globalVariable`, and then we tell where to import it from.

However, if you try to load the page `index.html` on your browser, you will not succeed again. This is because your browser does not allow to load modules from your local file system using the `file://` protocol. This is the error that you see:



One way you can bypass this obstacle is to start a local Web server at the root folder of your project. On Mac OS X, this is easily done with the python simple HTTP Server. The following command will start a local Web server listening at port 3000:

```

1 $ python -m SimpleHTTPServer 3000
2 Serving HTTP on 0.0.0.0 port 3000 ...

```

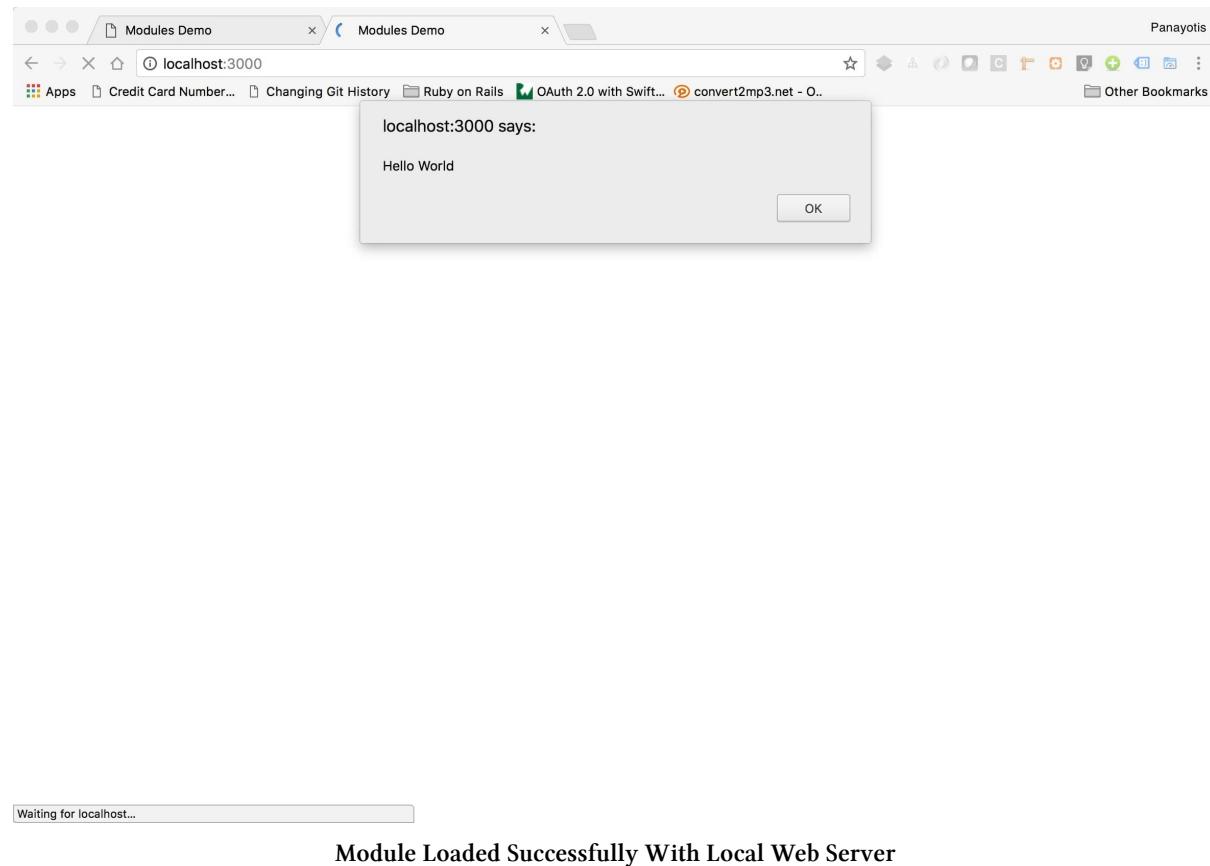
(the above code snippet online)

Now, try to visit the web page: <http://localhost:3000>. You will see the following logs in your terminal window:

```
1 127.0.0.1 - - [24/Nov/2017 07:17:38] "GET / HTTP/1.1" 200 -
2 127.0.0.1 - - [24/Nov/2017 07:17:58] code 404, message File not found
3 127.0.0.1 - - [24/Nov/2017 07:17:58] "GET /robots.txt HTTP/1.1" 404 -
4 127.0.0.1 - - [24/Nov/2017 07:17:58] "GET /assets/javascripts/script1.js HTTP/1.1\
5 " 200 -
6 127.0.0.1 - - [24/Nov/2017 07:17:58] "GET /assets/javascripts/script2.js HTTP/1.1\
7 " 200 -
8 127.0.0.1 - - [24/Nov/2017 07:17:58] code 404, message File not found
9 127.0.0.1 - - [24/Nov/2017 07:17:58] "GET /robots.txt HTTP/1.1" 404 -
```

(the above code snippet online)

And the page will be loaded on your browser:



## Importing All As Single Object

Another importing technique which is very interesting, is the one that allows you to import all references of a module as a single object. Let's see an example of that.

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5
6     <title>Import As Single Object</title>
7
8     <link rel="stylesheet" href="assets/stylesheets/main.css"/>
9
10    <script type="module" src="assets/javascripts/my-math-library.js"></script>
11    <script type="module" src="assets/javascripts/main.js"></script>
12  </head>
13
14  <body>
15
16    <div>Addition: <span id="add"></span></div>
17    <div>Subtraction: <span id="subtract"></span></div>
18
19  </body>
20 </html>

```

(the above code snippet online)

The above page loads two scripts as modules. The first one, `my-math-library.js` defines and exports two functions:

```

1 function add(a, b) {
2   return (a + b);
3 }
4
5 function subtract(a, b) {
6   return (a - b);
7 }
8
9 export { add, subtract };

```

(the above code snippet online)

The export technique that we use here is a little bit different from the previous one. But it is equally valid. We define the exports at the bottom of the module, with a separate `export` statement.

Then, the module `main.js` imports all the exported references as a single object called `MathsAgain`:

```
1 import * as MathsAgain from './my-math-library.js'  
2  
3 window.onload = function(){  
4     let element = document.getElementById('add');  
5     element.innerHTML = MathsAgain.add(5, 6).toString();  
6  
7     element = document.getElementById('subtract');  
8     element.innerHTML = MathsAgain.subtract(11, 8).toString();  
9 };
```

(the above code snippet online)

You can see the import statement on the first line. This is how we can tell that we want to import everything exported, and that we want it imported as a single object.

Note that the `index.html` page of this example includes also a CSS file `assets/stylesheets/main.css`:

```
1 body {  
2     font-size: 3.0rem;  
3 }
```

(the above code snippet online)

If you start your local HTTP Server for this new Web page and you visit the page <http://localhost:3000> you will see this:



Addition: 11  
Subtraction: 3

### Import As Single Object Demo

## Default Exports

You can declare one of your exports as default export of the module. But you cannot declare more than one. Here is an example of a module exporting a default and a non-default reference:

```
1 export let color = "red";
2
3 export default function(num1, num2) {
4     return num1 + num2;
5 }
```

(the above code snippet online)

The above module exports `color` and a default function. In your importing module, you will use this to do the import:

```
1 import sum, { color } from './script.js';
```

(the above code snippet online)

The default import takes place thanks to the `import sum`, i.e. the reference outside of the curly braces. The curly braces specify the non-default imports. Apparently, you can name the default export of a module with whatever name you like, when you do the import.

## Loading Modules in Browsers

We have seen how the modules should be loaded in a browser page. For example:

```
1 <script type="module" src=". /script1.js"></script>
```

(the above code snippet online)

It is the type with value "module" that makes the script load as a module. The difference to type="text/javascript" is that module scripts are always loaded as if the "defer" attribute is specified. This means that the module file begins downloading as soon as the HTML parser encounters `<script type="module">` with an `src` attribute, but does not execute the content until after the HTML document has been completely parsed. Also, you need to take into account that modules are executed in the order they appear in the HTML document.

Generally, you need to have in your mind that the sequence of browser actions with regards to modules is:

1. It first downloads and parses the modules, one by one, in the order they are defined.
2. Waits for the HTML document to be completely parsed
3. Then it executes the modules one by one in the order they are defined

However, you may want to specify the attribute `async` for a module, like you would do for a normal script. If you specify the `async` attribute, then execution of the modules content will take place as soon as possible, but after having parsed all the modules. Hence, although the downloading and parsing of the content of the modules is predetermined, the execution of them is not.

## Loading Modules as Workers

We have seen how we can specify JavaScript Workers in the "JavaScript Workers" chapter. You can load Workers as modules rather than as scripts using the following technique:

```
1 let worker = new Worker("script.js", {type: "module"});
```

(the above code snippet online)

The second argument given to the constructor, is an object that tells how the script should be loaded. When the `type` attribute has the value "module", then the Worker script is loaded as a module. The default value for the `type` attribute is `script`.

There are some differences when loading a Worker as a `module` vs the `script` mode.

1. Firstly, `script` Workers are limited to be loaded from the same origin as the Web page in which they are referenced. `module` Workers have the same limitation, but somehow relaxed, since they allow the loading of files that they are served with the appropriate Cross-Origin Resource Sharing (CORS) headers that allow access.
2. Secondly, although `script` Workers are importing other scripts using `self.importScripts()`, `module` Workers cannot do that. They have to use the `import` statements that we've learnt above.

## Promises

Promises are here to help JavaScript developers write cleaner code when they do asynchronous programming. We already had the events and callbacks, but promises are very useful when the situation is getting complicated.

A promise is a placeholder for the result of an asynchronous request. Instead of subscribing to an event or sending a callback function, a function can return a promise, like in the following example:

```
1 let promise = doBulkWork();
```

(the above code snippet online)

Assume that we have a function `doBulkWork()` that returns a promise, this function returns control to the calling part of the code, immediately, and does its work on the background, asynchronously. What it returns is a *promise*.

We use this promise in order to asynchronously find out about the success or failure of this long running process.

A promise has one of three states:

1. Pending
2. Fulfilled
3. Rejected

A promise is on state *pending* as long as the function that returned that promise has not finished doing its work. If the function finishes, then the promise goes to either *fulfilled* or *rejected* state. Goes to *fulfilled* when everything goes as planned/promised. Goes to *rejected* if something goes wrong.

We can do things on either of the two final states, using the `.then()` method call on the promise. This method takes two functions as arguments, like callbacks:

1. A function that will be executed on promise fulfillment. The function takes one argument which is the object with data that the success logic might find useful.
2. A function that will be executed on promise failure. The function takes again, necessary data to handle the failure.

Both arguments are optional, which means that we can handle either of the cases, or both.

Except from the `.then()` method, a promise also has a `.catch()` method which is called on failure. So, instead of using the second argument to `.then()` in order to handle failure, you can use the `.catch()` method instead.

## Create a Promise and use a Promise

In order to create a Promise, you need to do, more or less, the following:

(1) You need to call the constructor `Promise` and give it as argument a function definition. This function definition takes two arguments. A function definition reference for the *success* and a function definition reference for the *failure* of the promise fulfillment.

This is a blueprint Promise:

```
1 const promise = new Promise((success, failure) => {  
2  
3     .... the implementation of the promise goes here ...  
4  
5});
```

(the above code snippet online)

(2) Usually, the Promises are used as the interface of a long-running job. Let's assume that we have a function that implements a long-running job. This function should return a promise constructed as above:

```
1 const longRunningJob = ((argsForLongRunningJob) =>  
2     new Promise((success, failure) => {  
3  
4         .... the implementation of the promise goes here ....  
5  
6     })  
7 );
```

(the above code snippet online)

(3) The long-running job promise, will have to do the long-running job in the background and make sure that it calls the function `success` on successful completion, or the function `failure` on failing to complete successfully:

```
1 const longRunningJob = ((argsForLongRunningJob) =>  
2     new Promise((success, failure) => {  
3  
4         .... the implementation of the promise goes here ....  
5  
6         if (....state indicates successful completion ...) {  
7             success(data_to_send_out_on_success);  
8         }  
9         else {  
10             failure(data_to_send_out_on_failure);  
11         }  
12     })  
13 );  
14 ;
```

(the above code snippet online)

## Example of a Promise Implementation

That's it. Let's see an example. We will emulate a long-running job with the help of the `setTimeout()` function, which is helpful to execute a function in the future.

```
1  const longRunningJob = (secondsToSleep) =>
2    new Promise((success, failure) => {
3
4      console.log('Long Running Job starts ... ');
5      const timeOut = setTimeout(() => {
6
7        console.log('....Finished doing work');
8
9        clearTimeout(timeOut);
10
11       if (secondsToSleep % 2 === 0) {
12         success("Finished with even");
13       }
14       else {
15         failure("Finished with failure");
16       }
17
18     }, secondsToSleep * 1000);
19
20   })
21
22 ;
```

(the above code snippet online)

This is a long-running job. It is invoked as follows:

```
1 longRunningJob(5);
```

(the above code snippet online)

And sleeps for 5 seconds and then it calls the function given to `setTimeout()`. That function will call either `success` or `failure`, depending on what we have given as argument. And, when calling `success` (on even arguments to `longRunningJob`), it passes some data (`Finished with even`). When calling `failure` (on odd arguments to `longRunningJob`), it passes some other data (`Finished with failure`).

If you build a JSBin, or a JSFiddle with the following piece of code ...

```
1 const longRunningJob = (secondsToSleep) =>
2   new Promise((success, failure) => {
3
4     console.log('Long Running Job starts ... ');
5     const timeOut = setTimeout(() => {
6
7       console.log('....Finished doing work');
8
9       clearTimeout(timeOut);
10
11      if (secondsToSleep % 2 === 0) {
12        success("Finished with even");
13      }
14      else {
15        failure("Finished with failure");
16      }
17
18    }, secondsToSleep * 1000);
19
20  })
21
22 ;
23
24 longRunningJob(6);
```

(the above code snippet online)

... and then you run it, you will see in the console the following two messages:

```
1 Long Running Job starts ...
2 ....Finished doing work
```

(the above code snippet online)

which are the output generated from the lines 4 and 7. There is going to be a delay of 6 seconds between the first and the last line output. This is because the last is executed by the `setTimeout()` function handler.

Now, so far so good, but the piece of code that called `longRunningJob(6)` does not handle the success or the failure of the process being executed. Since `longRunningJob()` returns a `Promise`, we can call `.then()` and `.catch()` to deal with success and failure respectively:

```
1 ...
2 longRunningJob(6)
3   .then(dataOnSuccess => console.log("Job finished successfully and the data is:", \
4   dataOnSuccess))
5   .catch(dataOnFailure => console.log("Job finished with error and the data is:", \
6   dataOnFailure));
```

(the above code snippet online)

Lines 25 and 26 above add handlers for success and failure respectively. Now, if you run the same JavaScript program again, but with the two lines in place, you will also get the following line at the bottom:

```
1 Job finished successfully and the data is: Finished with even
```

(the above code snippet online)

If you run it with an odd argument:

```
1 Job finished with error and the data is: Finished with failure
```

(the above code snippet online)

These demos show you how the caller of the promise can handle the success and failure, and how it can have access to the data that are sent in each case.

By the way, instead of a call to `.catch()` you could have used the second argument to `.then()`:

```
1 ...
2 longRunningJob(6)
3   .then(dataOnSuccess => console.log("Job finished successfully and the data is:", \
4   dataOnSuccess),
5       dataOnFailure => console.log("Job finished with error and the data is:", \
6   dataOnFailure));
```

(the above code snippet online)

which is equivalent.

## Chaining Promises

Now let's see how we can chain promises. Each call to `.then()` or `.catch()` is actually returning another promise. The second promise is resolved only if the previous one has been fulfilled or rejected.

Chaining is used to pass data from one promise to the next. When a success handler returns a value, this becomes available to the next promise.

See this example here:

```

1  const longRunningJob = (secondsToSleep) =>
2    new Promise((success, failure) => {
3
4      console.log('Long Running Job starts ... ');
5      const timeOut = setTimeout(() => {
6
7        console.log('....Finished doing work');
8
9        clearTimeout(timeOut);
10
11      if (secondsToSleep % 2 === 0) {
12        success(secondsToSleep * 2);
13      } else {
14        failure(secondsToSleep - 1);
15      }
16
17    }, secondsToSleep * 1000);
18
19  )
20
21 ;
22
23 // ---- end of long running job definition -----
24
25 longRunningJob(6)
26   .then(dataOnSuccess => dataOnSuccess * 2)
27   .catch(dataOnFailure => console.log("Job finished with error and the data is:", \
28 dataOnFailure))
29   .then(dataOnSuccess => console.log("2nd success handler", dataOnSuccess));

```

(the above code snippet online)

In this example, the success handler of the first promise (see line 26) is returning its argument multiplied by 2, i.e. 24 - since line 12 multiplies 6 already by 2. Then the success handler of the third promise, the one returned by `.catch()`, it prints its argument, which ends to be 24.

## Responding to Multiple Promises

We have learnt how to monitor the progress of a single promise, but, some times, we want to monitor the progress of many promises at the same time. We might be interested in a series of promises and if all of them succeeds. Or we might be interested in a series of promises and if any of them succeed.

### All Succeed or Any Fails

The `Promise.all()` takes as argument a series of promises and it is resolved only if all of the promises are resolved, i.e. only if all of the promises succeed or fail.

This is a rough piece of code that shows how `.all()` basically works:

```

1 const p1 = new Promise((success, failure) => { ... });
2 const p2 = new Promise((success, failure) => { ... });
3 const p3 = new Promise((success, failure) => { ... });
4
5 const promise = Promise.all(p1, p2, p3);
6
7 promise.then(dataForSuccess => { ... });

```

(the above code snippet online)

The success handler on `.then()`, is going to be called only after all 3 promises finish. And the `dataForSuccess` is going to be an Array. The first element will have the success data of the first promise, the second element will have the success data of the second promise and the third element is going to have the success data of the third promise.

If a promise is rejected, then overall promise `promise` does not wait for the other promises to finish (although, they do continue their execution until they finish) and immediately calls the failure handler given in the `.then()` or the function given in `.catch()`. And in that case, the argument is the value that the failing promise pushed to the failure handler. It is not an Array like in the success case.

## Any Succeeds or Any Fails

The `Promise.race()` is used to set up a promise on an array of promises like we did with `Promise.all()`. The difference here is that the overall promise is going to be resolved, with a failure or success, at the moment any of the promises fails or succeeds.

The success and failure handlers will have the value that the succeeding or failing promise has pushed forward.

## Promise Example: `fetch()`

The `fetch()` API is an alternative to jQuery `.ajax()` method. It follows the Promise interface and it has become very popular in JavaScript.

Here is an example that you can run in JSBin or JSFiddle:

```

1 console.log('Requesting details of post with id 1...');
2
3 fetch('https://jsonplaceholder.typicode.com/posts/1')
4   .then((response) => {
5     console.log('...request finished');
6     if (response.ok) {
7       return response.json();
8     } else {
9       console.log('Something went wrong', response);
10    }
11  })

```

```

12     .then((jsonObject) => {
13         console.log(jsonObject);
14         console.log("Post title: ", jsonObject.title);
15     });

```

(the above code snippet online)

In order to use that, you will need to reference the `fetch()` resource. For example, by referencing any of the scripts given [here](#).

What does the above code do?

1. It calls the `fetch()` method passing a URL to fetch data from. (line 3)
2. It then calls `.then()` to handle the successful response. (line 4).
3. The successful fulfillment will have the `response` object holding all the necessary information, whether HTTP Response code was in the success range (200 - 299) or not.
4. That is why, on line 6, we check if the response was `ok` with `if (response.ok) {}`.
5. Then we just call method `.json()` in order to convert the data to a JavaScript object which we pass to the next promise handler.

Note that `fetch()` has some differences to the `.ajax()` call. For example, the success handler will be called even if the response is 4XX or 5XX. Also, it does not send cookies, by default.

You can read more about `fetch()` [here](#).

## async/await

We are now going to see another tool that can help you write asynchronous code look like synchronous code. We are talking about `async/await` constructs, which rely on Promises but they make using them easier.

Look at the following code:

```

1 constgetJSON = () => {
2     return new Promise((success, failure) => {
3         success({foo: 'bar'});
4     });
5 }
6
7 constmakeRequest = () =>
8     getJSON()
9     .then(jsonObject => {
10         console.log(jsonObject.foo);
11         return "done";
12     })
13
14 makeRequest().then(result => console.log(result));

```

(the above code snippet online)

The above is an example of calling `getJSON()` method, which is supposed to be a long-running task that has a `Promise` interface, using the `makeRequest` function.

If you run the above, you will see the words `bar` and `done` printed in the console.

However, things are much simpler with `async/await`:

```

1 const getJSON = () => {
2   return new Promise((success, failure) => {
3     success({foo: 'bar'});
4   });
5 }
6
7 const makeRequest = async () => {
8   const jsonObject = await getJSON();
9   console.log(jsonObject.foo);
10  return "done";
11 }
12
13 makeRequest().then(result => console.log(result));

```

(the above code snippet online)

See how we call `getJSON()` with an `await` prefix. The `makeRequest` also is flagged as `async`. Whatever we have inside the implementation of `makeRequest` it is supposed to be the success handler on calling `getJSON()`. Otherwise, everything else is the same.

One might ask, where shall we have our promise failing handler? You only have to wrap the call to `try { } catch() { }` block:

```

1 const getJSON = () => {
2   return new Promise((success, failure) => {
3     failure({message: 'something went wrong'})
4   })
5 }
6
7 const makeRequest = async () => {
8   try {
9     const jsonObject = await getJSON();
10    console.log(jsonObject.foo);
11    return "done";
12  }
13  catch(error) {
14    console.log(error.message);
15    return "ERROR OCCURRED";
16  }
17 }

```

```
18  
19 makeRequest().then(success => console.log(success),  
20           failure => console.log(failure));
```

(the above code snippet online)

In the example above, the promise fails (by calling `failure({message...})`). Then, we can see the message something went wrong when running the above, proving that the `try { ... } catch() { ... }` works well to catch the promise failures. You can also see the ERROR OCCURED which is printed thanks to the code on line 20.

## Closing Note

That was an introduction to the new JS6 features. Not all of them are supported by all browsers, but if you work with [Babel](#), then you will be able to use them without problem, because Babel makes sure to transpile your work to JS5.

## Tasks and Quizzes

**Before you continue, you may want to know that:** You can sign up to [Tech Career Booster](#) and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

### Task Details - Implement the following tasks

The following tasks differ in complexity. Some of them are simple, some of them require more effort from you. In any case, it is very important for you to practice with these tasks, because they will increase your confidence and skills in ES6.

Note: All your code needs to be uploaded to your Github account.

#### Task 1

Write a class called `Point`, which represents a point in two-dimensional space. A point has `x` and `y` properties, given as arguments to its constructor.

It also has a single method `plus`, which takes another point and returns the sum of the two points, that is, a new point whose `x` is the sum of the `x` properties of the two original points, and whose `y` is the sum of their `y` properties.

Example usage of these classes:

```
1 console.log(new Point(1, 2).plus(new Point(2, 1)));
```

which if run should print `Point{x: 3, y: 3}`.

## Task 2

(Continue work with [http://marijnhaverbeke.nl/talks/es6\\_falsyvalues2015/exercises/#Speaker-upgrade](http://marijnhaverbeke.nl/talks/es6_falsyvalues2015/exercises/#Speaker-upgrade))

Create a class Speaker that exposes the method `#speak()`. Each Speaker instance is initialized with the name of the speaker it represents. The `#speak()` method takes an argument and writes to the console the name of the speaker suffixed with the `#speak()` argument. Example call:

```
1 new Speaker('Bob').speak('come here!');
```

The above should log in the console the string "Bob says: come here!"

Moreover, implement the class Shouter, which should extend Speaker. Instances of class Shouter should have a `#speak()` that prints its argument upcased.

Hence,

```
1 new Shouter('Bob').speak('come here!');
```

should print the string "Bob says: COME HERE!" in the console.

## Task 3

Create a Customer with `firstName` and `lastName` attributes. Also, create accessor properties for the `fullName`. The `fullName` getter should return the `firstName` and `lastName` concatenated using a space in between them. The `fullName` setter should take a string, e.g. "John Woo" and should set the `firstName` and `lastName` accordingly (e.t. "John" for `firstName` and "Woo" for `lastName`).

## Task 4

Below, you can see a piece of JavaScript code.

```
1 const startNode = (type, value, options) => {
2   return YOUR_CODE_HERE
3 }
4
5 console.log(startNode("Identifier", "foo", {
6   sourceProperty: "src",
7   sourceValue: "bar.js"
8 }))
9 // □ {type: "Identifier",
10 //   value: "foo",
11 //   src: "bar.js"}
```

There is a placeholder `YOUR_CODE_HERE`. This is where you should write a piece of JavaScript code so that the `console.log(...)` call that ends this snippet, prints `{type: "Identifier", value: "foo", src: "bar.js"}`.

The idea is that you need to construct the third property of the object returned by `startNode()` so that it has the name given in the `options sourceProperty` value, and value given in the `options sourceValue` value.

## Task 5

Amend the following script so that it gives a `get()` method to the `ids` object. This method should return the next id and increment its next counter.

```
1 const ids = {  
2   next: 0,  
3 }  
4  
5 console.log(ids.get()); // print 0  
6 console.log(ids.get()); // print 1  
7 console.log(ids.get()); // print 2
```

## Task 6

A typical mistake we make in JavaScript is depicted in the following piece of code: We create a number of functions in a loop, and refer to an outside variable from these functions. All of them will end up referring to the same variable, which ends up being incremented to 10. Thus, `callbacks[2]` does not log 2. It logs 10, as do all functions in the array.

```
1 var callbacks = [];  
2 for (var i = 0; i < 10; i++) {  
3   callbacks.push(function() { console.log(i); })  
4 }  
5  
6 callbacks[2]();
```

If you run the above program in a JSBin or in a JSFiddle, you will see that it prints 10 and not 2 as one might have expected. This is because when we invoke `callbacks[2]` function, line 6, the variable `i` the function implementation uses, has already reached the value 10 and this is what is being actually used.

How can we solve this problem using ES6?

## Task 7

Look at the following program:

```
1 const account = {  
2   username: "martin",  
3   password: "xyz"  
4 }  
5  
6 account.password = "secret";  
7 console.log(account.password);
```

Is it wrong? Can we change the password value now that we have the account being const? If we can, what is the alternative? How can we make password being constant?

## Task 8

Write an expression using higher-order array methods (say, filter and reduce) to compute the total value of the machines in the inventory array.

```
1 const inventory = [  
2   {type: "machine", value: 5000},  
3   {type: "machine", value: 650 },  
4   {type: "duck", value: 10},  
5   {type: "furniture", value: 1200},  
6   {type: "machine", value: 77}  
7 ];  
8  
9 const result = YOUR_CODE_GOES_HERE  
10  
11 console.log(result);
```

## Task 9

You need to implement a class that represents a sorted array, and it should be used more or less like this:

```
1 const sortedArray = new SortedArray(A_COMPARISON_FUNCTION);  
2  
3 sortedArray.insert(5);  
4 sortedArray.insert(2);  
5 sortedArray.insert(6);  
6 sortedArray.insert(10);  
7 sortedArray.insert(1);  
8 sortedArray.insert(4);  
9  
10 console.log("array: ", sortedArray.content);
```

As you can see on line 1, we instantiate the `SortedArray` class with a comparison function. The comparison function should be used internally, by the `SortedArray` class, every time we call `insert()`. The insert will have to put its argument in the correct position based on the comparison function.

The comparison function should take two elements as input.

If you want the array to be sorted in ascending order, then, the comparison function should return a negative number if the first element is less than the second, a 0 if the two elements are equal and a positive number if the first element is greater than the second.

If you want the array to be sorted in descending order, the comparison function should return a positive number if the first element is less than second, 0 if they are equal, and a negative number if the first element is greater than the second element.

On line 10, we print the contents of the `.content`, which is the array the `SortedArray` object holds its state in. The string that line 10 prints depends on the comparison function given to `SortedArray` instantiation. If the comparison function compares in ascending mode, then line 10 will print `[1, 2, 4, 5, 6, 10]`. If the comparison function compares in descending mode, then the line 10 will print `[10, 6, 5, 4, 2, 1]`.

In order to solve this exercise, you might want to use the method `.findIndex()` which is described [here](#). It returns the first position of the first element that satisfies the condition encoded inside the function given as argument to `.findIndex()`.

Also, you may want to use the method `.splice()` which is described [here](#). It can be used to insert a new element into an array at a specified position.

## Task 10

Fill in the following piece of code:

```
1 const go = (YOUR_CODE_HERE) => {
2   YOUR_CODE_HERE
3
4   console.log("speed=", speed, "hyperdrive=", hyperdrive);
5 }
6
7 go(); // prints speed= 30 hyperdrive= 8
8 go({speed: 10, hyperdrive: 30}); // prints speed= 10 hyperdrive= 30
```

If you run this program, it should print

```
1 speed= 30 hyperdrive= 8
2 speed= 10 hyperdrive= 30
```

As you can understand, the argument to `go` is optional. Also, the default value for `speed` is 30 and for `hyperdrive` is 8.

Use ES6 syntax to solve this problem.

## Task 11

How can you make `start` having as default value the last index of the array?

```

1 const lastIndexOf = (arr, elt, start) => {
2   for(let i = start; i >= 0; i--) {
3     if (arr[i] === elt)
4       return i;
5   }
6   return -1;
7 }

8

9 console.log(lastIndexOf([1, 2, 1, 2, 5], 1)); // should print 2
10 console.log(lastIndexOf([1, 2, 1, 2, 5], 2)); // should print 3
11 console.log(lastIndexOf([1, 2, 1, 2, 5], 5)); // should print 4
```

Change the above piece of code so that `lastIndexOf()` works as expected. Currently, no matter what element we are searching for, it always returns `-1`. In order to fix that, you need to provide a default value for the `start` argument. The default value should be the last index of the array given as first argument.

## Task 12

The following piece of code includes a function `detectCollision()`, which returns the object that includes the point given.

```

1 const detectCollision = (objects, point) => {
2   for (let i = 0; i < objects.length; i++) {
3     let object = objects[i]
4     if (point.x >= object.x && point.x <= object.x + object.width &&
5         point.y >= object.y && point.y <= object.y + object.height)
6       return object
7   }
8 }
9

10 const myObjects = [
11   {x: 10, y: 20, width: 30, height: 30},
12   {x: -40, y: 20, width: 30, height: 30},
13   {x: 0, y: 0, width: 10, height: 5}
14 ]
15

16 console.log(detectCollision(myObjects, {x: 4, y: 2}))
```

If you run it, you will get `{x: 0, y: 0, width: 10, height: 5}` in the console, because this is the object, the last one, that includes the point `{x: 4, y: 2}`.

Can you use higher order functions in order to make this code cleaner? You might want to use the method `find` Array method, which returns the first object that satisfies (returns `true`) the function given as argument to `find`.

## Task 13

Practice the spread operator by simplifying the following three functions. The `replace`, replaces parts of an array (`array`) using elements of another `elements`. The function `copyReplace` does the same but does not alter the original array. Last, the function `recordBirds` push an object into the array `birdsSeen`. The object has two properties: a) `time` and b) an array of birds.

Note that the changes that you have to do are only inside the bodies of the functions, and possibly at the arguments that they take and they are being called with.

Finally, you may find it good to read an extensive article on the `spread` and `rest` operators [here](#)

```
1 const replace = (array, from, to, elements) => {
2   array.splice.apply(array, [from, to - from].concat(elements));
3 }
4
5 let testArray = [1, 2, 100, 100, 6];
6 replace(testArray, 2, 4, [3, 4, 5]);
7
8 console.log(testArray); // prints [1, 2, 3, 4, 5, 6]
9
10 const copyReplace = (array, from, to, elements) => {
11   return array.slice(0, from).concat(elements).concat(array.slice(to));
12 }
13
14 console.log(copyReplace([1, 2, 100, 200, 6], 2, 4, [3, 4, 5])); // prints [1, 2, \
15 3, 4, 5, 6]
16
17 let birdsSeen = []
18 const recordBirds = (time) => {
19   birdsSeen.push({time: time, birds: Array.prototype.slice.call(arguments, 1)});
20 }
21
22 recordBirds(new Date, "sparrow", "robin", "pterodactyl");
23 console.log(birdsSeen.length); // prints 1
24 console.log(birdsSeen[0].time); // prints the current date
25 console.log(birdsSeen[0].birds); // prints ["sparrow", "robin", "pterodactyl"]
```

## Task 14

This is an exersice for template strings, strings with interpolated values. Here is the code:

```
1 const teamName = "tooling"
2 const people = [{name: "Jennie", role: "senior"},
3                 {name: "Ronald", role: "junior"},
4                 {name: "Martin", role: "senior"},
5                 {name: "Anneli", role: "junior"}]
6
7 let role = "senior";
8 let message = YOUR_CODE_HERE
9
10 console.log(message);
```

Fill in the code so that it prints the following multiline message. Make sure that the string is built using the data and hard coded values.

- 1 There are 4 people on the tooling team.
- 2 Their names are Jennie, Ronald, Martin, Anneli.
- 3 2 of them have a senior role.

## Task 15

You need to fill in the following JavaScript function, `connectedValue`. This sums up all the values of the nodes of a graph that a given node is connected to, directly or indirectly. You need to find a way to traverse the graph starting from the given node. Make sure that you don't end doing infinite circles. You need to use a Set to keep track of the nodes that you visit.

This is an example graph:

```
1 80 => 90 => 78 => 90
2      => 30 => 20
3      => 50 => 78
4      => 80
5 100 => 90
6      => 80
```

For that graph, starting from node 80 the result should be 348 because we add the nodes: 80, 90, 30, 50, 78 and 20.

```
1 // Generate a random graph
2 const graph = [];
3 const numberOfNodes = 5;
4 const numberOfEdges = 20;
5 const maxNodeValue = 1000;
6 for (let i = 0; i < numberOfNodes; i++) {
7   const newNode = {value: Math.floor(Math.random() * (maxNodeValue + 1)), edges:\
8   []};
9   graph.push(newNode);
10 }
11 for (let i = 0; i < numberOfEdges; i++) {
12   let from = graph[Math.floor(Math.random() * graph.length)];
13   let to = graph[Math.floor(Math.random() * graph.length)];
14   if (from.edges.indexOf(to) != -1)
15     continue
16   from.edges.push(to);
17 }
18
19 const connectedValue = (node) => {
20   YOUR_CODE_HERE
21 }
22
23 console.log(connectedValue(graph[0]));
```

## Task 16

Implement your version of the `Map` class. Call it `MyMap`. Here is the placeholder code:

```
1 class MyMap {
2
3   // YOUR CODE HERE
4
5 }
6
7 const customers = new MyMap();
8
9 class Customer {
10   constructor(firstName, lastName, age) {
11     this.firstName = firstName;
12     this.lastName = lastName;
13     this.age = age;
14   }
15   get fullName() {
16     return `${this.firstName} ${this.lastName}`;
17   }
18 }
```

```
19  
20 let customer = new Customer('John', 'Smith', 36);  
21 customers.set(customer.fullName, customer);  
22  
23 customer = new Customer('Mary', 'Poppins', 28);  
24 customers.set(customer.fullName, customer);  
25  
26 console.log(customers.get('John Smith').age); // prints 36  
27 console.log(customers.size); // prints 2  
28  
29 customers.delete('John Smith')  
30 console.log(customers.get('John Smith')) // prints undefined  
31  
32 customer = customers.get('Mary Poppins');  
33 console.log(customer.fullName, customer.age); // prints 'Mary Poppins', 28  
34  
35 customers.clear();  
36 console.log(customers.get('Mary Poppins')); // prints undefined
```

Your MyMap class needs to implement the following methods so that the above program runs successfully:

1. #set(key, value)
2. #get(key)
3. #delete(key)
4. #get size()
5. #clear()

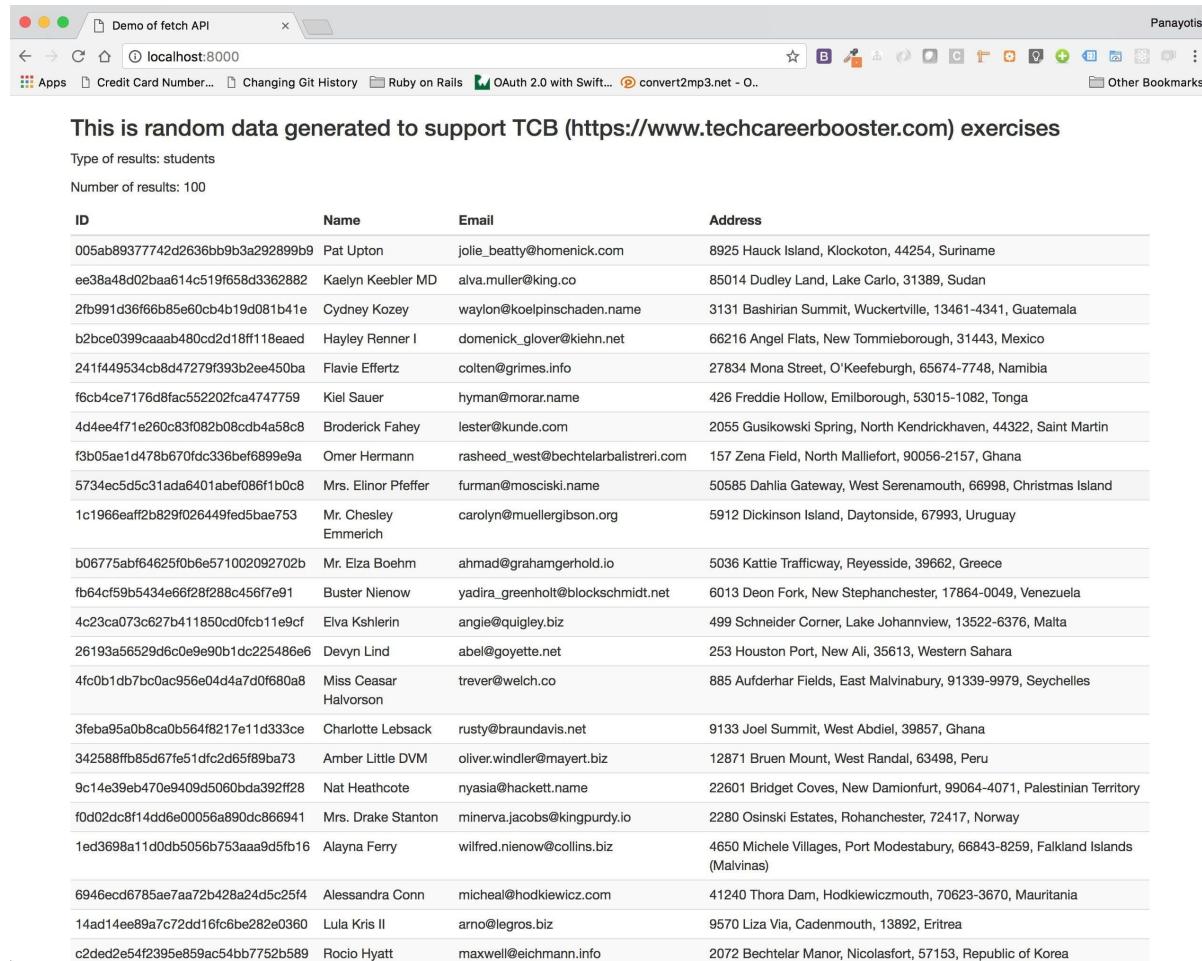
These methods should do what their corresponding methods on actual Map do.

Don't use the Map class internally. Use other structure. We don't care for performance. We just want you to implement the methods.

## Task 17

You need to use the `fetch` API in order to fetch data from [TCB Students Fake Data](#)

Implement a Web page that when loaded will display the random data as in the following screenshot:



The screenshot shows a web browser window titled "Demo of fetch API" at "localhost:8000". The page displays a table of 100 student records. The columns are labeled "ID", "Name", "Email", and "Address". Each row contains a unique ID, a name, an email address, and a specific address. The browser's toolbar and address bar are visible at the top.

ID	Name	Email	Address
005ab8937742d2636bb9b3a292899b9	Pat Upton	jolie_beatty@homenick.com	8925 Hauck Island, Klockoton, 44254, Suriname
ee38a48d02baa614c519f658d3362882	Kaelyn Keebler MD	alva.muller@king.co	85014 Dudley Land, Lake Carlo, 31389, Sudan
2fb991d36f6b85e60cb4b19d081b41e	Cydney Kozy	waylon@koelpinschaden.name	3131 Bashirian Summit, Wuckertville, 13461-4341, Guatemala
b2bce0399caaab480cd2d18f118eaed	Hayley Renner I	domenick_glover@kiehn.net	66216 Angel Flats, New Tommieborough, 31443, Mexico
241f449534c8d8f7279f393b2ee450ba	Flavie Effertz	colten@grimes.info	27834 Mona Street, O'Keefeburgh, 65674-7748, Namibia
f6cb4ce7176d8fac552202fca4747759	Kiel Sauer	hyman@morar.name	426 Freddie Hollow, Emilborough, 53015-1082, Tonga
4d4ee4f71e260c83f082b08cdb4a58c8	Broderick Fahey	lester@kunde.com	2055 Gusikowski Spring, North Kendrickhaven, 44322, Saint Martin
f3b05ae1d478b670fdcc336bef6899e9a	Omer Hermann	rasheed_west@bechtelbalistreri.com	157 Zena Field, North Malliefort, 90056-2157, Ghana
5734ec5d5c31ada6401abef086f1b0c8	Mrs. Elinor Pfeffer	furman@mosciski.name	50585 Dahlia Gateway, West Serenamouth, 66998, Christmas Island
1c1966eaff2b829f026449fed5bae753	Mr. Chesley Emmerich	carolyn@muellergibson.org	5912 Dickinson Island, Daytonside, 67993, Uruguay
b06775abf64625f0b6e571002092702b	Mr. Elza Boehm	ahmad@grahamgerhold.io	5036 Kattie Trafficway, Reyesside, 39662, Greece
fb64cf59b5434e66f28f288c456f7e91	Buster Nienow	yadira_greenholt@blockschmidt.net	6013 Deon Fork, New Stephanchester, 17864-0049, Venezuela
4c23ca073c627b411850cd0fcb11e9cf	Elva Kshlerin	angie@quigley.biz	499 Schneider Corner, Lake Johannview, 13522-6376, Malta
26193a5652d6c0e9e90b1dc225486e6	Devyn Lind	abel@goyette.net	253 Houston Port, New Ali, 35613, Western Sahara
4fc0b1db7bc0ac956e04d4a7d0f680a8	Miss Ceasar Halvorson	trever@welch.co	885 Aufderhar Fields, East Malvinabury, 91339-9979, Seychelles
3feb95a0b8ca0b564f8217e11d333ce	Charlotte Lebsack	rusty@braundavis.net	9133 Joel Summit, West Abdiel, 39857, Ghana
342588ff85d67fe51dfc2d65f89ba73	Amber Little DVM	oliver.windler@mayerit.biz	12871 Bruen Mount, West Randal, 63498, Peru
9c14e39eb470e9409d5060bda392ff28	Nat Heathcote	nyasia@hackett.name	22601 Bridget Coves, New Damionfurt, 99064-4071, Palestinian Territory
f0d02dc8f14dd6e00056a890dc866941	Mrs. Drake Stanton	minerva.jacobs@kingpurdy.io	2280 Osinski Estates, Rohanchester, 72417, Norway
1ed3698a11d0db5056b753aaa9d5fb16	Alanya Ferry	wilfred.nienow@collins.biz	4650 Michele Villages, Port Modestabury, 66843-8259, Falkland Islands (Malvinas)
6946ecc6785ae7aa72b428a24d5c25f4	Alessandra Conn	michael@hodkiewicz.com	41240 Thora Dam, Hodkiewiczmouth, 70623-3670, Mauritania
14ad14ee89a7c72dd16fc6be28e0360	Lula Kris II	arno@legros.biz	9570 Liza Via, Cadenmouth, 13892, Eritrea
c2ded2e54f2395e859ac54bb7752b589	Rocio Hyatt	maxwell@eichmann.info	2072 Bechtelar Manor, Nicolasfort, 57153, Republic of Korea

### Demo Fetch API

Your page should be fetching the demo random data [from this end point](#).

Note that all the data displayed on the page, including the top description, the type and number of results, should be dynamically filled in using the information returned by the end-point.

We have used Twitter Bootstrap to style the page. You may want to use that too.

## Task 18

Look at the following code. It is using the property `_content` relying on the `_` prefix to mark it as private.

```
1 class Queue {  
2     constructor() {  
3         this._content = []  
4     }  
5     put(elt) {  
6         return this._content.push(elt)  
7     }  
8     take() {  
9         return this._content.shift()  
10    }  
11 }  
12  
13 let q = new Queue  
14 q.put(1)  
15 q.put(2)  
16 console.log(q.take())  
17 console.log(q.take())
```

Do you know how you can change the code to use a symbol instead of an underscored-named property? You may want to read this too in order to better understand how privacy can be achieved in JavaScript.

## Task 19

Make a class of EvenNumbers. It should be used like this:

```
1 const evenNumbers = new EvenNumbers([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);  
2  
3 for(let num of evenNumbers) {  
4     console.log(num);  
5 }
```

The above prints the numbers 2 up to 10.

You need to make the class iterable.