



New York, U.S.A.

Tony de Araújo

JavaScript In Plain Language

A Self-Study Method



0,1,2,3,4...

Result: 0,1,"yellow", 2, 3, 4...

First item position to be acted upon

Number of items to delete if any

Items to be added



*"empty your cup,
start at the beginning"*



Figure 20.

Let's try a few samples to illustrate the concept.

Deleting items with `splice()`:

JavaScript in Plain Language

A Self-Study Method

JSON and AngularJS Prep

by

Tony de *Araujo*
Technical Instructor

New Jersey – USA

Fully updated on October 2015

ALSO BY TONY DE *ARAU* JO

[JavaScript Objects Functions and Arrays Explained](#)

[Small Projects to Learn JavaScript - Drawing Six 'Lucky' Numbers](#)

[AngularJS Supplement: Easy Projects on Filters and Directives](#)

copyright © Tony de Araujo
All Rights Reserved

To my family and friends
all my readers, Amazon
and Amazon supporting staff...
Thank you so much!

This eBook works best when using the free **Kindle Reading App** due to all the resources given and all the linked exercises.

You can download an Amazon Kindle Reading App from the following link:

[amazon.com/gp/digital/fiona/kcp-landing-page](https://www.amazon.com/gp/digital/fiona/kcp-landing-page)

Table of Contents

[Introduction](#)

[What you should know before reading this material](#)

[What is JavaScript](#)

[PART I: AT THE GATES OF ENLIGHTENMENT](#)

[1.1 The browser interface](#)

[1.2 Material and tools needed](#)

[1.3 Declaring a variable – an alias name for a memory location](#)

[About var](#)

[About the variable name](#)

[About the semicolon ;](#)

[Adding data to variables](#)

[Changing the value in the variable](#)

[Adding words to a variable instead of numbers](#)

[Let's talk about *strings*](#)

[Which quotes to use: *single* or *double*?](#)

[Assigning variables to other variables](#)

[1.4 Lab work 1](#)

[Opening the Console](#)

[Declaring some variables](#)

[Inspecting the data type of a variable](#)

[What is *typeof*?](#)

[Why it “undefined” and what is undefined?](#)

[1.5 Manipulating variable data](#)

[Methods and Properties](#)

[The DOT operator](#)

[The Bracket notation](#)

[An example of a method](#)

[What's with the parentheses \(\) ?](#)

[1.6 Lab work 2](#)

Declaring variables, assigning values, using basic properties and methods

1.7 Adding some more JavaScript operators

+, -, +=, -=, /, /=, %, *, *=

The + plus operator

Using the + plus operator to add two numbers

Using the += assignment operator

The ++ (double plus) incrementing operator

The -, — and -= subtraction operators

The Division / or /= Operators

The Modulus % Operator

The Multiplication Operator *

1.8 Lab work 3

Part one: working with +, +=, ++ and —

Part two: Working with *, /, *=, /= and %

1.9 Comparison operators and booleans

===, !==, <, >, <=, >= true, false

What is a Boolean?

1.10 Lab work 4

Practicing with ===, !==, <, >, <=, >= true, false.

1.11 Introducing console.log, alert, prompt, confirm

The console.log

alert()

confirm() and prompt()

1.12 Logical Operators

And: &&

Or: ||

Not: !

1.13 Lab work 5

Getting feedback from a user

PART II: ENTERING THE SECOND REALM

2.1 Adding more syntax

[Code blocks - Using an editor](#)

[Commenting code](#)

[2.2 Conditional Branching](#)

[if, else conditionals](#)

[Reviewing if\(\), else](#)

[What about else?](#)

[Adding a second condition in the middle of if\(\), else](#)

[How to physically write the braces in a conditional statement](#)

[When to use an else if\(\)](#)

[Is Programming an Art or a Science?](#)

[2.3 The anatomy of a function](#)

[Preliminary things to know about a function](#)

[Scripting a function](#)

[What is a function call?](#)

[What is the purpose of return?](#)

[What happens to the data when a function returns?](#)

[2.4 Lab work 6](#)

[Create a program to convert Fahrenheit to Celsius](#)

[Create a program to convert Celsius to Fahrenheit](#)

[In JavaScript a function always returns something](#)

[A program to convert Fahrenheit to/from Celsius](#)

[2.5 Filling out a form to request a loop](#)

[What no one tells you about loops](#)

[2.6 The for loop](#)

[2.7 Looping over a string of characters](#)

[2.8 Lab work 7](#)

[Searching for the position of a string character](#)

[Counting numbers and declaring their odd/even quality](#)

[Counting apples](#)

[2.9 Array lists](#)

[Strings versus arrays](#)

[Array syntax](#)

[Using push\(\) and unshift\(\) to insert elements into an array](#)

[Printing array items with a for loop](#)

[2.10 Lab work 8](#)

[Creating a method to add items into an array](#)

[Create a generic method to print out the contents of an array](#)

[Extra bonus: Create a generic method to add items into an array](#)

[2.11 Other useful arrays methods](#)

[sort\(\)](#)

[join\(\)](#)

[indexOf\(\)](#)

[pop\(\) and shift\(\)](#)

[slice\(\)](#)

[splice\(\)](#)

[Splice lab work](#)

[2.12 Lab work 9](#)

[2.13 Lab work 10](#)

[Building a mechanism to fetch a variable from an array](#)

[Adding French to the array](#)

[Printing out the array contents](#)

[2.14 Unordered lists](#)

[Hashes, objects, associative arrays, unordered lists](#)

[Creating an object](#)

[Why should we wrap key labels in double quotes?](#)

[Assigning new key-value pairs to objects](#)

[Using dot notation with objects](#)

[Spanning an object declaration across multiple lines](#)

[What is a property?](#)

[What is a method?](#)

[Looping through an object](#)

[Introducing the for in loop](#)

2.15 Lab work 11 - redo lab work 10

Transforming the for in loop into an object method

PART III: STRENGTHENING THE WARRIOR'S ARSENAL

3.1 Taking an inventory of what we've learned

3.2 More about functions

What are expressions and statements?

Assigning functions to variables

What is an anonymous function?

Using variables as pointers

The balloon concept

Function parameters and arguments

The object arguments

3.3 Manipulating variables with functions

Passing data by value

Passing data by reference

Functions are *closed- in* structures

"*This*" as an object placeholder

3.4 Lab work 12

Part A: Expressions, statements, arguments

Part B: Passing data by value and by reference

Part C: Find and replace a word in a string

Part D: Using a function to save and recall our code

Part E: Making the search and replace more portable

3.5 More branching techniques

The switch

3.6 Lab work 13

13a: Switch - using Boolean matches

13b: Switch - using literal matches

13c: A repeating switch routine

13d: Using a while loop to repeat the script

PART IV: ENTERING THE THIRD REALM

[4.1 About this section](#)

[4.2 An introduction to JSON](#)

[Use jsonlint, a tool to check JSON validation](#)

[Assigning a JSON object to a variable](#)

[Editing the JSON object](#)

[JSON library methods](#)

[JSON.stringify](#)

[Using a second argument as a filter on stringify](#)

[Using a third argument on stringify\(\) for controlling white space](#)

[JSON.parse](#)

[In summary](#)

[Document-oriented databases for JSON data](#)

[PART V: DEFEATING THE DRAGON](#)

[5.3 First solid steps into AngularJS](#)

[Intro](#)

[Directives](#)

[What is a library?](#)

[What is a directive?](#)

[What does ng-app mean?](#)

[No more DOM manipulation tasks](#)

[What is an expression?](#)

[ng-init](#)

[ng-bind](#)

[ng-cloak: Avoiding display flickering](#)

[5.4 Lab work 14](#)

[Part A: Initializing an object and outputting its contents in a paragraph](#)

[Part B: Initializing variables and outputting a calculation](#)

[5.5 Going a step further into modularity](#)

[Assigning a module name to ng-app](#)

[Configuring a Module interface](#)

[5.6 Preparing to separate data from presentation](#)

[Creating a controller](#)

[\\$scope](#)

[View](#)

[Controller](#)

[Your turn: Add a controller to your exercise](#)

[Create a program to convert Fahrenheit to Celsius](#)

[Placing the controller on a separate file](#)

[The ng-model](#)

[Lab work, creating the conversion script](#)

[More Project Ideas](#)

[Directives summary and resources](#)

[In conclusion](#)

[Can you do this for me?](#)

[Errata, Contact, Updates](#)

Introduction

This book is for everyone who wants to learn basic programming using the JavaScript language without taking a formal course. It may also serve as a classroom supplement for practicing and review.

- The intended audience is beginning, intermediate, or an advanced reader looking to restudy the subject.

This book contains lots of exercises with answers and explanations. Each section is one page or less long and each topic is followed by lab work with links to raw files. There is also a *forum board* to facilitate further reading and practice. Since all scripts are short, these links are there just to clear any possible doubt you may have about your own work.

I suggest spending at least 30 minutes per day reading and practicing each chapter. Do not rush the material because the topics covered are the foundation to master the language. Take time to absorb each concept. Do not go into a new subject without taking a conscious break from the computer to do some physical activity like for example a minor stretch.

Read each subtopic and do the exercises. Follow the book from beginning to end; then later you may want to use this material as a comprehensive reference to refresh your mind or for practicing purposes. For further practicing refer to the dedicated forum platform presented later in the book.

If you study every day you will complete the course in about two weeks.

If you just read it as a review, you may finish it on a weekend's time.

The book takes advantage of your browser's JavaScript Console. I recommend using the Console from *Google Chrome* and I will explain how to access it and use it for our purposes. *Opera* and *Epic* will work the same way. Of course, you can utilize any other test method you wish as long as you know how to access it. The new Microsoft Edge browser also has a JavaScript Console accessed by the shortcut F12.

Besides recommending *Google Chrome*, I will also recommend the usage of *jsbeautifier* (a very useful online editor), as well as my own website forum where you will find all the linked exercises and further reading and practicing materials.

Don't worry about it for now; there will be plenty of reference links posted at the appropriate time.

What you should know before reading this material

The only prerequisite for reading and practicing the projects in this book is that you are familiar with basic computer skills such as *copy/paste* and *editing* a line of text.

For a more complete experience you need access to the *Internet*.

You can always review the material offline but you will not be able to access any of my sample files if you don't go online.

In order to fully understand the last section of the book which introduces **AngularJS** in plain words, you will need to know basic HTML syntax like for example, understanding how a basic HTML page is created, the HTML tag, the BODY tag, how to create a paragraph or a DIV – nothing too complex.

The book will give you the foundation to become comfortable with JavaScript so that you can advance firmly regardless of which area of the language you plan to specialize on.

Basic math is a plus, but even if you think math is not your call, all math examples are well explained in the book. No one should have any problem with them.

The book will prepare you to understand modern JavaScript. This includes a firm introduction to **JSON** and **AngularJS** in plain words, which will open doors and break down barriers for you to get into those very hot technologies.

The Spiral approach for code learning

This eBook utilizes the idea of a spiral approach for code learning as outlined below:

- 1- The history of development is preserved.
- 2- Simple solutions are provided first and then replaced by more advanced, robust solutions.
- 3- Only the concepts used in a spiral are explained, with links for further reading material.

Each project conserves its incremental history as a snapshot of something useful to review, like a spiral. The spiral approach is a technique often used in teaching where first the basic facts of a subject are learned, without worrying for detail. Then as learning progresses, more and more details are introduced while at the same time they are related to the basics which are reemphasized many times to help the internalization of concepts.

No memorization is needed.

- The book must be read in sequence for a more effective, powerful and meaningful experience.
- The exercises should be repeated not just right after finishing the current lab work, but also after finishing the whole section or reading the whole book.
- Practice like you would be practicing guitar: every day and with intent. Think of JavaScript exercises as scales to be played.
- After finishing an exercise, always think on how you would apply the concepts covered for a different recipe. Think like a master chef. Programming is like cooking.
- Programming is also like plumbing. We create interfaces by facilitating connections.
- Finally, programming is part science part art. It brings together the best of both worlds.

What is JavaScript

JavaScript is a programming language designed to interact with the computer browser. There is currently no other language with the same browser compatibility as JavaScript. Most of those languages trying to replace JavaScript actually [decompile](#) their code into JavaScript.

What is the difference between JavaScript and HTML?

HTML is a presentation markup language. It creates objects (container boxes) that get displayed on the page. It does not make decisions based on user interaction. HTML paints the computer screen.

JavaScript extends HTML in the sense that it makes the page dynamic by interacting with the user based on certain conditions and outcomes.

In order to create outcomes and make decisions we need to learn JavaScript syntax and that's what this book is about.

In later chapters we will learn how to create modern web interaction using one of the latest implementation techniques called AngularJS.

The book will not make you an expert in AngularJS. It will however give you a very solid foundation to empower you and propel you forward into more advanced concepts.

This book follows a hands-on approach. Please be prepared for an extensive workout.

PART I: AT THE GATES OF ENLIGHTENMENT

“Empty your cup; return to the beginning”.

1.1 The browser interface

Every time we go online we get to use an [interface](#) to communicate with the internet. This particular interface is known as “[the browser](#)” and its purpose is to translate the incoming data from the outer world into our computer screen. The most popular browsers are Internet Explorer (known as IE and now replaced by Microsoft Edge), Google Chrome, Firefox, Opera and Safari but there are hundreds of [other browsers](#).

Browsers place *text and images* on the screen based on instructions given by the programmer in a markup language known as [HTML](#). Then this text and images are rearranged or beautified by a styling language called [CSS](#). Finally, the website dynamically interacts with the user via [JavaScript](#) which is the subject of this book.

JavaScript is a computer language designed to interact with and recognized by all browsers. This means that we can create useful programs to run in a browser by writing instructions in JavaScript.

Other programming languages don't have this capability and they need to run on special platforms acting as further interfaces to the browser and they ask for JavaScript to assist them in this interaction. This is the reason why JavaScript has become the most popular programming language on the [internet](#). Computers use it, websites use it, other software such as for example Photoshop, Word, PDF use it, and last but not least, computer servers are starting to use it on their own backend processing which is great news for the future of JavaScript.

JavaScript can be an easy language when learned the right way, but it can also be a difficult language to learn because it is very dynamic and rules change based on conditions on the ground.

No one can promise to teach you JavaScript since learning is a personal experience. But I can promise you this: read the book in sequence, think about each concept without memorizing it, do all the exercises, and you will be a much better programmer by the time you reach the end of this very short book.

As an author, I may be a programmer or a technical writer, but above all, I am a technical Instructor. My writing experience comes from teaching, coaching, and debugging code from thousands of students. When I write about these concepts I always think about the struggling I've experienced when I was learning this material, and the struggle I see on other students when they ask for help. As I write, those problems come to the surface and I try to address them ahead of time in order to prevent you from falling into common traps.

1.2 Material and tools needed

All the exercises on this book will be tested using the JavaScript Console from the Google Chrome's browser. Occasionally I also use **Opera** which works similarly. Please feel free to use any test console you so desire since every browser has one. Firefox is also a good alternative but I find the other two options cleaner and easier for beginners to code.

If you don't have one of those browsers on your computer you can download it from the links given below. I recommend using both of them. Experimenting with different browsers is part of being web-savvy and a must for web developers:

Google [Chrome](#) | [Opera](#)

- 1- To access the console, load you Chrome browser or Opera, and then press the following key sequence:

For Windows systems: **CTRL SHIFT j**

For Mac: **CMD OPT j**

This will bring up the console.

- 2- If this is your first time using the console, you can detach the console from the browser by clicking on the ***small square*** shown at the bottom left of the console (in the latest browser this button seems to be on the top right). Detaching the Console will give you more room on the screen since you will be able to minimize the browser while keeping the console up.
- 3- Make sure the tab on the top of the console is set to *console* and not to any of the other available options.
- 4- You may see a few **error messages** on the Console when you first open it. This is normal. Just clear the screen by right-clicking and selecting "*Clear console*".

Let's now read a few more pages and code mentally. We will do some *hands on* work in a few minutes when we get to the Lab section.

1.3 Declaring a variable – an alias name for a memory location

Preliminary information:

Browsers use computer memory. The more memory a computer has, the faster the browser works. Since *working memory* is limited, programs need to constantly rearrange data in memory by deleting what expires and repositioning what is still in use.

When we program in JavaScript we store data in memory. This data is short lived since once we close the browser or end the program, the data is wiped out.

To temporarily store data in memory we need to label the memory address location with a word that makes sense to humans, so that we can program or manipulate that data. We can't just tell the browser to store the string of characters "hello world!" in memory by giving it the address of [0x7ffffff](#). Those days are long gone as it would be too complex and unpractical as well as ever changing since data is shifted around from location to location due to automatic memory rearrangements. So to address this problem, we use names as *alias* for assigned locations and allow the computer to manage the real physical address of the data as it sees fit.

Those memory location *aliases* are called **variables**. The reason they are called variables is twofold: the address of the data can be changed without notice, and the data contained in each variable can also be changed at any time. What remains constant is the variable name, which is the only link humans have to this piece of memory real estate.

As for the change of a value in a variable, that may be either done manually by the programmer further down the script instructions, or automatically by JavaScript itself in a dynamic reassignment of values.

- **Know this:** To create a memory variable we have to *declare* it (declaring a variable is like introducing the variable to JavaScript). Example:

```
var phoneBill;
```

(Just read along. You will have a chance to try it yourself when we get to the Lab Work section at the end of the chapter. All chapters are purposely short. This is the time to read and understand conceptually. Then you will test yourself during lab. All lab exercises are followed by the correct answers so that you can double check your work).

About var

Think of *var* as the command used to create a variable (this helps you remember to use *var* when you introduce a variable).

Another thing that *var* does is to ground the variable to the location where it is being introduced, sort of like making it a local variable (more about that later).

About the variable name

After writing *var* we provide the variable **name** which in our case is *phoneBill*. The name does not matter, it could have been called *x* since JavaScript does not care about the ‘human meaning’ we choose to give it.

However, JavaScript cares about the choice of characters we use. Make sure the first character is a lowercase letter of the alphabet (the lowercase is not mandatory, just good practice as you will see later). The variable cannot start with a number or a symbol. Subsequent letters can be numbers, but symbols are not allowed, ever, except for the underscore `_`.

- In summary: always start your variable names with a lower case letter. Forget about the underscore as the first character because it is usually used in very special situations and therefore should not be considered for normal usage.

My own variable *phoneBill* has a capital B in the middle. This is a common style of writing variable names. JavaScript doesn’t care about styles, but it does make it easier for the human eye when it comes to inspect the script.

Another popular way of writing names is with an underscore separating the ‘human meaning’ of each word in the variable name. The reason for the underscore is because spaces or dashes are not permitted in variable names: *phone_bill*

- **Remember this:** The ‘meaning of a word’ is only for human consumption. If instead of introducing *phoneBill* we introduced just *x*, JavaScript would have accepted it without complaining. When we program, we do it both for the computer to understand and also for a human to understand so that he/she can troubleshoot or update the program later. Choose your own style of writing variables and stick to it. *Consistency is important.* However, become familiar with all common styles because when you work in a team you need to adjust to an agreed common style for the whole team. Practice flexibility.

About the semicolon ;

The semicolon at the end of the *statement* tells JavaScript that you are done declaring the variable:

```
var phoneBill;
```

The semicolon acts as a command **terminator**, or a separator between two different and independent commands. In other words, we are instructing JavaScript to stop and save that command or expression as an independent instruction from the next upcoming instruction.

Going back to the variable name,

- **Keep this in mind:** JavaScript is case sensitive. *A* and *a* have independent meanings in JavaScript. *phoneBill* and *phonebill* are not the same word. At this moment *phoneBill* is an existing declared variable, and *phonebill* is nothing, JavaScript will throw an error if you write *phonebill* by mistake instead of *phoneBill* since you have declared *phoneBill* as the variable, and not *phonebill*.

Of course we could also have two independent variables, one named *phoneBill* and another named *phonebill* and that is the point, for JavaScript they are two different words.

Adding data to variables

Once a variable is declared, a memory location is reserved for that variable and then we can assign the variable to some data.

Isn't it easier to just remember *phoneBill* rather than making a written map of reserved memory locations?

To assign data into a variable we use the = operator:

```
phoneBill = 200;
```

Do not use \$ before 200. That would trigger an error. You may use \$ later when you output the data if you so wish, but not when you enter your data into the computer memory. All JavaScript wants is the number 200 in its pure form.

I have just saved the number 200 in a memory location listed as *phoneBill*. Notice the semicolon at the end. It terminates my *statement*.

We could have declared and also included the data in just one-*step*:

```
var utilitiesBill = 400;
```

Sometimes we just want to tell JavaScript to reserve a location under a certain name. Other times we reserve the location and fill it with data all at once. Don't worry about the size of memory needed. Contrary to other languages, JavaScript is dynamic: it adjusts the memory automatically.

Changing the value in the variable

What if we want to change the value of our `phoneBill` from 200 to 150?

To change the value of a variable all we have to do is to *reassign* it (no declaration needed):

```
phoneBill = 150;
```

The 200 gets deleted and the computer memory at location *phoneBill* now holds the value of 150.

- **Remember this: Never use *var* twice** on the same variable name. If you use *var* as a prefix when you rewrite an existing variable, you wipe out the old variable since you have just re-declared the variable again. This may be something you want to do on purpose, but for the most part, this is not a common practice. I have seen this done many times as a mistake by new programmers.

Adding words to a variable instead of numbers

So far we have been assigning data of *type number* to our variables. We can actually assign anything to a variable.

Since a variable is just a memory location, we can assign it a string of characters also known as a value of type *string*, or even other types of data such as *arrays*, *functions*, *objects*... more about those later.

Let's talk about *strings*

A string of characters can be assigned to a variable but we need to let JavaScript know it is a string. The way we let it know, is by wrapping our string of characters in *quotes*:

```
var myName = "Tony";
```

Any word not wrapped in quotes is considered to be a reserved term stored in the JavaScript library.

- If I just write `var myName = Tony`; (with no quotes for the string 'Tony'), JavaScript will check its library in the computer's memory looking for a command or a variable named *Tony*. Since Tony does not exist as a reserved term, JavaScript will throw an error on the screen. Any *word data* needs to be wrapped in quotes.

After we declare the variable myName, the word myName then becomes a reserved term until it gets wiped out from memory or until the program is over. No quotes for variable names.

- **Remember this:** Data of type *string* needs to be wrapped in quotes as opposed to declared variables names which are **never** written in quotes.

When I say "remember this" it doesn't mean to memorize it. It only means to focus a few seconds of extra attention to the statement that is about to follow.

Which quotes to use: *single* or *double*?

In JavaScript, a pair of single or double quotes work the same way. Just make sure the closing quote matches the style of the opening quote. Some other languages act differently in single or double quotes; not JavaScript.

Below you will find another variable declaration and value assignment all at once. You will see how combining quotes can be useful. I start with a single quote, followed by a double quote. However JavaScript ignores the double quote because it knows the closing quote must be single in order to match the opening quote:

```
var quoteOfTheDay = `"living is not essential; creating is essential" - Fernando Pessoa`;
```

- **Remember this:** Never wrap a variable name in quotes. If you do that, you convert the name into a string. I mean, `quoteOfTheDay` and `"quoteOfTheDay"` don't have the same meaning for JavaScript. The first one is a declared variable stored in the JavaScript temporary library. The second one is just a plain string of characters. JavaScript will not map it to the memory location assigned to *quoteOfTheDay*.

Assigning variables to other variables

We have seen already how to reassign a variable to a new value:

```
phoneBill = 600;
```

Now, phoneBill is no longer 150. It holds the value of 600.

What about assigning a variable to another variable?

It works the same way as assigning a regular value:

```
phoneBill = quoteOfTheDay;
```

Note: No quotes are being used because both words are previously declared variables. The real value, which is a string value, is being passed inside of the variable quoteOfTheDay.

- **Remember this rule:**

Assignments work from *right to left*. What is on the right gets assigned to the left operand.

The value of *phoneBill* has now been changed from the numeric 600 to the string “*living is not essential; creating is essential*” - *Fernando Pessoa*.

Basically, JavaScript **copied** the string value from *quoteOfTheDay* to *phoneBill*. We now have two strings of the same value in two different memory locations. This copying is not always true as we will see later. Copying only happens with *primitive* data types, such as numbers and strings. When we reassign complex data such as *arrays* and *functions*, also known as *reference* types, we do not duplicate the data; we just point those two variables to the same data. But this is an advanced concept. For now, just think of duplicating variable *values* when we reassign a variable containing simple data to another variable.

We will see more about *primitive* and *reference* types later in the book. Sometimes I throw in terms without an explanation. The idea is to get you used to the term as common language when you read along and without memorizing it. A former introduction will happen at the appropriate time. Occasionally you will see a link to sources such as Wikipedia. The link is there in case you want to explore the topic further, but no external link is essential in order to understand the material we are covering. Wikipedia is my favorite source of general information for two reasons: It gets updated quickly and it does not have annoying popup commercial messages.

1.4 Lab work 1

Ready for some lab work?

Opening the Console

Almost all modern browsers have a *test Console* built into the browser to assist us in debugging or experimenting with JavaScript. All Consoles provide basically the same options once we have them open. The biggest differences are how we access the Console and what the Console looks like on the screen.

Select one of the following browsers. Download it if you don't have it installed on your computer. The reason to use one of these browsers is so that my instructions may coincide with your actions as we code along using the JavaScript Console that comes with it. Please feel free to use any other Console if you so desire, but the way to access it may be different.

[Google Chrome](#).

[Opera](#).

Open one of those two browsers and then press the following key sequence to open the Console:

- 1- If you are using Windows or Linux:

CTRL SHIFT j

If on a MAC:

CMD SHIFT j or ***ALT-CMD-j***

Most likely a window will open at the bottom of your screen.

- 2- Make sure the tab *Console* is selected.

If you get any *error* messages, just clear the screen by right clicking on the white area and selecting **“Clear Console”**. These errors will happen almost every time you open the console because the Console is inspecting the current opened document, the one you see on screen before you access the Console, and things may not be 100% as expected by the Console, hence the error warnings. Just clear the message so that we gain more window space.

- 3- If you want (recommended), you can *detach* the Console from the browser and minimize the browser so that it gets out of your way. After you detach the Console for the first time, it will always stay detached unless you attach it again by toggling the same button.

To detach the Console look for a double square icon on the tab menu of the Console, and click on it. Now you should have two different windows. Minimize the browser to get more room on your Console screen.

Declaring some variables

Let's start coding...

- 1- *Declare* variables *x*, *y* and *z*

(declaration only, no assignments of values at this moment)

and press ENTER after each semicolon in order to activate the expression:

It should look like this:

```
var x;
```

```
var y;
```

```
var z;
```

You could also just use one *var* and separate the variable names with a *comma*. That might save you some time if you are simultaneously declaring a bunch of variables with no values given. Don't forget the *semicolon* at the end:

```
var x, y, z;
```

- 2- Now, declare another variable but assign a value at the same time.

I'm using variable *a* with the value of 347:

```
var a = 347;
```

- 3- Declare a variable *b* containing your first name (remember, your name is a *string* of characters):

```
var b = "Tony";
```

- **NOTE:** You can read the value of each variable on the Console by just typing the variable name, followed by the semicolon, and pressing ENTER. Example:

```
b;
```

It should display "*Tony*";

Typing the variable by itself in order to get a value displayed is only possible because we are using a test Console. In real life we would have to use a print command as we will see later.

Inspecting the data type of a variable

Great! So far we have the following variables declared:

x, y, z, a, b.

What kind of variable is variable *b*? I mean, what type of data does it contain? We know *b* holds your first name, right?

- 4- To inspect the variable use the command *typeof*:

```
typeof b;
```

It displays “*string*“. Variable *b* contains your first name in *quotes* and therefore it is a string of characters.

- 5- What about variable *a*?

```
typeof a;
```

It displays “*number*“. Variable *a* contains the number 347 and therefore it contains data of type *number*.

- **NOTE:** Did you notice how the Console tries to finish the word *typeof*? If you don't want to write the whole word yourself just press the TAB key as soon as you see the hint on the display. Or if you're like me, you'd rather practice typing, just be aware of misspells. The automatic completion can become very handy to speed up programming and avoid typing mistakes, especially when it comes to long commands. Auto typing also helps us remember how each command is written. The test Console is really useful; I use it in real life hundreds of times a day because I'm constantly testing stuff.

What is *typeof*?

typeof is a unary operator. It is called unary (opposite of binary) because it only takes one operand, the variable to the right of it. An example of a binary operator would be the + sign where we need a left and a right operands.

Let's do a few more inspections.

6- What type of data is contained in variables *x*, *y*, and *z*? (test them all)

Answer to all: "*undefined*".

Why it "undefined" and what is undefined?

These variables hold the value of *undefined* because we haven't assigned any data to them yet.

JavaScript reserves some memory for each one of these variables and puts a *placeholder* as data. This placeholder is of type "*undefined*". It is still a value, a special value, and we can take advantage of it for decision making as you will see later.

Clear the screen by right-clicking and selecting “Clear console”. Note: Clearing the screen does not erase the variables from memory; it only gives us more space on the screen.

Let’s assign some values to variables *x*, *y*, *z*:

- 7- Assign the following string value to *x*. Make sure to wrap your string of characters with quotes:

```
x = "The quick brown fox jumps over the lazy dog";
```

- 8- Verify its type of value:

```
typeof x;
```

It now displays “*string*” as a type, rather than “*undefined*”.

- 9- Assign the variable *b* to variable *y*
(assignments go from right to left):

```
y = b;
```

y should now contain a copy of your first name just like *b* does, and its type of value should also be “*string*”.

- **Remember:** Assignments work from the right operand to the left operand. Also, *y* and *b* values are independent of each other. The = sign does not mean equality, it means to assign right to left.

- 10- Assign a value of type *number* (any number) to variable *z*.

- **Remember:** numbers don’t take quotes. If you wrap a number in quotes it becomes a string of characters and you can’t perform any calculations with it.

- 11- For further practicing, declare some new variables of your own, assign some data to them and check their type of value.

Just like any other language, the only way to learn JavaScript is to practice daily. Memorizing will not do it; just plain practice will succeed and persist.

END OF LAB

1.5 Manipulating variable data

When we store a value in the computer's memory under JavaScript supervision, this value inherits some *properties* and *methods* characteristic of the *type* of data the value represents.

Where do these properties/methods come from?

They exist in the JavaScript library located in your browser. When we create a new instance of one of the common types, our instance immediately inherits lots of tools we can use to help us program outcomes.

Methods and Properties

Properties are *qualities* the new data inherits from its data type. It will make more sense later.

Methods are functionality that creates actions upon the data.

- There is no need to memorize these terms. They will be repeated so many times that you will know them by heart before you finish reading the book. Just read along and understand the concepts. However, if you're curious, each term has a link to Wikipedia where you can read to your heart's delight.

To see an example of a property in action let's declare a new variable of the *string* type:

```
var myName = "Tony";
```

Since "Tony" is of type *string*, there are things I can do with this type of data, or things I can find out about *string* data, such as its length in characters:

```
myName.length;
```

The Console will display 4 and it means that the value of variable *myName* which is "Tony", is 4 characters long.

length is one of the properties inherited from the *type string*. Other types of data also use this property because it comes very handy in decision making as you will see over and over again.

Did you notice the **dot** between *myName* and *length*?

The DOT operator

The dot in *myName.length* glues variable *myName* to its property *length*.

The Dot operator connects the variable to its property, or the parent to the child. We place the root first, or the owner of the property. Then map its members or children by using dots:

```
parent.child.grandchild;
```

or in a more explicit example:

```
window.myName.length;
```

window is actually the parent of variable *myName*. We will come back to it in a while.

It's almost like saying "What is the person's name?", Do you see the 's'? That ties the name to the person. The dot notation works under the same principle.

In this case it is pretty simple: we start at the variable name and go down one level to its property *length*. Other times the targeting address may become more complex, perhaps another term or two down the pipe. We will have a chance to work with the dot operator throughout the book.

When working with the *dot operator* we are using what is called **Dot Notation**, but there is another alternative syntax which is called **Bracket Notation**.

Please refer to the next page for a brief introductory explanation of bracket notation.

The Bracket notation

I don't mean to confuse you by mentioning two different notations at the same time. We will be using bracket notation extensively later. Consider this as a primer about it.

Instead of instead of: *myName.length*;

Type in your console the following:

```
myName [ "length" ] ;
```

It displays the number 4, just like it did with Dot Notation. Notice how *length* is wrapped in quotes inside of that bracket. We are passing “length” as a string, or as a label, and JavaScript will reconvert it back to its rightful rank of a property from its library. There are advantages of using Bracket Notation because it is more versatile, but don't break your head over it just yet. We will use both Dot Notation and Bracket Notation. Bracket Notation is mostly used in dynamic situations. You will see later that Dot Notation is “hard wired” and Bracket Notation is “soft wired” (when we don't know the real value at the time of writing the code).

Sometimes we go out on the sedan, and other times we take the pickup truck! It is the same thing with notation syntaxes.

An example of a method

The string data also inherits methods such as **.toUpperCase()**:

```
myName.toUpperCase();
```

It displays “TONY” instead of “Tony”.

Was the variable *myName* changed to uppercase?

No, it wasn't. It only displayed it in upper case mode, but the value remains as it was originally declared.

We could however convert the original value to uppercase by reassigning the variable *myName* to itself as it gets transformed:

```
myName = myName.toUpperCase();
```

Now *myName* contains “TONY” as its value. The original Tony was discarded. Do you see how it happened? We assigned the original variable to its own transformation. It's almost like the dog that was able to catch its own tail.

What's with the parentheses () ?

toUpperCase is a method, it processes some *action*. That sounds like a *function*, which we haven't covered yet. The **()** represents action, or function processing, or *evaluation*.

Whenever you see a pair of parentheses think of something being evaluated by JavaScript, from which a result will be returned back. In this case the data is being converted to uppercase; that's the result of the evaluation.

More about the parentheses later. Don't worry about it for now, just acknowledge it and move forward.

Here's another method: **toLowerCase()**:

```
myName.toLowerCase();
```

Now the value of *myName* is “tony”. Did you notice the T is in lowercase? To transform a string to lowercase except the first character would be called *capitalizing*. Unfortunately there is no direct method in JavaScript to capitalize. It's either all lower case or all uppercase. We can do it with a combination of methods but that is a bit advanced for now. Also, if you know CSS and you are displaying the result in an HTML page you can always use the CSS text transformation property *capitalize*.

- **Note:** both words *toUpperCase()* and *toLowerCase()* have certain characters capitalized and others characters in lowercase, right? JavaScript is case sensitive, it sees **A** and **a** as two different characters.

The reason why I wrote *toUpperCase* and *toLowerCase* in this way was not because camelCase is my preferred writing style; it is because that is the correct name given

to these methods by JavaScript itself. If we write *toLowerCase*, it will not work. As you remember, *typeof* was all in lowercase, we couldn't do *typeOf*. That would not work. This is one of the reasons why the automatic completion feature on the Console is very helpful: it takes the spelling doubts out of the equation.

When you are finished with this book, if you want to study all the properties and methods of all the types in JavaScript, you may consider my other eBook which is a JavaScript follow up from this one:

[JavaScript Objects Functions and Arrays Explained.](#)

There are plenty of exercises and projects in there.

Speaking of exercises, let's do some more lab work.

1.6 Lab work 2

Please fire up your JavaScript Console again
(*CTRL SHIFT j* or *CMD SHIFT j* or *ALT-CMD-j*).

Declaring variables, assigning values, using basic properties and methods

(*See the results on the next page*)

- 1- Declare a variable named **x** and give it a string value of
 “The quick brown fox jumps over the lazy dog”.
- 2- Display **x**;
- 3- Find out the *length* of the data in variable **x**.
- 4- Display **x** in **uppercase**. Hint: `toUpperCase()`
- 5- Now convert the original **x** to uppercase, instead of just displaying it.
- 6- Call **x** again.
- 7- Finally convert the variable **x** to **lowercase** again. Hint: `toLowerCase()`
- 8- Call **x** just to be sure.
- 9- **BONUS**:
 Declare variable **y** and assign it the value of **x**, but in uppercase.
- 10- Call **y**.
- 11- **EXTRA, EXTRA BONUS**:
 Declare a variable named **z** and assign to it the numeric value of the *length* of **x**.
- 12- Call **z**.

(See the results on the next page. At this time, do the same exercises as you verify the answers on the next page. Practice, practice, practice)

Results:

- 1- Declare a variable named *x* and give it a string value of *"The quick brown fox jumps over the lazy dog"*:

```
var x = "The quick brown fox jumps over the lazy dog";
```

- 2- Display *x*:

```
x;
```

It displays *"The quick brown fox jumps over the lazy dog"*

- 3- Find out the *length* of the data in variable *x*:

```
x.length;
```

It displays *43*, as of *43* characters including spaces.

- 4- Display *x* in uppercase:

```
x.toUpperCase();
```

It displays *"THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG"*

- 5- Now convert the original *x* to uppercase, instead of just displaying it:

```
x = x.toUpperCase();
```

- 6- call *x* again:

```
x;
```

It displays *"THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG"*

- 7- Finally convert the variable *x* to lowercase again:

```
x = x.toLowerCase();
```

- 8- Call *x* just to be sure:

```
x;
```

It displays *"the quick brown fox jumps over the lazy dog"*

- 9- *BONUS*:

Declare variable *y* and assign it the value of *x*, but in uppercase.

```
var y = x.toUpperCase();
```

- 10- Call *y*:

```
y;
```

It displays *"THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG"*

- 11- *EXTRA, EXTRA BONUS*:

Declare a variable named *z* and assign to it the numeric value of the length of *x*:

```
var z = x.length;
```

12- Call z :

z ;

It displays 43, the value of z is 43.

END OF LAB

1.7 Adding some more JavaScript operators

`+, -, +=, -=, /, /=, %, *, *=`

+ If both operands are numbers, it adds the left operand to the right operand.
If one or both operands are strings, it concatenates the two words or characters.

- Subtracts the right numeric operand from left numeric operand.

+= Adds left to right and automatically **assigns** to the left^{**}.

-= Subtracts right from left and automatically **assigns** to the left^{**}.

/ Divides the left operand by the right operand.

/= Divides the left operand by the right operand and automatically **assigns** the *quotient* to the left operand^{**}.

% modulu (fetches the remainder of the left operand divided by the right operand).

***** Multiplies the left operand by the right operand.

***=** Multiplies the left operand by the right operand and automatically **assigns** the product to the left operand^{**}.

^{**} Notice: Because **+=**, **-=**, **/=** and ***=** are assignment operators, the left operand must be a variable, it cannot be a direct number.

We will expand these concepts as we code along.

The + plus operator

The + sign has a dual function depending on the type of data JavaScript is processing.

In a string type, the + operator [concatenates](#) words like in the following examples:

```
"Hello" + "World!";
```

JavaScript puts them together as "HelloWorld!"

If you really want to space out the words there are three possible ways to accomplish the task:

(1)

```
"Hello" + " " + "World!";
```

(2)

```
"Hello " + "World!";
```

(3)

```
"Hello" + " World!";
```

On the first example I've added *an independent* blank space by wrapping it in quotes.

On the second example I've placed a space after Hello and before the closing quotation mark.

On the third example I've added a trailing space on the word World!

The first example is useful when we add two variables together because we cannot combine leading or trailing spaces with variables due to lack of quotation marks.

Here's what I mean:

```
var x = "Hello";
```

```
var y = "World!";
```

Now I concatenate both variables together and assign them to some other variable:

```
var z = x + y;
```

The result for z is "HelloWorld!". The only way to add a space between **x** and **y** is to use the first technique:

```
var z = x + " " + y;
```

Now **z** contains "Hello World!".

Using the + plus operator to add two numbers

When it comes to numbers, the + operator adds the operands like shown on the following example:

```
3 + 4 + 5;
```

That expression will result in 12.

```
var a = 6;
```

```
var b = 10;
```

```
var c = a + b;
```

Variable *c* is assigned to 16 which is the sum of *a* and *b*.

Using the += assignment operator

The += is very useful. Suppose variable *i* has the value of 3, and you want to add 7 to it, I mean 3 + 7:

```
var i = 3;
```

We could do it this way:

```
i = i + 7;
```

That will assign variable *i* to 10.

However, this operation is so common (as you will see ahead), that programmers created a shortcut operator, the += (call it *plus equals* and never write equals plus because it will not work):

```
i += 3;
```

Whenever you see += it means to take the value of the left operand and add the right operand to it.

So now *i* which was 10 is now 13. What happens if I do it again?

```
i += 1;
```

Now *i* is 14.

What about this one:

```
i += i; <- that's i not 1.
```

The above example is the same as 14 + 14, which results in 28.

The += operator can also be used in strings to add more characters to the right side of the original value:

```
var n = "T";
```

```
n += "o"; <- results in "To"
```

```
n += "n"; <- results in "Ton"
```

```
n += "y"; <- results in "Tony"
```

This can be very useful in *loops* in order to automatically construct a phrase or a list of names as we will see later.

The ++ (double plus) incrementing operator

Many times we want to increment the value by just 1 unit. This is a very common technique in programming.

We could do it in three ways:

Assuming **var x = 5;**

x = x + 1; <- now x is 6

x += 1; <- now x is 7

x++; <- now x is 8

x++; <- now x is 9

This last way of incrementing is limited to the value of 1 at a time. The other ways are more versatile since we can increment by as much *at* a time as we wish. However, the ++ operator is extremely popular in loops because loops normally (but not always) increment in single steps. We will come back to it when we get to loop around a bit.

The -, — and -= subtraction operators

The subtraction operator is only used in numbers. It cannot be used to subtract characters from a string.

`5 - 2;` \leftarrow The result is 3

Examples:

`var x = 100;`

`x - 5;` \leftarrow The result is 95, but x is still 100 because it was not re-assigned.

`x = x - 5;` \leftarrow Now x is 95.

`var y = x - 100;` \leftarrow The result for y is -5, because x is 95

We can also use the subtraction [unary](#) operator if we want to subtract by 1:

`y-;` \leftarrow now y is -6

This operator — is the same as `y = y -1` or `y -= 1;`

Read the `-=` as “minus equals” (never as equals minus).

The Division / or /= Operators

To divide two numbers we use the forward slash:

```
8 / 4; <- It results in 2
```

Examples:

```
var x = 12;
```

```
var y = 3;
```

```
var z = x / y;
```

Variable `z` gets assigned the division of 12 by 3, resulting in 4.

We could also assign the result of a division back to the original variable:

```
x = x / 3;
```

or as a shortcut:

```
x /= 3;
```

The “divide/equals” is not used as much as `+=` or `-=` but it is good to know it exists.

The Modulus % Operator

Sometimes we don't want the result of a division, I mean its *quotient*. We may be looking for the *remainder* or the *modulus* of the division. This is very common in programming so please make sure to understand it.

When I divide 14 by 3, I get a result of 4 and also a remainder of 2. If my intention is only to get the remainder I can ask JavaScript to get the remainder with the modulus % operator:

```
14 % 3;
```

JavaScript divides the two numbers and displays the remainder of the division which is 2.

The modulus becomes very handy for things like figuring out odd and even numbers.

In **12 % 2**, the result is zero which tells me that 12 is an *even number*.

In **13 % 2**, the result is 1 which tells me that 13 is an *odd number*.

in **2014 % 4**, the result is 2 which tells me that 2014 is not a leap year since leap years are divisible by 4.

In **2016 % 4**, the result is zero, which tells me that 2016 will is a leap year.

There are many applications for the modulus operator and we will play with it on some exercises throughout the book.

The Multiplication Operator *

To multiply numbers or numeric variables we use the *asterisk* because we can't use the x symbol. x is normally used as a variable name. So in computer science we use the asterisk to represent multiplication:

```
3 * 6; <- results in 18
```

```
var x = 7;
```

```
var y = 5;
```

```
var z = x * y; <- results in 35
```

When it comes to assigning the result back to the original variable we can do it in two different ways:

```
x = x * 2; <- x is now 14
```

or

```
x *= 2; <- x is now 14
```


1.8 Lab work 3

Let's practice a bit to cement the concepts covered in this chapter.

Please fire up your Console with **CTRL SHIFT j**

or **CMD SHIFT j** for Mac

or **ALT-CMD-j**.

Part one: working with +, +=, ++ and —

- 1- Add 3 and 4
- 2- Declare a variable *x* with the value of 3 and a variable *y* with the value of 5.
Then create variable *z* and assign to it the sum of *x* and *y*.
Call variable *z*, it should contain the value 8.
- 3- *x* is 3. Increment *x* to 4 using the unary operator ++. Then call *x* to see the result.
- 4- Decrement *x* by 1, using the unary operator —. Then call *x* to see the result.
- 5- Concatenate “Hello” to “World!”. Make sure to include a blank space between the two words.
- 6- Declare a variable named ***myName*** and assign it only the first character of your name.
- 7- Now add the second character of your name to ***myName*** using the += operator.
- 8- Do the same for all the other characters of your name. Call variable *myName* to see the result.

(see the results on the next page)

Part one results

- 1- Add 3 and 4:

```
3 + 4;
```

- 2- Declare variable *x* with the value of 3, and variable *y* with the value of 5. Then create variable *z* and assign to it the sum of *x* and *y*. Call variable *z*, it should contain the value 8.

```
var x = 3;
```

```
var y = 5;
```

```
var z = x + y;
```

```
z; <- should be 8
```

- 3- *x* is 3. Increment *x* to 4, using the unary operator ++. Then call *x* to see the result.

```
x++; <- Displays 3 (see note below)
```

```
x; <- x is now 4
```

Note: You may see the number 3 displayed on the Console instead of 4. That is just an automatic feedback from the Console. The ++ increment happens after the feedback occurs. Had we programmed it as ++x instead of x++ the increment would have happened before the automatic feedback from the Console. The end result for the variable x is 4 in both ways. We will cover this in more detail at a later lesson when we get to loops. For the most part stick to **x++**, it is the common way of writing it.

- 4- Decrement *x* by 1 using the unary operator --. Then call *x* to see the result.

```
x--;
```

```
x; <- It displays 3.
```

- 5- Concatenate "Hello" to "World!". Make sure to include a blank space between the two words.

```
"Hello" + " " + "World!";
```

or this way:

```
"Hello " + "World!";
```

or this way:

```
"Hello" + " World!";
```

- 6- Declare a variable named **myName** and assign it the first letter of your name.

```
var myName = "T";
```

- 7- Now add the second letter of your name to **myName** using the += operator.

```
myName += "o";
```

- 8- Do the same for all the other letters of your name. Call variable *myName* to see the result.

```
myName += "ny";
```

```
myName; <- it displays "Tony".
```

Part two: Working with `*`, `/`, `*=`, `/=` and `%`

Clear the Console display by right clicking on it and select “Clear Console”.

Let's work with multiplication, division and modulus:

- 1- Declare three variables: $x = 12$, $y = 2$ and $z = 3$
- 2- Declare another variable, ***a***, and assign to it the multiplication of ***y*** by ***z***. Call ***a*** to see if it holds the value 6.
- 3- Declare variable ***b*** and assign it the result of the division of ***x*** by ***a***. Call ***b*** to inspect its result.
- 4- Reassign ***b*** to the multiplication of itself. Use `*=` for that effect. Now ***b*** should be 4.
- 5- Reassign ***b*** to the division of itself. Use `/=`. Now ***b*** should be 1.
- 6- The value of ***x*** is 12. Use the *modulus* operator to grab the remainder of dividing ***x*** by 2. Do not assign it to anything, just display the result. It should be *zero*.
- 7- At this moment ***x*** is still 12. Use the *modulus* operator to see what the remainder is when we divide ***x*** by 5. It should be 2 left.

(see the results on the next page).

Part two results (answers):

- 1- Declare three variables: $x = 12$, $y = 2$ and $z = 3$.

```
var x = 12;
```

```
var y = 2;
```

```
var z = 3;
```

- 2- Declare another variable, a , and assign to it the multiplication of y by z . Call a to see if it holds the value 6

```
var a = y * z;
```

```
a;
```

- 3- Declare variable b and assign it the result of the division of x by a . Call b to inspect its result, it should be 2.

```
var b = x / a;
```

```
b;
```

- 4- Reassign b to the multiplication of itself. Use $\ast=$ for that effect. Now b should be 4.

```
b *= b;
```

- 5- Reassign b to the division of itself. Use $/=$. Now b should be 1.

```
b /= b;
```

- 6- The value of x is 12. Use the *modulus* operator to grab the remainder of dividing x by 2. Do not assign it to anything, just display the result. It should be *zero*.

```
x % 2;
```

- 7- At this moment x is still 12. Use the *modulus* operator to see what the remainder is when we divide x by 5. It should be 2 left.

```
x % 5;
```

There will be plenty more exercises to use these new skills along with other more advanced ones.

Let's keep going forward.

END OF LAB

1.9 Comparison operators and booleans

`===, !==, <, >, <=, >=` **true, false**

In math we use the equals sign to determine *equality*.

Not in JavaScript! In JavaScript the equals sign is an **assignment** operator: it assigns the right operand to the left operand:

```
x = 12;
```

In the example above, 12 is given to x. This is an assignment, not a *question* to inspect equality.

To inspect equality, we use the triple = operator `===`

```
x === 12;
```

The expression above is not an assignment, it is a **question** and the question is: “*are the values in x and in 12 the same?*” in other words, does the variable x contain the numeric value 12?

Every time JavaScript interpreter sees a triple equals (`===`), it **always** evaluates and replies with a Boolean *true* or a Boolean *false*.

What is a Boolean?

A Boolean is a binary evaluation result that has one of two possible value outcomes: **true** or **false**. In JavaScript a Boolean is a type of value just like string, number, undefined, null, etc.

true and **false** are permanently reserved words in JavaScript. We can literally write true and false without quotes because JavaScript knows what they mean. However, if we ever write “*true*” or “*false*” in quotes, JavaScript will assume it is a string which has no value as a Boolean. So if you ever see “*true*” in quotes, it is because that information is being spoken to a human being, not to the computer itself. JavaScript only accepts *true* and *false* as Boolean values when the terms are not wrapped in quotes and in lower case.

So, we could create a question on the console for JavaScript to answer (this question is a command to get a Boolean result):

```
x === 12;
```

If we have previously declared x with the value of 12, JavaScript will reply **true** because the statement is true.

Else, it will reply **false**, like in the following example:

```
x === 14;
```

Question:

In `x === 14` what would JavaScript say if x was not previously declared as a variable?

Since `x` is not wrapped in quotes, it is not a string value and therefore it must be a variable name. However JavaScript doesn't find it in its library because it hasn't been declared.

In that case, instead of replying with a Boolean *true* or *false*, JavaScript will throw an error like this:

ReferenceError: x is not defined

Let's learn a few more comparison operators and ask more questions to JavaScript.

Please remember, these symbols compare the left operand, to the right operand.

Less than: `<`

Greater than: `>`

Less or equals than: `<=`

Greater or equals than `>=`

Not the same as: `!==`

Examples:

`12 < 10;` The Boolean answer is *false*

`12 > 10;` The Boolean answer is *true*

`12 === 12;` The Boolean answer is *true*

`12 === "12";` The Boolean answer is *false**

`12 !== 13;` The Boolean answer is *true*

`12 !== 12;` The Boolean answer is *false*

`12 <= 12;` The Boolean answer is *true*

`12 = 12;` The Boolean answer is *true*

***Note:** You may have seen double equals `==` used in JavaScript in another book somewhere. In a different programming language that should be ok. However, in JavaScript the double equals comparison operator is not recommended. This is because in JavaScript the double equals `==` does not fully check the condition. I mean it does not check the type of data the operands contain. That could lead to errors because sometimes we have numbers in quotes which makes them strings, a "12" is not the same value as 12. The first is a string of characters and the second is a number.

In JavaScript always use the triple equals: `===` for comparing values.

1.10 Lab work 4

Practicing with `===`, `!==`, `<`, `>`, `<=`, `>=` `true`, `false`.

Go to your favorite JavaScript Console and create some variables and values.

- 1- Ask JavaScript if 12 is greater than 14.
Then ask if 12 is smaller than 14.
- 2- Declare variable *num* with a value of 12.
Then declare variable ***num2*** with the value of **“12”**.
Later, ask if *num* is the same as *num2*.
- 3- Ask if *num* is not the same as *num2*.
- 4- Ask if 7 is *greater or equals* to 8.
Also if 7 is *less or equals* to 7.
- 5- What is the difference between ***true*** and “*true*”?
- 6- Find out the *typeof* ***true***.
- 7- Find out the *typeof* **“true”**.
- 8- Find the type of value **“7”**.
- 9- Find the type of value 8.

(see results on the next page).

Results:

- 1- Ask JavaScript if 12 is greater than 14.

Then ask if 12 is smaller than 14.

```
12 > 14;
```

```
false
```

```
12 < 14;
```

```
true
```

- 2- Declare variable *num* with a value of 12.

Then declare variable *num2* with the value of "12".

Later, ask if *num* is the same as *num2*.

```
var num = 12;
```

```
var num2 = "12";
```

```
num === num2;
```

```
false
```

- 3- Ask if *num* is not the same as *num2*.

```
num !== num2;
```

```
true
```

- 4- Ask if 7 is *greater or equals* 8.

Also if 7 is *less or equals* 7.

```
7 >= 8;
```

```
false
```

```
7 <= 7;
```

```
true
```

- 5- What is the difference between *true* and "true"?

true is a reserved Boolean term.

"true" is just a string of characters.

- 6- Find out the *typeof* true.

```
typeof true;
```

```
"boolean"
```

- 7- Find out the *typeof* "true"

```
typeof "true";
```

```
"string"
```

- 8- Find the type of value "7".

```
typeof "7";
```

```
"string"
```

- 9- Find the type of value 8.

```
typeof 8;
```

```
"number"
```


1.11 Introducing console.log, alert, prompt, confirm

So far we have been testing the value of variables by just typing the name of the variable and pressing the ENTER key.

This works because we are using a *test Console*.

However, as our scripts get more complex we will need to have a more explicit mechanism for outputting *data* to the screen.

That's where *console.log()* comes handy .

The console.log

The Console has a way to explicitly log data to the screen by using a mechanism called *log()*.

The data to be displayed goes inside of the parentheses. If this data is in the form of string we wrap it *in quotes*. However, if the data comes from a variable or from a number, we always write the variable name or the number without quotes.

Examples:

```
log("Hello bluebird");
```

 <- This will not work as written.
Please wait to read the note on the bottom of the page before you try it on your Console.

Another example is *log(x)*. If *x* is a declared variable, this will display the value of *x*. If *x* is not a variable, JavaScript will throw an error because any word that is not a variable should be in quotes like this: *log("x")*.

We can also combine strings and variables:

Like for example:

```
var x = "big bird";  
log("Hello " + " " + x);
```

Which displays *Hello big bird*

Or if you remember, we could do it this way:

```
log("Hello " + x);
```

NOTE:

log() by itself will not work. We need to use **Dot syntax** to bind *log()* to the Console (lower c):

```
console.log("Hello " + x);
```

That is the correct way of using *log()*, attaching console to *log()* with dot syntax. Notice how console is in lower case. The correct way to refer to the Console when programming is console in lowercase. It is an object named *console*. Capitalization for console will not work because JavaScript is case sensitive.

Here are some more examples:

```
console.log( 3 + 5 );
```

 <- It displays 8

```
console.log( 3 > 5 );
```

 <- It displays false

Or a combination of a string and a number comparison:

```
console.log("3 > 5 is " + (3 > 5) );
```

It displays: ***3 > 5 is false***.

We will use `console.log()` extensively throughout the book.

alert()

Before `console.log()`, there was `alert()`. Contrary to `console.log()`, `alert()` is a true JavaScript command (`console.log` is not really from the JavaScript core, it is a method introduced by browsers and for test purposes). Programmers were getting tired of having to create a popup box with `alert()` just for testing a quick output, and that's how `console.log()` was invented.

When we transfer a program from testing to production, we need to convert all `console.log()`s into something that works with HTML, and `alert()` is one of such output statements. There are many other output implementations, which is a good thing since `alert()` is synonymous with ugly advertisement popup boxes.

Just like `log()` is called from the Console, such as `console.log("Hello!");` `alert` is called from object *window*, like this:

```
window.alert("hello!!");
```

Why window?

The object *window* is the mother of everything we do on a browser. Even the Console itself runs under *window*. (no relation to Microsoft Windows). The complete statement is actually:

```
window.console.log("hello!");
```

However, since we are coding inside of the Console, we don't really need to explicitly address the outer part of the Console in the browser. I mean, `console.log()` is good enough since JavaScript starts looking from where we are coding and moves outward from there. So it finds the command `log()` as part of the Console. In any case, even *alert* does not need to use its parent *window* in order to work (because *Console* is a child of *window* and a child has access to its parent's variables, properties and methods.).

This is the most popular way to code an alert:

```
alert("hello!");
```

We will have an opportunity to try using `alert()` on the next lab assignment.

confirm() and prompt()

Another popular popup box is *confirm()*

Just like `alert()`, `confirm()` also comes from object *window* and we normally code it like this:

```
confirm("are you sure you want to proceed?");
```

It pops up a message and we click one of the two possible buttons:

Ok or *Cancel*

Ok will return a Boolean **true**, and *Cancel* will return a Boolean **false**. We don't have to take advantage of these Booleans, but we could and we will when we do a project on *conditional statements*.

Another useful output method is *prompt()* which also acts as an input method.

prompt() is very useful to gather information from a user because `prompt()` will open a widow where the user can introduce some data. We can then assign this data to a variable and use the value to create some other outcome.

For example:

```
var answer = prompt("yes or no?");
```

Then we could further manipulate the answer by evaluating it with a Boolean result:

```
answer === "yes";
```

That is actually a question for which JavaScript will answer back as **true** if the answer is "yes", or as **false** if the answer is "no".

This will make more sense when we arrive a *conditional statements*. I just don't want to give you all the information in bulk when you get there and that's why I'm introducing these concepts right now. Hopefully when we get to conditional statements you will remember the basic ingredients used on a *conditional recipe*, but don't break your head over these concepts, we will revisit them again.

1.12 Logical Operators

And: **&&**, Or: **||**, Not: **!**

There are three logical operators in JavaScript.

And: **&&**

With **&&** both operands need to be Boolean *true* in order to get a *true* output.

Examples:

(10 > 9) && (10 < 11); <- The JavaScript answer is *true* because true and true is true.

(10 > 14) && (10 < 11); <- The JavaScript answer is *false* because false and true is false.

5 === 7 && 5 === 5; <- The JavaScript answer is *false*

5 === "5" && 5 === 5; <- The JavaScript answer is *false*

5 === 5 && 7 > 5; <- The JavaScript answer is *true*

Or: **||**

The two vertical bars denote an OR logical operator. In US Windows you can access the **||** by pressing SHIFT and the last key on the right at the row starting with *qwerty*.

With logical or **||**, one of the operands needs to be true in order to get a Boolean true as the output.

Normally JavaScript will not check the right operand in an OR statement, if the left operand is already true. This is important to know, especially when it comes to conditional decision making as we will see soon.

Examples:

(10 > 9) || (10 < 9); <- The JavaScript answer is *true* because the first expression is true.

(10 < 9) || (10 > 9); <- The JavaScript answer is *true* because the second expression is true.

(10 < 9) || (10 < 8); <- The JavaScript answer is *false* because none of the expressions are *true*.

Not: !

The not ! operator is used to invert a statement. In JavaScript any data value is considered *true*, with the following exceptions:

Boolean *false*, the number *zero*, the value known as *undefined*, and the value known as *null* which we haven't covered yet.

Let me declare a few variables and values in order to do some tests:

```
var x = 3;  
var y = 0;  
var z = "Tony";  
var a;
```

Based on my first statement, is variable *x* *true* or *false*? It is true because it has a value. On the other hand, variable *y* is false.

Let me introduce a JavaScript method to find out when a value is *true* or *false*: (we can use this method in our scripts for decision making but it is not used that much. Just be aware of it):

Boolean();

And now we are going to compare the normal result of the method with the result of inverting the question. Remember, we are asking JavaScript to see if the statement is *true* or *false*, not to inspect the variable and give us a value. It's all about our statement inside of the Boolean() method. Let's try:

In for example, Boolean(x); We are asking the following: "Is this true?", "I say x exists as true, is my statement true?", and then JavaScript replies with *true* or *false*:

Boolean(x) ; <− the statement is *true* (since our x from the previous examples is 3)

Boolean(!x) ; <− the statement is *false* (since x has a value, !x is a false statement)

Boolean(y) ; <− the statement is *false* (y is 0 and zero is false)

Boolean(!y) ; <− the statement is *true* (since y is 0, !y is a true statement)

Boolean(z) ; <− the statement is *true* (z is "Tony" and Tony is true)

Boolean(!z) ; <− the statement is *false* (since z has a value, the !z statement is false)

Boolean(a) ; <− the statement is *false* (a is *undefined* and we can't say it is true)

Boolean(!a) ; <− the a statement is *true* (since a is *undefined*, !a is a true statement)

Let's do some lab work to review all these concepts.

1.13 Lab work 5

Getting feedback from a user

In this lab session we will work together step by step. Please turn on your JavaScript Console and play along with me:

- 1- Let's use *prompt()* to ask a visitor for his/her name:

```
prompt("What is your name");
```

- 2- A popup window should appear on your browser after you press ENTER, asking for your name.

Answer the question but do not write your name in quotes. Being a *text box* you don't need quotes. The input mechanism will put the quotes automatically:

Write *Tony* or your first name, and then press ENTER

Ok, what happened? Nothing much happened, right? The name was displayed on the Console as an automatic feedback and that's about it.

This is because we didn't *catch* the input name given by the user to the *prompt()* mechanism. In order to grab the input name, we need to assign the result to a variable.

Let's start from the beginning by doing it all over again:

- 3- Declare a variable name ***userName*** and assign to it the same *prompt()* as before. I'm not using the word *name* by itself because it is a reserved word in the browser, which makes it a forbidden term to use as a variable name):

```
var userName = prompt("What is your name?");
```

When the popup window appears type your first name on the box and press ENTER.

We still can't see the name, but on our next step we are going to use `console.log()` to display the value of variable ***userName*** which should be your first name. *By the way, the automatic feedback message was probably the term *undefined*, right? That's because the feedback happened when the variable was declared, and then the assignment happened. You don't need to know this; I just want to inform you so that you don't get distracted trying to figure out what really happened there.*

- 4- Use `console.log()` to display the value of `userName`:

```
console.log(userName);
```

Mine displays *Tony*

If you get an error like for example "ReferenceError: myName is not defined", make sure the name of the declared variable and the variable name you used on the `console.log()` are the same. A common mistake is to write *username* and then on the second time write *userName*. As you know, they are not the same.

- 5- Let's make our output a bit prettier. Use `console.log` to display the following sentence:

“Hi, Tony! Nice to meet you” where Tony is the name you have entered on the prompt. You will have to use the variable *userName* as a placeholder (see the next explanation).

```
console.log("Hi, " + userName + "! Nice to meet you.");
```

Notice the white space after *Hi*, also notice that I have used the variable *userName* instead of the real name. That's because as programmers we create scripts to work with any name, not just our name. Always code generically!

- 6- Did it work for you? Did you get the correct name displayed on your output sentence?

Now it's your turn...

On the next few exercises you are going to repeat what we've just done. Are you ready to do it on your own? Please read the following recipe instructions and then program it yourself.

- 1- Declare a variable *color* and assign to it a *prompt()* asking a user to define his/her favorite color.
- 2- Program a console.log that says *Hmmm... orange is also one of my favorites!* where orange is the variable color which represents the color picked by the user.
- 3- For practicing purposes, add an extra line with a *confirm()* to display this message: "Are you ready for more JavaScript?".

(See the results on the next page).

Results:

- 1- Declare a variable *color* and assign to it a *prompt()* asking a user to define his/her favorite color:

```
var color = prompt("What is your favorite color?");
```

- 2- Program a console.log that says *Hmmm... orange is also one of my favorites!* where *orange* is the color they have picked.

```
console.log("Hmmm.. " + color + " is also one of my favorites!");
```

- 3- For practicing purposes, add an extra line with a *confirm()* to display this message :
"Are you ready for more JavaScript?":

```
confirm("Are you ready for more JavaScript?");
```

END OF LAB

Congratulations!

You should now be ready for the second level of this programming adventure.

Let's start by introducing conditional branching techniques so that we can go a little faster on the highway!

Thanks for sticking around.

PART II: ENTERING THE SECOND REALM

“I try all night to play a pretty note.”

Jimi Hendrix

2.1 Adding more syntax

Code blocks - Using an editor

Up to this point we have been writing *one liner* statements and as soon as we press ENTER on the Console, the code gets executed. We could actually write several lines of code in the Console by holding the *shift-key* when we press *Enter*. That would take us to the second line where we could continue writing our script. Although this works, it is not an ideal arrangement. It is much better to write code on a **text editor** and then copy/paste it to the Console when we are finished.

I'm sure you have some sort of a text editor on your computer. Do not use *Word* or *WordPad* because they add hidden formatting code which interferes with your script. Microsoft Windows comes with *Notepad*. Notepad works and I use it hundreds of times a day to filter out text as I copy from one source to paste onto another because it strips any visible or hidden format. But I also use other free editors such as the free [Notepad++](#), [Programmer's Notepad](#), and [Brackets](#), to write real code.

NOTE: Since May 2015, Microsoft has released a really cool free source code editor called [Visual Studio Code](#) which works in Windows 7 and up, Linux and Mac. This is highly recommended but for now I am going to recommend something much simpler to do our exercises and you will see why. Please read on.

Of course you may use any one of your favorite *plain* text editors, but let me introduce to you an online editor that comes really handy because you don't have to rely on your computer to get going. I use this online tool many times a day because it also allows me to beautify the code when I click the **beautify** button (creating proper white space):

jsbeautifier.org

Please **Bookmark** the site. The only disadvantage with this arrangement is that you can't save the code directly, but right now we are not saving anything, we are experimenting and jsbeautifier works well for this purpose because it actually teaches us how JavaScript syntax should be presented.

What is a code block?

A code block is usually a collection of different statements that come together to create a certain functionality. The best way to save groups of code is by creating a *function*. We will get to play with functions soon.

A code block is usually limited (surrounded) by an opening curly brace **{** and a closing curly brace **}**.

Here's an example of a code block:

```
{  
  var x = "Hello";  
  console.log(x);  
}
```

We will use code blocks from our next project on.

Commenting code

Sometimes we want to make a note in our code to explain the purpose of a code line to another human being but we don't want JavaScript to throw an error when it sees our comment.

There are two ways to make comments:

1- **Using two forward slashes.**

With this style of commenting we can't press the Enter key and write a second comment line. It must be done all in one continuous line:

```
// This is a comment. Hello fellow reader, how are you?
```

2- **Using a forward slash and an asterisk to start the comment and an asterisk and a forward slash to end the comment: /* */**

This comment can span for as many lines as you wish:

```
/* With this comment I can comment a whole page  
and create as many lines as I want. Once upon a time in  
a very far away land called New Jersey, formerly known  
as Schejachbi, there was an American Indian tribe known  
as Lenape... */
```

Let's move on to *conditional branching* which is a very important step into real programming.

2.2 Conditional Branching

We have just learned how to get feedback from a user but we could not do much with it because there were no tools to make decisions based on the user's given feedback. Well, we are going to get some of those tools right now.

Suppose we ask a user if he/she wants to play along and based on the user's answer, we either keep playing or exit the program. It sounds like we need to apply an `if()` else conditional code statement, right?

if, else conditionals

To make branching decisions in JavaScript we need to know about the *if()*, *else* conditional statements.

Branching is when we either go this way, or go that way, depending on the Boolean statement presented at the point of branching. In other words, “*if true go this way, else go that way*”.

The basic *if, else* syntax is as follows:

```
if(Boolean true) { do this stuff; } else { do that stuff; }
```

The *else* part is optional.

Notice the term **Boolean true**. This is a very important concept that many code students miss at the beginning. Inside of the parentheses we have a *lock*. Independently from what you write inside of those parentheses, the condition must always evaluate to a Boolean *true* in order to unlock the code within the subsequent curly braces. The Boolean **true** is always the correct key to unlock the *if()* statement. JavaScript will only execute the code on the next code block if that condition is true. On the other hand, if the condition evaluates to *false* the code block will be hidden from JavaScript, and JavaScript will exit or execute the next code block if there is one available, example: *else { }*.

- An *if()*, *else* statement is in a sense a logical OR (or XOR):
Either the first code block is executed, or the *else* code block is execute, but never both.
Notice how the else does not have a condition, it works automatically when and only when the Boolean `if()` evaluates to *false*.

So when the Boolean condition is *false*, JavaScript jumps over the curly braces and moves on. In this case we can catch the JavaScript execution with an optional *else* statement. The *else* statement does not take conditions; it is just a catcher in case the first condition is not met. We include an *else* option when we want JavaScript to choose one of two paths.

Sometimes we want a certain code to execute only if a condition is met, but we do not have an alternative condition. In this case, we use the *if(){} by itself* without an *else{}.*

An *if()* statement by itself is like saying “*Nothing to do here. Let's keep going*” when the condition is false. Or “*Wait! There is something to do here. Let's take a detour and do this*

thing before we keep going” when the condition is true.

- **Please note:** From now on, some lab work and theory lectures are going to coexist. Please code along as we discuss the subject. **Part two** is going to be a *hands-on* adventure.

Let's construct an example just to see how the *if()*, *else* works. Since some of these scripts span for several lines, I have provided a link to an original file which is located on one of my support websites. I recommend writing your own files because you will not master the language if you copy/paste them, but at times it becomes convenient to just copy and paste. *Use your best judgment.*

The link to the file on my server will be located below the image, like in the example seen on the next page.

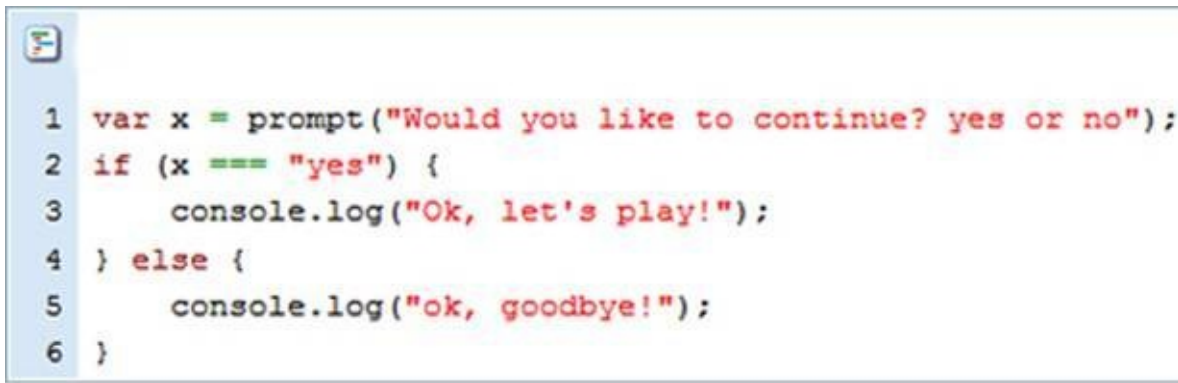
Let's start coding:

- 1- On your editor (or jsbeautifier.org), assign a *prompt()* to a variable *x*. In the *prompt()*, ask the user to say *yes* if he/she want to continue, or to say *no*, if he/she wants to leave the program. Something like this:

```
var x = prompt("Would you like to continue? yes or no");
```
- 2- Construct an `if(x === "yes")` statement that prints to screen the message "Ok, let's play!"
Otherwise, it prints to screen "Ok, goodbye!"

After you finish coding, if you are using jsBeautifier, press the button "Beautify JavaScript", or CTRL-ENTER to make your code more presentable. Notice how the editor adjusts the spacing to conform to standard JavaScript styling.

See the image below.



```

1 var x = prompt("Would you like to continue? yes or no");
2 if (x === "yes") {
3     console.log("Ok, let's play!");
4 } else {
5     console.log("ok, goodbye!");
6 }

```

Fig 1 See original file:([here](https://bit.ly/1uN96zN)) bit.ly/1uN96zN

- 3- Copy your script from your own editor *or from my own linked page* and paste it on the Console. Then press ENTER.

If you get an error, recheck your code. Debugging is what programmers do best because that's where they spend 90% of their time (see step 4 to learn how to repeat the execution on the Console).

When the prompt appears, type **yes** without quotes. If your script is well constructed you should have the following printout on your Console:

Ok, let's play!

To copy the code from my own website, click on the *2-page icon* located to the right of the code script.

- 4- **To repeat the execution**, focus your mouse on the Console and press the **Up Arrow**. That will take you up to the last executed script. From there, (you can edit it if you need to do so) and then press *Enter* to run it again.

This time, answer **no** to the prompt question.

You should get: **ok, goodbye!** as a response.

- 5- Did you notice how the *prompt()* takes you to the main browser but then you have to look back at the console for an answer?

You could replace *console.log()* with *confirm()* or with *alert()* in order to also get your answer on the browser.

Just for practicing purposes, change your first console.log to **confirm()** and your second console.log() to **alert()** . Don't worry about this inconstancy, we are just practicing.

Use the editor at jsbeautifier to write your script.

Then test both *if, else* outcomes like we did on steps 3 and 4:

```

confirm("Ok, let's play!");
alert("ok, goodbye!");

```

Reviewing if(), else

Inside of an `if() { do this stuff; }` parentheses, we include a condition. Keep in mind that it is not the condition itself that unlocks the code block: it is the truthiness of the *question* you write in there that unlocks the code.

Examples:

`if(10 === 10) {}` ← since 10 is the same as 10, JavaScript will evaluate the statement as **true** and it unlocks the next code block.

`if(10 === "10") {}` ← since 10 is a number and "10" is a *string*, JavaScript evaluates the statement as **false** and jumps over the next code block without executing it.

`var x = 33;`

`if(x) {}` ← since *x* has a value other than 0, *undefined*, or *null*, the statement is **true**.

`if(!x) {}` ← since *x* has a value other than 0, *undefined*, or *null*, this statement is **false**.

`var y = "yes";`

`if(y === "yes") {}` ← since the value of *y* is string *yes*, this statement is **true**.

`if(y !== "yes") {}` ← since the value of *y* is *yes*, this statement is **false**.

`if(y === "no") {}` ← since the value of *y* is *yes*, this statement is **false**.

`if(y === "no" || y === "yes") {}` ← since one of the statement is *true*, the statement is **true**.

`if(y === "no" && y === "yes") {}` ← this statement is **false** since one of the operands is *false*.

What about else?

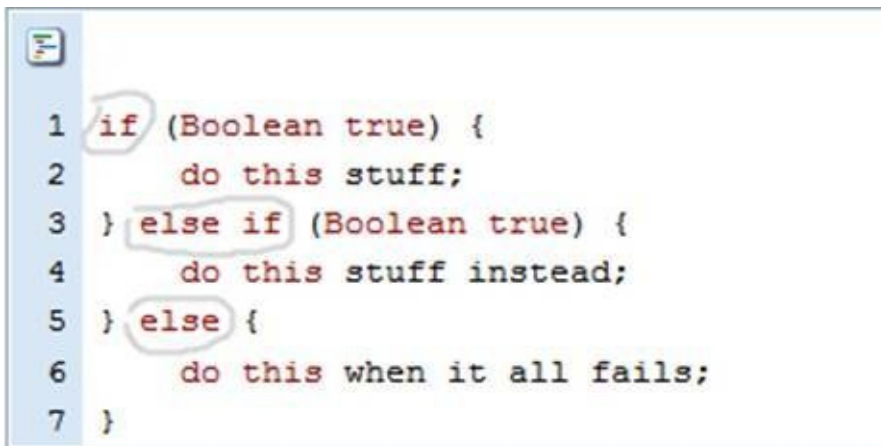
else is optional. We can have an `if()` conditional statement all by itself. However, we can't have an *else* by itself since *else* is just a catcher helper for when *if()* fails. Or in other words, the alternate of `if()`.

Adding a second condition in the middle of if(), else

Sometimes an *if()*, *else*, is not enough to accommodate all the outcomes. In that case there is an intermediate statement and it takes a condition just like *if()* does.

This intermediate conditional is called *else if()*

Here's the basic syntax:



```
1 if (Boolean true) {
2     do this stuff;
3 } else if (Boolean true) {
4     do this stuff instead;
5 } else {
6     do this when it all fails;
7 }
```

Fig 2

How to physically write the braces in a conditional statement

Notice the opening brace on line 1. This is the JavaScript common way of doing it. If you come from an older language such as C , you may be used to code the opening brace on the second line by itself. That is perfectly fine. Whichever way you choose to use just make sure you don't mix styles.

Also, the closing brace is always at the beginning of a line (example: lines 3, 5 and 7). After the *if()* closing brace we may write the *else* on the same line, or the *else if*, when applicable, and finish the line with an *opening* curly brace (see line 3).

The very final closing brace always stand on its own (line 7). That is common to all languages that use braces. A very popular mistake is to write the closing brace to the right of the last statement. That is very ugly and very difficult to troubleshoot.

Sometimes you will find JavaScript conditionals without braces and it also works. However I do not recommend such practice because it leads to inconsistencies and it may trigger an error sooner or later because the JavaScript interpreter adds semicolons automatically when it thinks they are missing, and the lack of a curly brace may trigger the interpreter to think that it has reached the end of a statement. Until you master this styling concept use jsBeautifier to style your code after you are done writing it.

When to use an else if()

Just like *else*, the *else if()* does not exist without first having an *if()*. We can only have one if() and one optional else, but we can have as many else if()s as we wish.

Question:

What is the difference between coding *else if()*s or just writing a new *if()* statement for a new subsequent condition?

Answer: It is a matter of semantics. When we write several *if()*s, we are creating *independent* statements. This means that JavaScript interpreter will evaluate all of them and output all those that are *true*. On the other hand, when we have *else if()*s after an *if()*, we are telling the JavaScript interpreter that we want just one output since both conditions are tied together. In other words, when the *if()* condition is true, output it and do not execute the next *else if()* condition because we are done here.

- **Side note:** When two conditions could possibly be true for both *if()* and *else if()*, try to write the most common condition first, in order to avoid an incorrect output.

Here's what I mean: If we are probing for numbers that are divisible by 3 or divisible by 5, **or both**, make sure the divisibility 3 && 5 condition goes before the divisibility by 3 or by 5.

Let's practice some more:

- 1- Use your editor to create this code, then paste it to the Console.
Declare a variable *x* and assign it the number 15.
Then create a condition that prints to the screen (console.log) one of the following:
"yes, x is divisible by 3"
"yes, x is divisible by 5"
"yes, x is divisible by 3 and 5"
"No, x is not divisible by 3 nor 5"

Remember to use the *modulus* in order to see if *x* is divisible by a certain number, like for example:

if (x % 3 === 0) Which means that if the remainder is zero, *x* is divisible by 3, right?

See the next page for a script discussion and a link to my own file.

Is Programming an Art or a Science?

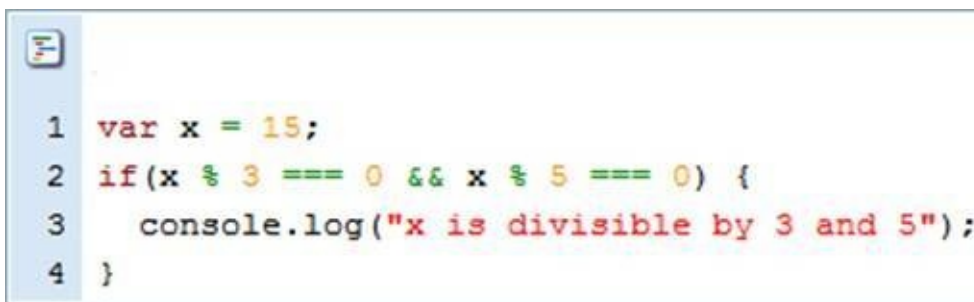
There are many ways to finish the task. Programming is part science and part art. In a way it is like planning and cooking a meal.

First we need to look at possible solutions. In this project we probably need three conditional statements, which means that we may have to employ some *else if()*s in the middle before we get to the *catch all else* solution at the end. It's a good idea to take a walk outside while putting all the ingredients together in our mind for the recipe we are about to create. Sketching the ingredients on a piece of paper also helps a bit (real paper and pencil).

As mentioned before, we have to address the common condition first, which is when *x is divisible by both 3 and 5*. That should become our main *if()* condition. Then we create two additional *else if()*s for numbers that are divisible by 3 or by 5.

Finally, we assign the output “No, *x is not divisible by 3 nor 5*” to the catcher *else*.

2- Let's do the first condition and test:



```
1 var x = 15;
2 if(x % 3 === 0 && x % 5 === 0) {
3     console.log("x is divisible by 3 and 5");
4 }
```

Fig 3 (See original file: [here](https://bit.ly/1uoc1N8)) bit.ly/1uoc1N8

On **line 1** I have declared variable *x* with the value of 15.

On **line 2** the **&&** logical operator is used to ask JavaScript if both left and right operands are *true*. Notice how each operand has to be explicit. Many new programmers tend to write it this way: *x % 3 && 5 === 0*, JavaScript will not understand this syntax. An individual Boolean condition for each case is necessary.

We could wrap each operand in parentheses to make it easier in the human eye:

```
if ( (x % 3 === 0) && (x % 5 === 0) ) {
```

If you do it this way, be sure to close all parentheses. There is an even number of them.

On **line 3**, I used `console.log` to print the appropriate message.

3- Test your script to look for errors and to make sure you get the correct output. If you wish, reassign *x* to another number, like for example, 14, and run the conditional statement again. You can run it by using the **Up Arrow key** until you get to the script, edit the variable *x* and press ENTER. When *x* is 14 you will not get any output which means that JavaScript jumped over the code block because the condition was no longer *true*. In that case you will see an *undefined* message on the Console. That is ok, it is just an automatic echo from the Console and it does not

have any influence on your script.

Note: Subdividing our code project into *little test steps* is a good practice. There is nothing worse than spending an hour coding a script only to realize in the end that it does not work.

4- Now we are ready to create our first *else if()* statement.

Which one of these conditions should we do first:

the one that checks for divisibility by 3,

or the one checking divisibility by 5?

At this point it does not matter since we have resolved the ambiguity of being divisible by both, but I would do it in numerical sequence writing the 3 condition first. See lines 4 through 6 on the image below:



```
1 var x = 15;
2 if(x % 3 === 0 && x % 5 === 0) {
3   console.log("x is divisible by 3 and 5");
4 } else if(x % 3 === 0) {
5   console.log("x is divisible by 3");
6 }
```

Fig 4 (See original file: [here](https://bit.ly/1uoc1N8)) bit.ly/1uoc1N8

On **line 4** notice how *else if* is coded to the right of the brace that closes the previous *if()*. This is not mandatory but it looks more professional.


On **line 6** the curly brace closes the *else if()* statement.

5- Paste your script onto the Console and press ENTER to run it. You should still get the first output because variable *x* is still 15 which makes the first condition *true*.

In order to test the second condition, reassign variable *x* to **9**. To change the value of variable *x* press the *Up Arrow key* until you see the whole script, then place your cursor at the variable declaration, change the value to 9 and press ENTER.

Now we should get the second output: '*x is divisible by 3*' because 9 is divisible by 3.

6- It's time to write the final *else if* and also the catcher *else* at the very end. See the next image:



```
1 var x = 15;
2 if (x % 3 === 0 && x % 5 === 0) {
3     console.log("x is divisible by 3 and 5");
4 } else if (x % 3 === 0) {
5     console.log("x is divisible by 3");
6 } else if (x % 5 === 0) {
7     console.log("x is divisible by 5");
8 } else {
9     console.log("No, x is not divisible by 3 nor 5");
10 }
```

Fig 5 (See original file: [here](https://bit.ly/1uoc1N8)) bit.ly/1uoc1N8

On **line 6** we can see the second *else if()* condition.

On **line 8**, *else* finally catches anything that is not divisible by 3 or divisible by 5. No condition is necessary since this is the “all else” optional outcome. JavaScript will throw an error if we ever write a condition for *else*.

On **line 10** lies the last closing brace which closes the *else* statement.

Don't you feel like you are getting there? I hope so. *Conditional branching* is an important part of creating real programs.

We will see how to code this script without using *else if()* in a bit, just to illustrate another important concept: nested conditionals. For now, let's just take a break by doing something else.

2.3 The anatomy of a function

On the last exercise we kept pressing the *Up Arrow* to recall the program in order to test it. What if we didn't have an Up Arrow? We would have to write the script again and again, correct?

This is where functions come handy. *Stop!* Don't think of math when I mention functions in JavaScript. Think of *functionality*. In JavaScript functions are containers of program routines. We could have saved our `if()`, *else* script in a function and just recall it by *calling* the function. That's one word, versus 10 lines of text.

Preliminary things to know about a function

- A function is a closed container like the blue rectangle in the picture seen below.
- To feed data into a function in order to be processed, we need one or more input parameters.
- To *export* data from a function in order to reuse it, we need to implement an interface called return, *or* alternatively assign some outer variable to the new result.

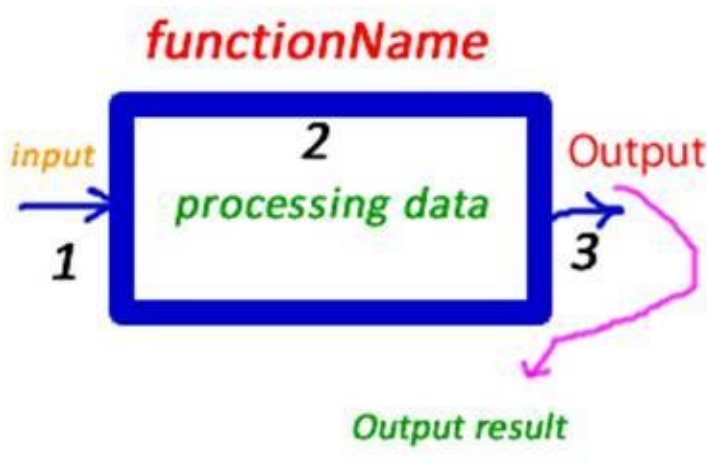


Fig 6

Scripting a function

A representation of a function goes like this:

```
function funcName() { stuff to do; }
```

The above function does not take any input data, it just does stuff.

To create a function that takes an input parameter and processes it, we need to include the parameter inside of the parentheses:

```
function funcName(inputParameter) { stuff to do; }
```

When we say to “*call* a function” it means to have the function execute its *magic*. All we have to do to call a function, is to write the function’s name and attach a pair of parentheses as a postfix :

```
funcName () ;
```

Or, if the function has an input *parameter* we must include an *argument* that will replace the parameter:

```
funcName (123) ;
```

when 123 is a number.

or

```
funcName (x) ;
```

when x is a variable.

or

```
funcName ("Tony") ;
```

When Tony is a string of characters.

Here’s a function arbitrarily named *sizeMeUp* that converts a string value into uppercase:

```
function sizeMeUp(x) {console.log(x.toUpperCase()) ; }
```

Here, x is the input parameter that gets replaced by the data we pass in when we call the function.

- 1- Calling the function in order to up case “red white blue”:

```
sizeMeUp("red white blue") ;
```

It displays *RED WHITE BLUE*.

Let's go back to our previous *conditional* exercise...

Remember how we had to use the Up Arrow in order to repeat the code for another test run?

We could make things easier by inserting our conditional exercise inside of a function because it saves us from having to retype the script every time we need to test it.

All we have to do is to *call* the function by its name:

```
functionName();
```

Shall we redo it?

Here's the previous script placed inside of a function named test1:

[link](https://bit.ly/1wSjW9Z) (bit.ly/1wSjW9Z).

- 1- Copy the code from the link and paste it on your Console.

(To copy the code from the forum post, click on the two-page icon at the top right side of the code).

After pasting the code on the console, call the function and press ENTER:

```
test1();
```

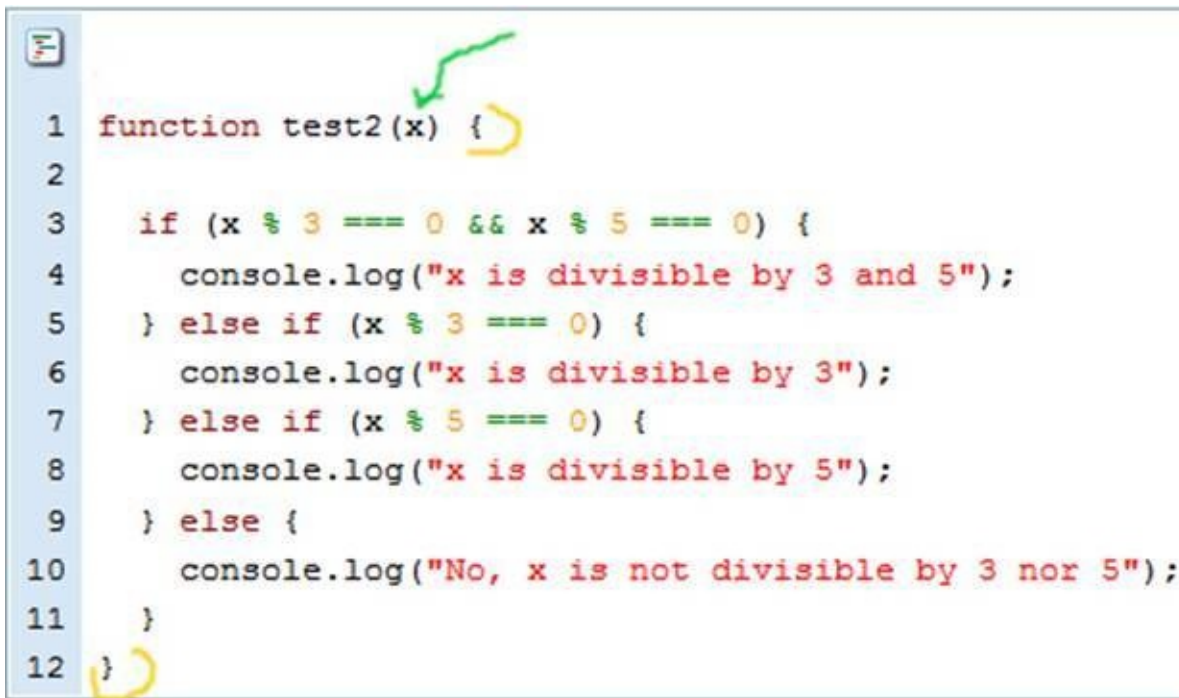
- 2- Call it again, and again until you realize that the output is always the same because variable *x* never changes. Use the *Up Arrow* to recall the function or just *call* the function manually by writing the function call each time.
Don't forget to include the parentheses ().

Another cool feature of a function is on the following explanation:

We could create a temporary variable that could be modified by some *argument data* we pass into the function when we call the function.

Don't worry about understanding everything right now. We will get there. Just go along with me and try the next step.

- 3- Let's modify the function by removing the permanent *x* variable declaration on the top (before the function script), and placing *x* as an input parameter. Also change the name of the function to *test2*. See the next image for more details:



```
1 function test2(x) {  
2  
3   if (x % 3 === 0 && x % 5 === 0) {  
4     console.log("x is divisible by 3 and 5");  
5   } else if (x % 3 === 0) {  
6     console.log("x is divisible by 3");  
7   } else if (x % 5 === 0) {  
8     console.log("x is divisible by 5");  
9   } else {  
10    console.log("No, x is not divisible by 3 nor 5");  
11  }  
12 }
```

Fig 7 (See original file: [here](https://bit.ly/1wAbQjv)) bit.ly/1wAbQjv

Now all we have to do is to call the function and *pass in* a number so that we can test it:

- 4- Call function *test2* and see the results for each one of these numbers: 3,5,15,21:

```
test2(3);  
test2(5);  
test2(15);  
test2(21);  
test2(11);  
etc.
```

The input parameter(s) of a function act as a temporary **internal** variable. This is really useful because we can send data into a function to be processed. That's what happened during our *test2()* function *calls*.

Once we master the basic syntax of a function we start to create some real cool stuff.

Functions may still be confusing for you. I will cover all the basics as we code along. It is not a difficult subject but it needs to be approached in the proper sequence.

What is a function call?

A picture is worth a gazillion words:

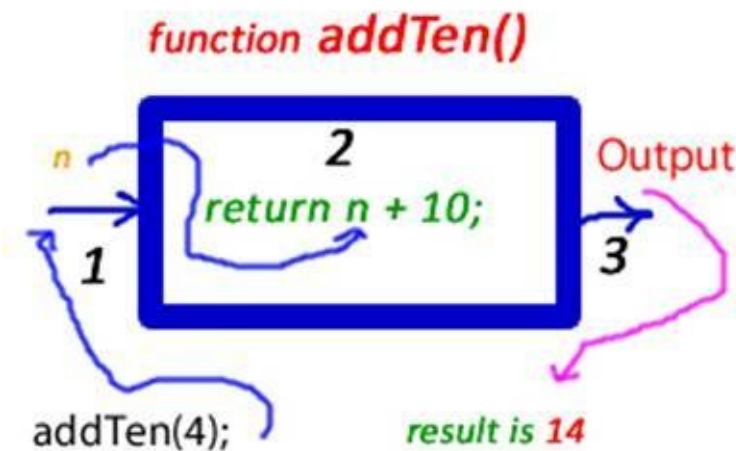


Fig 8 (See original file: [here](https://bit.ly/1raw6pT).) bit.ly/1raw6pT

Looking at figure 8, on the bottom left side we are calling function `addTen()` and passing an argument data of 4 to the function parameter `n`.

Then 4 will become the temporary value for parameter `n`, and the sum of $10 + 4$ is executed.

Finally, the **return** mechanism exports the result which is the numeric value of 14.

We could have used a `console.log(n + 10);` but the `console.log` would display the value on the screen without exporting it out of the function. In other words, we would not be able to use the number 14 outside of the function because `console.log()` does not export data, it actually displays everything as a string value on the screen, even if it looks like a number.

So, **return** is a very important mechanism in functions. Actually it is native to functions. We cannot use the command `return` outside of functions. JavaScript will not allow it.

What is the purpose of return?

The purpose of return is twofold:

- To export data from inside of a function in its raw form.
- To stop and end the function call. This is important. The return mechanism acts as a break and ends the act of calling a function. Anything written in the function below the return statement will not be processed because JavaScript exits the function as it returns. After the function returns, the function call is wiped out from the working memory. You don't have to be concerned with this at this moment but it is good to know that function calls don't linger in memory forever. They free up space by getting out of the way. This could be a third reason to include your code inside of a function: memory efficiency.

What happens to the data when a function returns?

If we want to reuse the data for further processing, we need to catch the *returned* data. Otherwise it just gets lost in space.

How do we catch the data being returned from a function?

One way to catch it is to assign the returned value to a variable. In other words, assign the function call to a variable.

In the example of figure 8 (see the link below the image), instead of just calling the function:

```
addTen ( 4 ) ;
```

We could assign it to some variable:

```
var y = addTen(4) ;
```

Now *y* holds the value 14. The function call gets wiped out of memory but the value is retained by variable *y*.

- ***In summary***, data coming from a function needs to be *returned* and assigned to a variable if we need to use it further down the script. When we output data from a function via a `console.log()`, `alert()`, `prompt()` or `confirm()`, we are just displaying it, not retaining it.

2.4 Lab work 6

Ready for more lab work?

Create a program to convert Fahrenheit to Celsius

Operator [precedence](#) determines the order in which operators are evaluated. JavaScript precedence is close to math precedence:

- Parentheses are evaluated before multiplication or division, and multiplication or division are evaluated before subtraction or addition. All being equal, operands are evaluated from left to right and from inside of parentheses out.
...nothing to memorize, just practice a lot.

The following is the formula to convert Fahrenheit to Celsius:

Deduct 32 from the variable F,
then multiply the result by 5,
then divide the new result by 9.

We need to include the subtraction within parentheses so that it is done first. Then write the multiplication next so that it is done before the final division which should be placed on its right side. The formula looks like this:

$$(f - 32) * 5 / 9$$

That was the hard part. Now let's code.

- 1- Create a function named **f2c** that takes 1 input parameter called **f**.
- 2- Inside of the function send the **result of the formula** to the screen.
In other words use **console.log()** to print to screen. This is because we only want to display it, not to reuse it. Notice how the input parameter's name **f** coincides with the variable given to our formula. That was done on purpose so the JavaScript interpreter knows where to plug the input number we are going to give it when we call the function.
- 3- Call the function and pass in an argument of 212 degrees Fahrenheit. (write just the number by itself). Your displayed answer should be 100.

See my own solution here (**first script**): [f2c](#) | bit.ly/1Dp2SKk

Create a program to convert Celsius to Fahrenheit

The formula to covert Celsius to Fahrenheit is

Multiply c by 9,
then divide the result by 5,
then add 32 to the new result.

There is no need to insert parentheses, just make sure the multiplication comes first:

$c * 9 / 5 + 32$

- 1- Create a function named **c2f** that takes 1 parameter called c .
- 2- Inside of the function send the result of the formula to the screen.
In other words use **console.log()** to print to screen.
- 3- Call the function and pass in an argument of 100 degrees Celsius (just the number).
Your answer should be 212.

See my own solution here (second script): [c2f](https://bit.ly/1Dp2SKk) | bit.ly/1Dp2SKk

Why are we using console.log() instead of return?

As you may remember, `console.log()` was designed for quick testing outputs. At this time we are not concerned with returning a value for further processing. When we need to reuse a value we will use **return** and assign the value to a variable.

However, please read the next topic for information about the return mechanism.

In JavaScript a function always returns something

In case you are wondering whether the *function call* gets cleared out of memory when we don't explicitly use *return*, the answer is *yes*, in JavaScript it does.

In JavaScript every function returns automatically when the code interpreter reaches the closing brace, if not before.

When that happens, JavaScript will return the value of *undefined* if a return statement was not manually written.

In production you will not visually see this automatic return but on the test Console you will see *undefined* in the end if you didn't ask for a specific return. If your Console is still open, look at the last function call. You will see the term *undefined* below the number 212.

This automatic return assures proper memory management. Memory management is actually done by the browser, not by JavaScript. Different browsers have different [garbage collection](#) mechanisms.

A program to convert Fahrenheit to/from Celsius

In this project we are going to introduce a function with two input parameters. The first input parameter will take a numeric value, and the second input parameter will serve as a condition to toggle between Celsius and Fahrenheit.

How will we toggle between Celsius and Fahrenheit?

We can use an *if()*, *else if()*, *else* statement:

- a) If, the user enters an *f* for Fahrenheit, the *if()* output will convert Fahrenheit to Celsius.
- b) Else if, the user enters a *c*, the number gets converted from Celsius to Fahrenheit.
- c) Else, the script will output something like
"Sorry, that conversion is not supported".

When we create a function with multiple input parameters like for example...

```
function test(x, y) { return x - y; }
```

...we need to *call the function* and *pass in* the data arguments in the proper sequence.

The value to be assigned to *x* (which is the leftmost input parameter) should be written in the function call as the first data argument. On the following function call, 10 is grabbed by parameter *x* and 3 is grabbed by parameter *y*:

```
test(10, 3);
```

Notice how we separate them by commas. Had I entered 3 first, 3 would become *x* and the output result would be incorrect.

Let's start:

- 1- Create a function named **conversion** that takes two input parameters named *num* (reminds me of number), and *degrees*. Notice the comma separating the input parameters:

```
function conversion(num, degrees) { }
```

- 2- Inside of the function create your first *if()* statement that says
`if(degrees === "f") {`
and then it console.logs the formula to convert Fahrenheit to Celsius.
The formula is **(f - 32) * 5 / 9** However, please read the note below:

NOTE: you must replace the f in the formula with our new generic input parameter **num**.

Bonus: add a **+** **"c"** just before the closing parentheses of your console.log(). This is so that the output will look like *100c*. Do the same on your next step for "f".

- 3- Next create the *else if()* statement that says

```
else if(degrees === "c") {  
and then it console.logs the formula to convert Celsius to Fahrenheit.  
The formula is  $c * 9 / 5 + 32$ 
```

NOTE: you must replace the *c* in the formula with our new generic input parameter ***num***.

- 4- Finally, create a catch all *else* statement that console.logs *"Sorry, that conversion is not supported"*.
- 5- Close the function with a curly brace **}**.
- 6- Paste your code onto the Console and test it with the following function calls:
`conversion(212, "f");` <- should result in 100c
`conversion(100, "c");` <- should result in 212c
`conversion(300, "b");` <- Should result in "Sorry, that conversion is not supported".
Remember, the character input is a string and it should be in quotes.

(See my own solution here (***third*** script): [conversion](https://bit.ly/1Dp2SKk) | bit.ly/1Dp2SKk)

Congratulations!

Don't stop here!! Please take a look at the next page for more practice ideas.

You need to start creating other programs of your own.

Below please find some suggestions to create other conversions:

a) Perimeter

Suggested function design: calcPerimeter(length,width) {

Formula: length + length + width + width

b) kilometers into miles

Suggested function design: kiloMiles(km) {

Formula: km * 0.6214

c) feet to meters

Suggested function design: feetM(ft) {

Formula: ft * 0.3048.

Go for it : spend a few hours creating new stuff.

If you would like to save your scripts, paste them onto a plain text editor such as Windows NOTEPAD and save them with the extension of .txt which represents the plain text format.

We will cover other ways of storing executable JavaScript programs later.

END OF LAB

2.5 Filling out a form to request a loop

A loop is a *subroutine* that repeats a sequence of programming instructions over and over until it gets a *halt* command.

We have seen how to create a function to store code that runs every time we call the function.

We can also store code in a loop that repeats itself for a determined number of times. Then if we want to reuse the loop at will, we can insert the loop into a function and call the function later at any time to run the loop without having to re-write another loop script.

- Beware of infinite loops.

Always make sure you design a way to stop the loop. That should be your first decision when creating a loop: **when and how should this loop end?** JavaScript will only end it when it runs out of memory. We must instruct JavaScript to end it much earlier than that!

- If you ever run into trouble while testing, click on the universal key sequence **CTRL** **c** to stop the program. Most of the times it will stop it.

How long do we want our loop to repeat itself? Just once, 5 times, 100 times, or until a certain condition happens like for example when the value of a certain variable becomes something other than the original value.

All this sounds so abstract, right? Let's remove the *abstractness* out of the equation.

What no one tells you about loops

We don't really create a loop ourselves. This mechanism already exists in the JavaScript library located in your browser and it exists in several different styles (for loops, while loops, do while loops, etc.).

What we have to do is to fill in a request form for a certain *loop style*, and then provide instructions to JavaScript for what we want it to accomplish while the loop runs.

Yes, there are several styles of loops in the JavaScript library. They all loop around but how they loop can make a difference in your choice of *loop style* selection.

One common style is the ***for loop***. The *for loop* repeats itself for a determinate number of times.

Then we have the ***while loop***. A *while loop* repeats itself while something remains true. If that something is always true, the while loop will run forever. Sometimes we want a loop to run forever, like when the computer scans the keyboard waiting for a human to press a key. If the human does not press a key any time soon, the while loop will keep scanning. Don't worry about memory because this type of scanning is not using much memory, it is just looking around like a night watchman.

JavaScript also has in its library a loop style known as ***do while***. A *do while* loop is slightly different than the regular *while*. The *do* part makes it run once whether the condition is true or false.

Why does a *do while* loop run at least once? Because the *while* condition part of the loop only shows up at the end of the script. So JavaScript will run the loop and then at the end checks the condition to see if it should run a second time. It is like saying "*shoot first and ask questions later*".

- Remember: in order to use a loop we must fill in a loop style application.

We will cover all these styles of loops in due time. For now let's talk about the *for loop*.

Don't burn your brains out trying to memorize something you don't yet need. This was just a brief introduction. Now let's get our hands dirty by creating loop projects.

2.6 The for loop

We are going to learn how to fill in the *form request* for a loop in the style of a “for loop” because it will be used on our next lab project.

The idea of a *for loop* is to repeat itself for a specified amount of times.

The basic form declaration has a **header** and a **body** as shown on the following [pseudo](#) code:

for (x starting at zero; and for x less than 5; increment x) { execute this code }

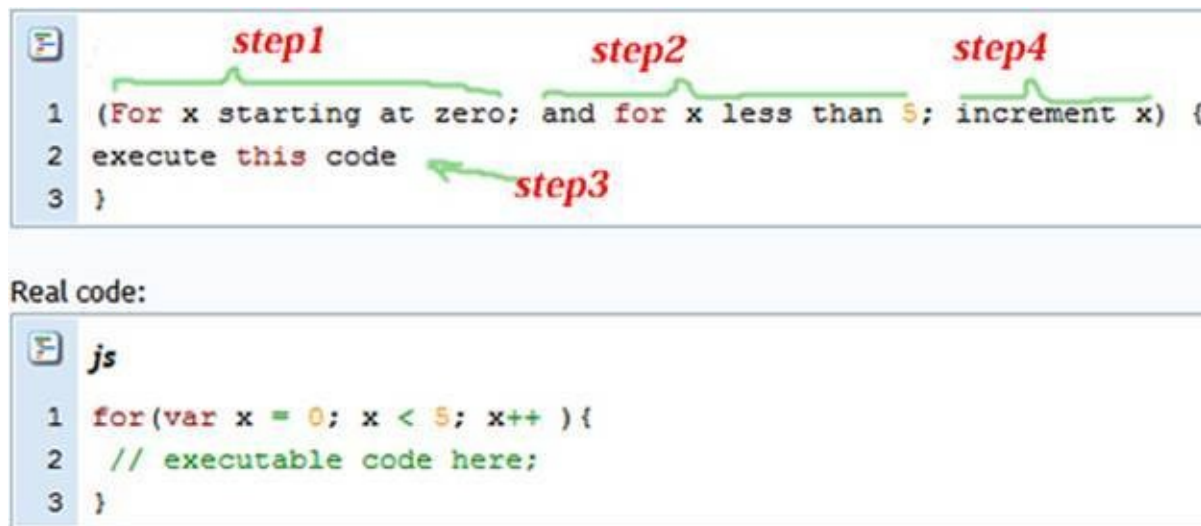


Fig 9

We have three instructions to be evaluated in this form, hence the semicolon separating on the first two instructions (to make them independent of each other).

for x starting at zero;

Here JavaScript stores *x* as a temporary variable with the value of zero. The name *x* is unimportant, most programmers use the name *i* instead. The correct script is `for(var x = 0; or for (var i = 0; etc...`

and for x less than 5;

At this point JavaScript evaluates this condition. Since *x* is < 5 because it is zero, the condition is *true* and JavaScript executes the code in the body once. The correct expression is:

`x < 5;`

increment x ++

After JavaScript runs the code block the first time (step3), it returns to the *for loop* declaration and it increments the value of *x* (step4). Now *x* is 1 instead of zero. The correct increment script is:

`x++ or x = x + 1`

Since `x++` is the last expression to be evaluated, the semicolon is not necessary. Actually it is undesirable since it will trigger an error, preventing JavaScript from running the code block because it thinks it is at the end of the *for loop*. The first semicolon after `x++` should

be at the end of the first executable statement inside of the body code block.

Once JavaScript runs the loop the first time, it returns to the form for a second evaluation. The second evaluation will be as follows:

(For x starting at one; and for x less than 5; increment x) { execute this code }.

On the second iteration x is 1 but still less than 5 and therefore the code block is executed again, and x is incremented again at the end. And so on until x becomes 5.

Let's create a simple for loop to see how it works.

The most common name for the temporary variable to be used as x is *i*. Perhaps the reason is because it reminds us of the word *index*. So let's use the classic name *i* instead of x:

```
for(var i = 0; i < 5; i++)
```

That is the header form which means:

for this declared variable i which starts with the value of zero; as long as i is less than five; increment i by 1 after each loop cycle.

On line 2 we include the executable instructions in the code block. These instructions will be repeated 5 times by JavaScript:

The following image illustrates the concept:

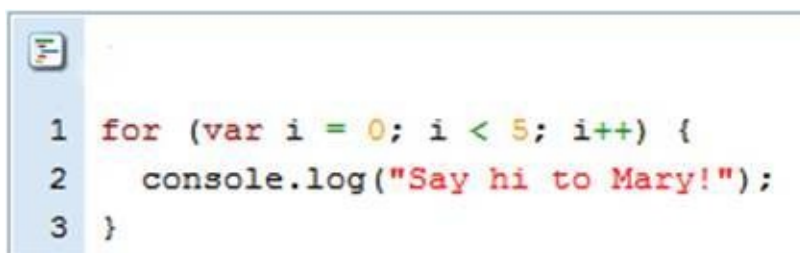


Fig 10

- Please note: If you code this loop and paste it onto your console you may have a one line result with a prefix of (5) telling you that the Console ran it 5 times. This is normal behavior of Consoles. They don't usually repeat the same output more than once in order to save display real estate, they just tell you how many times the loop ran with it, but in real life you would get five lines displayed. However, if the print is different for each time, then the Console displays all the 5 iterations separately.

To see a real display of a loop at work, replace `console.log("Say hi to Mary!");` with **`console.log(i) ;`**

Now it prints the value of variable *i* in each of the loop repetitions and since the output is different each time, the Console must display each one of the iterations.

If you want a more presentable output you can try this one:

```
console.log("the value of i is now " + i);
```

And you should get:

the value of i is now 0

the value of i is now 1

the value of i is now 2

the value of i is now 3

the value of i is now 4

It displays 0 through 4 because the count of *i* started at 0 and the fifth time is 4. This is ok, it still loops 5 times. However, if your intention is to display 1 through 5, then make *i* equals 1 from the start, instead of from zero.

2.7 Looping over a string of characters

Do you still remember *bracket* notation and *dot* notation?

I introduced it on Part One, in the chapter *Manipulating variable data*.

Let's review and expand a bit more about the topic.

I'm going to declare variable `x` with "orange" as the string value:

```
var x = "orange";
```

The value "orange" is a collection of sequential characters and each character has a location number starting at zero, like `o` is zero, `r` is one, `a` is two, `n` is three, etc.

We can use *bracket notation* to find the character data in any location of the string:

`x[location Number]`; \leftarrow it gives us the value inside of the location.

Example:

```
x[2];
```

It will display character "`a`" which is the third character and position 2 (counting from zero)

I can't use *dot notation* here because *dot notation* does not accept numbers or symbols:

`x.2` \leftarrow this will give me an error.

What can we do with this bracket notation syntax?

Bracket notation allows us to address a specific character on the string value.

Here's an example of displaying character `g` from variable `x` by sending `x[4]` via a `console.log`:

```
console.log(x[4]);
```

Here's another example: characters `ge`:

```
console.log(x[4] + x[5]);
```

So with *bracket notation* we can probe into a string of characters and fetch the contents of a location number.

What about dot notation?

Dot notation is good to grab properties of string values, such as `length`, remember?

```
x.length;
```

or

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

- Dot syntax cannot be used with numbers, symbols, and names which values are not yet known when the code is being written. This prevents Dot syntax from being used in loops since the value in a loop will change for each loop cycle. So **when it comes to dynamic data, only Bracket syntax can be used.**

2.8 Lab work 7

Searching for the position of a string character

In this project we will have variable `x` assigned to `"orange"`.

The idea is to search for character `"g"` and display its location on the screen.

- 1- Declare variable `x` and assign it the string `"orange"`.

```
var x = "orange";
```

- 2- Fill in a *for* loop form,
starting with `i = 0;`
and the range of `i < x.length;`

- *orange* has 6 characters so `x.length` will be 6 but we should not use number 6; we should program *generically* as much as we can because most of the times we don't know the length of the value to be worked on.

- 3- In the body of the *for* loop create an *if()* condition that probes for the existence of character `"g"`. The Boolean expression should look like this:

```
if(x[i] === "g") {
```

- In other words: if the character at current position `i` is `g`...
- 4- If the condition is *true*, print the position number of `"g"` to the screen.
 - 5- Close the *if()* condition and then close the loop.
 - 6- Test your code. It should display the number 4 as a result.

Extra practice:

- a) Replace `"g"` on the *if()* statement with another character not contained in `"orange"`, and create an *else* statement to catch it with the following output:
`"That character was not found!"`

(See discussion and result on the next page).

(Answer) Searching for the position of a character

See my own solution(s) here:

[forum](https://bit.ly/XScVak) | bit.ly/XScVak

- 1- Declare variable *x* and assign it the string “orange”.

```
var x = "orange";
```

- 2- Fill in a *for loop* form

starting with *i* = 0;

and the range of *i* < *x.length*;

```
for(var i = 0; i < x.length; i++){
```

- 3- In the body of the *for loop* create an *if()* condition that probes for the existence of character “g”.

```
if(x[i] === "g"){
```

- 4- If the condition is *true*, print the position number of “g” to the screen.

```
console.log(i);
```

- 5- Close the *if()* condition and then close the loop.

```
}  
}
```

If something is not clear, please read the previous topic ***Looping through a string of characters***.

The last output, `console.log(i);` displays the position number of the character we are looking for.

What is the purpose of this exercise?

There are many applications that use these concepts. Right now I am preparing you to understand the next topic which will be about *array lists*. By the time you get there you will not have to struggle with concepts such as these.

For practicing purposes please do the next exercise on your own.

Counting numbers and declaring their odd/even quality

- 1- Create a for loop that counts from 0 to 10
- 2- For each count display their odd/even quality like for example:
1 is an odd number, 2 is an even number, etc.

See my own solution here (sample number 3):

[odd/even](https://bit.ly/XScVak) | bit.ly/XScVak

Counting apples

- 1- Create a for loop that counts up to 3 apples. Zero is not allowed and the output must say 1 apple if it's single apple, then 2 apples, and finally 3 apples.

Instead of using *i* as a counter variable I am going to use a variable named *apples*. Please feel free to use whatever you want. The goal for this exercise is to practice more *for* loop constructs and to think on how would we distinguish between a single output and a plural output.

See my own solution here on exercise *number 4*:

[apples](https://bit.ly/XScVak)) `bit.ly/XScVak`

END OF LAB

2.9 Array lists

Strings versus arrays

Up to now we have been working mostly with *string* values. Here's an example of a string value assigned to variable *x* which has already been initialized by *var* :

```
x = "orange";
```

As far character mapping is concerned, *o* is the first character located in position *zero*, *r* is the second character located in position 1, *a* is the third character located in position 2, etc.

Here's another example:

```
y = "orange banana apple";
```

In the above example, *b* is the eighth character (we must count the white spaces) and it's located in position number 7 (0,1,2,3,4,5,6,7).

What if we wanted to count the words instead of counting the number of characters?

We can't separate the words with individual quotes because JavaScript will throw an error:

```
y = "orange" "banana" "apple"; <- this would not work.
```

And even if we separate them with commas,

```
y = "orange","banana","apple";
```

JavaScript will first assign *y* to *orange*, then reassigns *y* to *banana*, and finally to *apple* which, in the end, will be the only value stored in *y*.

So in string values, mapping is **limited to individual characters**. The whole string is considered one unit of multiple characters.

- ***In short***, if we want to save words as separately independent units on a variable, we need to use a different *data type* because *string* will not do it. One of such types is called an ***array*** (there are others).

The difference between an *array* type and a *string* type is that we can store individual words. Another advantage of an array is that we can also store values of type *number*, as well as other types such as *functions*, *objects* and *Booleans* in individual packets of the same variable.

Just like a string value, in arrays we also address each position numerically starting from location zero. In fact, a string value is a very simple array of individual characters.

Think of an array as a pocket folder.

Each pocket is sequentially labeled and in each pocket, we can store anything we want.

To retrieve the contents of each pocket we call the array with bracket syntax and use the location number to address the data we are trying to fetch.

Array syntax

To declare a value of type *array* we use a pair of square brackets:

```
var x = [ ];
```

You may read on the web about [bracket syntax](#) as being referred to as [array syntax](#) and the reason is that bracket syntax comes from the idea of array pocket locations.

The above example is an empty array because it does not have any data inside of the brackets, but the square brackets have instructed the JavaScript interpreter to declare variable *y* as a *type array*.

Going back to our last string example, if we want to save individually independent words we can declare and assign variable *y* in the following manner:

```
var y = ["orange","banana","apple"];
```

Now if we call *y*'s location number 2, we get "apple" as a result:

```
y[2]; ← it displays "apple"
```

or more implicitly:

```
console.log(y[2]); ← it displays apple
```

Knowing that our array *y* has three values spanning from location zero to location 2, we can add more values to it by addressing subsequent locations. Example:

```
y[3] = "peach";
```

Now if we call variable *y* on the Console by typing *y*; we get:

```
["orange", "banana", "apple", "peach"]
```

What happens if we reassign one of the existing locations such as for example location zero?

```
y[0] = "grape";
```

We overwrite the previous value because we have just reassigned the location to a different value. The array will not shift positions to accommodate one more items, it just deletes the existing value by reassigning the location to something else.

Array *y* now still contains the previous number of values as follows:

```
["grape", "banana", "apple", "peach"]  
"orange" was deleted.
```

Using push() and unshift() to insert elements into an array

As arrays get larger it becomes almost impossible to know how many items the array contains which makes it very difficult to add new items since when we add an item to a position in the array and this position is already filled, JavaScript replaces the new item with the old item. The old item is deleted from the array.

To prevent this deletion from happening, JavaScript has two mechanisms that help us add items to an array: *push* and *unshift*.

Push()

The most common way of adding new items to an array is by adding them to the very end of the list.

- The array method *push()* adds items to the very end of the list.

Do you remember how we appended characters to a string by using `+=` ?

The problem with `+=` is that it adds the new item to the last item as one unit (by attachment).

For example, in

```
var y =["orange", "banana", "apple", "peach"] ;
```

If we write `y += "cherry" ;`

we will get *"orange, banana, apple, peachcherry"*

and that's not what we want to accomplish in an array, right?

We want independent items.

- The correct syntax to add items to the end of an array is as follows:

```
y.push("cherry") ;
```

We call the array method `push()` with *dot syntax* and pass in "cherry" as a data argument. It works just like calling a function with an argument data.

Now the array contains one more element:

```
["orange", "banana", "apple", "peach", "cherry"]
```

Let's add one more item, *avocado*:

```
y.push("avocado") ;
```

Now array `y` contains the following items:

```
["orange", "banana", "apple", "peach", "cherry", "avocado"]
```

The meaning of push:

Imagine a stack of bricks. When we place a new brick on the stack it pushes down the others because of its weight. The top of the stack is the last position of the array. We push

it in order to insert a new item into the stack.

Now, if you visualize the array in a horizontal manner instead of vertically, the last item is on your right side.

We will use `push()` many times throughout the book since it is the most common way to insert new array items.

unshift()

The method *unshift()* adds items to the beginning of the array.

When we visualize the array vertically the beginning of the array is the very bottom of it, and horizontally, it is the left most side of the array.

Taking the last instance of array *y* from the previous topic as an example, let's add *blueberry* to the beginning of the array:

```
y.unshift("blueberry") ;
```

Array *y* is now:

```
["blueberry", "orange", "banana", "apple", "peach", "cherry", "avocado"]
```

Notice how "orange" moved to position 1, or second item, to give way to "blueberry" as the new position zero.

The meaning of unshift:

In order to insert a new brick at the bottom of the stack, we need to unshift all the bricks up so that we can squeeze a new brick as the first item of the stack since the first item is at the bottom.

On the other hand, when we do a push to insert a new item to the top of the stack, we are actually shifting everything down as we push, even though there is nowhere to go at the bottom. (Shift is actually used to remove an item from the bottom as we will see later).

When we remove the item the bricks get shifted down but they are not being pushed because we are not inserting a new item at the top of the stack. This is just to help you understand the terms and where they came from. Don't worry about *shift* for now. I will cover it later.

Right now we know about *push()* and *unshift()*.

Printing array items with a for loop

As you've probably noticed, the visual display of an array on the screen when we call it by name is in its raw visual format:

```
["blueberry", "orange", "banana", "apple", "peach", "cherry", "avocado"]
```

How do we display the items by themselves?

We can use the array *length* property, and a *for loop*, to do such thing.

The *length* property works just like it did for strings. We can find out the length of an array by attaching *.length* to the array variable name:

y.length; <— it display the number 7

Now, as you remember from playing with the string “orange” earlier (Lab work 7), when we know the length of the item we are going to scan, we initialize a **for loop** in the following manner:

```
for(var i = 0; i < 7; i++) { do this stuff; }
```

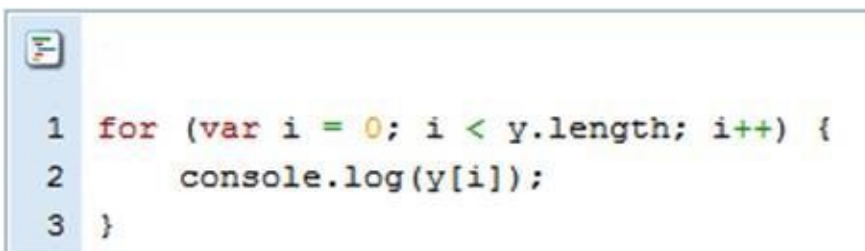
However, for the most part we don't know the length of the item we are trying to scan and that's when *.length* comes to the rescue:

```
for (var i = 0; i < y.length; i++) { do this stuff; }
```

This last implementation is much more efficient because we don't have to worry about redoing the loop if the array ever changes length when we delete or add new items to it.

The rest of the code needed to manipulate the array will go where the “do this stuff” code block is. There we could print, search, or create many other possible implementations. Also and as you know, the code block is usually written on separate lines.

Here's a sample loop to print all items to the screen:



```
1 for (var i = 0; i < y.length; i++) {  
2   console.log(y[i]);  
3 }
```

Fig 11

On **line 2** the `console.log()` is addressing each array location. We accomplish this by assigning *i* as the number for the location. In this way, as the loop counts the *i* numbers, it changes the position of the `y[i]` and each items is displayed accordingly (and in separate lines). Let's try it next.

2.10 Lab work 8

Let's do some programming to cement our understanding of the array creation, addition of items, and traversing the array with a *for* loop.

Please fire up your JavaScript Console and let's get our hands dirty!!

- Try doing this exercises yourself by following the next 7 steps, or jump to the guided exercise that follows in order to do it as I explain the concepts.

NOTE: These seven steps are not a project. Each step is an independent exercise to see how each implementation is done.

Preliminary project

- 1- Create an empty array assigned to a variable named *misc* (short for miscellaneous).
- 2- Create another array assigned to a variable named *greekLetters*, and with the following items: "alpha", "beta", "gamma", "delta"
- 3- Assign the number **99** to the first position in array *misc*, assign the word "**bottles**" to the second position in array *misc*, and assign the word "**beers**" to the ninth position in array *misc*.
- 4- Print to screen the length of array **greekLetters**.
- 5- Print to screen the length of array *misc*.

Using **push** and **unshift**

- 6- Add a new item "**epsilon**" to the end of array *greekLetters*.
- 7- Add a new item "**folk song**" to the beginning of array *misc*.

(See my answers on the next page)

(Guided exercise) Preliminary project

- 1- Create an empty array assigned to a variable named *misc*
(short for miscellaneous)
`var misc = [] ;`
- 2- Create another array assigned to a variable named *greekLetters*, and with the following items: “alpha”, “beta”, “gamma”, “delta”
`var greekLetters = ["alpha", "beta", "gamma", "delta"] ;`
- 3- Assign the number **99** to the first position in array *misc*,
assign the word “**bottles**” to the second position in array *misc*,
and assign the word “**beers**” to the ninth position in array *misc*.
`misc[0] = 99 ;` (no quotes for number)
`misc[1] = "bottles" ;`
`misc[8] = "beers" ;`
- 4- Print to screen the length of array *greekLetters*.
`console.log(greekLetters.length) ;` ← results in 4
- 5- Print to screen the length of array *misc*.
`console.log(misc.length) ;` ← results in 9

Did you notice how we were able to add a ninth position to the array *misc*?

Did you also notice how the array *misc* now has 9 items, which means that 6 items are reserved with the placeholder value of *undefined*?

Using push and unshift

- 6- Add a new item “epsilon” to the end of array *greekLetters*.
`greekLetters.push("epsilon") ;`
- 7- Add a new item “folk song” to the beginning of array *misc*.
`misc.unshift("folk song") ;`

Practicing is the only way to get good at it.

Let’s practice with *for* loops on *arrays* by wrapping functionality in *functions* and use some *if else* conditionals in the mix. These projects will cover important concepts that can be used often.

Creating a method to add items into an array

Ready for a simple challenge?

We are going to create a function that automatically adds an item into the array when the function is called. This will allow us to practice with *array.length* as an alternate solution for *push*.

- Use your [editor](#) first before pasting onto the Console. Try doing this exercise by yourself first by following the next 5 steps, or jump to the guided exercise in order to do it as I explain the concepts:
 - 1- Declare an empty array assigned to the variable *x*.
 - 2- Create a function named ***addMe*** that automatically adds an item to array *x*:
 - a) Use one input parameter called **item**.
 - b) In the function body assign the input *item* to the first empty position in the array, based on *x.length*.

x.length is always one more number than the address of the last item. By assigning an item to location [*x.length*] it will always match the next position available in the array. Give it some thought and then look at my answer if you are unsure on how to do it.
 - c) Also inside of the function, create a *console.log* to print out the whole array in its raw display form (no loops yet).
 - 3- Paste the script in the Console and Call the function by passing in “**grape**” as an input item.
 - 4- Then Call the function again and pass in “**apple**” as an input item.
 - 5- Do the same for “**banana**”.

(See the answer on the next page).

(Guided exercise) **Create a method to add items into an array**

- 1- Declare an empty array assigned to the variable x.

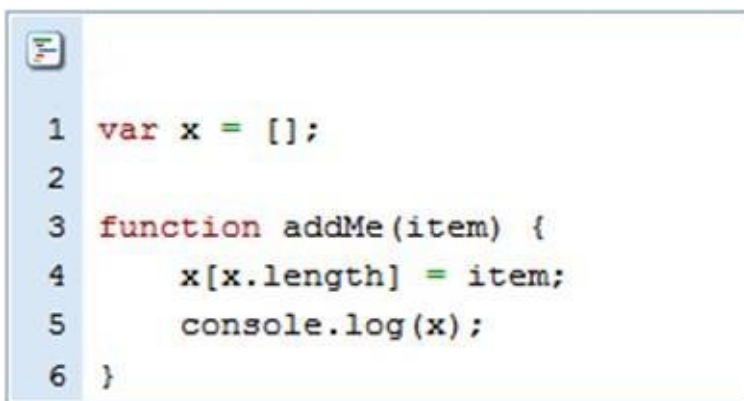
```
var x = [];
```

- 2- Create a function named ***addMe*** that automatically adds an item to array x.

- d) Use one input parameter called **item**.

- e) In the function body assign the input *item* to the first empty position in the array, based on *x.length*.

- f) Also inside of the function create a *console.log* to print out the whole array in its raw display form (no loops yet).



```
1 var x = [];  
2  
3 function addMe(item) {  
4     x[x.length] = item;  
5     console.log(x);  
6 }
```

Fig 12

On line 1 the array x was declared.

On line 3 a function *addMe* with *item* as an input parameter was declared.

On line 4 whatever data is *passed in* when we call the function, the data gets assigned to the maximum length of array x. As you know, the length of an array always shows the number of items in existence. It so happens that the number of items is always 1 more than the last location of the last item. Therefore, the length of the array always corresponds to the next position after the last existing location. So we can use this number to calculate the next location available. This is actually what *push()* would do automatically but here we are doing it manually in order to understand a bit more about arrays.

On line 5 we print the array x to screen in its new raw form.

- 3- Paste your script on the Console and
Call the function by passing in “**grape**” as an input item.
Then Call the function and pass in “**apple**” as an input item.
Do the same for “**banana**”.

```
addMe ("grape") ;  
addMe ("apple") ;  
addMe ("banana") ;
```

The final result is: ["grape", "apple", "banana"].

Great!

Whenever you're ready, please move on to the next page. We are going to create a method that prints out the items of an array, any array. Coding generically is an important goal as programmers.

Create a generic method to print out the contents of an array

Generic is synonymous with portable. In programming scripts should be made as portable as possible. It leads to reusability and memory efficiency. By creating a portable array method we can use it with any array, not just the one we are programming at the moment.

- 1- Create a method (a method is a function mechanism) named *printArray* that takes an input parameter arbitrarily named *inputArray*
- 2- Then inside of the function create a ***for loop*** that initializes *i* to zero and runs as long as *i* is less than the length of the array given to *inputArray*.
- 3- In the body of the loop, `console.log` the `inputArray[i]`, this will correspond to each item location of the given array, one at a time.
- 4- Close the loop, close the function. You're done.

Testing...

- 5- If you still have array *y* on the console, print array *y*. If not, add array *y* as shown below:

```
var y = ["grape", "apple", "banana"];
```
- 6- Print array *y* by calling the function ***printArray()*** and pass *y* as an argument.
- 7- Create several other arrays and use the generic `printArray()` method to print them.

(See my answer and a link to file on the next page).

(Guided exercise)

Create a generic method to print out the contents of an array

- 1- Create a method (a method is a function mechanism) named ***printArray*** that takes an *input parameter* arbitrarily named ***inputArray***.

```
function printArray(inputArray) {
```

- 2- Then, inside of the function create a *for loop* that initializes *i* to zero; and runs as long as *i* is less than the length of the array given to inputArray.

```
for(var i = 0; i< inputArray.length; i++){
```

- 3- In the body of the loop, console.log ***inputArray[i]***, this will correspond to each item of the given array, one at a time.

```
console.log(inputArray[i]);
```

- 4- Close the loop, close the function. You're done.

```
}  
}
```

- 5- If you still have array *y* on the console, print array *y*. If not, add array *y* as shown below and print it:

```
var y = ["grape", "apple", "banana"];
```

- 6- Print array *y* by calling the function ***printArray()*** and pass *y* as an argument.

```
printArray(y);
```

- 7- Create several other arrays and use the generic ***printArray()*** method to print them. See my file: [forum](http://bit.ly/1rdSRuA). (bit.ly/1rdSRuA).

Discussion

Generic methods are very useful. It is always a good idea to save them for future use. However, since you are practicing, saving code for a future copy/paste might not be as beneficial to you yet. Code snippets act like recipes for cooking. There are many cookbooks for programmers out there and although they may not help you at this moment, they may come handy in the future. (Personally I don't own any of those books since I enjoy creating my own stuff but in a professional environment they can be useful, especially when it comes to more advanced, time consuming implementations).

The inputArray parameter serves as a placeholder for whatever *array name* you pass in when you call the function printArray. Instead of printing the array to screen with a console.log() we could have done something else with the array, like for example sorting it. We will get there when I introduce more properties and array methods from the internal JavaScript library.

Remember, this book is about learning how to use JavaScript, it is not a dictionary of methods. You can always refer to my other [eBook](#) for that purpose, but only after you

finish this one.

Extra bonus: Create a generic method to add items into an array

To do this exercise, go back to the first exercise of this lab work (on Fig 12), and recreate it in a way that we could add items to any array, not just the array `x` as the original example shows.

- 1- You need to add two input parameters to the function I'm calling ***addMe2***:
 - a) One input for the array name
(I will call it ***inputArray***),
 - b) and one input for the item you want to add to the array
(I will call it ***inputItem***).

- 2- Then *in* the function body, create a script that assigns the location ***inputArray.length*** to the new item (***inputItem***):

```
inputArray[inputArray.length] = inputItem;
```

Note: Since the script gets a bit more complicated than the original one, I am using names that make sense to humans in order to help them understand what the script does. Even if we code for our own use (as opposed to other people), we will not remember what it is when we return tomorrow.

- 3- If you wish, add a `console.log(inputArray);` to the function body in order to monitor the output.
- 4- Finally, create some empty arrays outside of the function, and then add items to each one of them by calling the function.

Whenever you're ready, please refer to my own script:

[forum board](https://bit.ly/1pnR5S8). | bit.ly/1pnR5S8

END OF LAB

2.11 Other useful arrays methods

sort()

In order to sort the contents of an array we can use the JavaScript library method *sort()*:

Using the following array,

```
var y = ["grape", "apple", "banana"];
```

We can sort it like this:

```
y.sort();
```

Or, for displaying purposes:

```
console.log(y.sort());
```

- Please keep in mind that sorting is permanent, it affects the original array.

When it comes to strings, a *string type* value does not have a method to sort because sorting needs to use individual elements and a string is just one element all together. If you ever want to sort a string you will have to convert it to an array, then sort it, and then reconvert to string again.

We will actual do this exercise in a bit.

join()

The method *join* outputs an array value as a string format:

Examples:

```
var y = ["grape", "apple", "banana"];  
y.join();
```

It outputs: "grape,apple,banana"

We can also use a parameter to determine what separates each word.

Here are a few examples of separation methods:

- Adding a pair of quotes as the parameter but without a space between quotes:

```
y.join("");
```

It outputs: "grapeapplebanana"

- Adding a space in between quotes:

```
y.join(" ");
```

It outputs: "grape apple banana"

- Adding a comma and a space:

```
y.join(", ");
```

It outputs: "grape, apple, banana"

- Adding a dash in between the quotes:

```
y.join("/");
```

It outputs: "grape/apple/banana"

- Adding a space, dash, space:

```
y.join(" / ");
```

It outputs: "grape / apple / banana"

Notice that I have been writing "*it outputs*". In order to convert an array value into a string value we must catch the output by assigning it to a variable. If we assign it to the original variable we convert the array into a string. When we assign it to a different variable, we *copy* the values into a string format. The original array remains intact.

What about sorting before converting to string?

```
y.sort().join(" ");
```

It converts to a sorted array, and then it outputs a string:
“apple banana grape”

Notice the **dot syntax** at work for multiple attached methods.

- **Important:** Although *join()* does not affect the original array unless we assign the output to the original variable name, *sort()* works differently. When we sort an array, it physically rearranges the locations of each item. Sorting is permanent and it does not have to be explicitly assigned.

Example:

```
var y = ["grape", "apple", "banana"];
```

```
y[0];
```

 ← It displays “grape”

Now we sort it (without an explicit assignment):

```
y.sort();
```

```
y[0];
```

 ← It displays “apple”

Sorting is permanent.

indexOf()

- The `indexOf()` method returns the location position of an item.
- If the item does not exist, it returns a -1.

This is a very useful method to check if a certain item already exists in an array before we add it, or to search for a specific item as we will see on the next lab work.

In the array

```
var y = ["grape", "apple", "banana"];  
y.indexOf("banana"); ← returns 2 (for location 2)  
y.indexOf("chestnut"); ← returns -1 ( as in 'not found')
```

Example:

In

```
var y = ["grape", "apple", "banana"];
```

We could see if *chestnut* exists and code a script to dialog with the user:

```
if(y.indexOf("chestnut") < 0) {  
  console.log( "Not found, would you like to add it?");  
}
```

It outputs:

"Not found, would you like to add it?"

Then we would write a script to add *chestnut* to the array as we have done in a previous lab work.

- The < 0 bit is to check if the *indexOf* is -1, which means that the item does not exist.
- On the other hand, an *index* of zero or above, means that the item is already in the array.
- Each *index* corresponds to an item location.

pop() and shift()

The opposite of *push()*, which is the method used to add an item to the top or rightmost place in an array, is called **pop()**.

Think of “Pop! Goes the Weasel”!

pop() removes one item from the very end of the array.

In

```
var y = ["grape", "apple", "banana"];
```

```
y.pop();
```

Will remove “banana”. Now *y* is only ["grape", "apple"]

shift() is the opposite of *unshift()*.

We *unshift* all items to insert an item at the bottom or leftmost place.

When we remove an item from the bottom or leftmost place, all the other items get shifted down.

Let’s remove “grape” which is currently the first item in array *y*:

```
y.shift();
```

Notice how *pop()* and *shift()* don’t take any argument in their parentheses. That’s because they specifically remove the last or the first item, not a named item.

Do you still remember how to put the items back?

```
y.push("banana");
```

```
y.unshift("grape");
```

Now array *y* is back to its original state:

```
["grape", "apple", "banana"]
```

slice()

The method `slice()` extracts specific items from one array and outputs those items in the same format type: array.

This means that, if we catch the output by assigning it to another variable, we create another array.

What is the difference between `join()` and `slice()`?

- `join()` converts the output into a *string* value. We use the parentheses to determine how the string words will be separated.
- `slice()` converts the output from an array into another array. Since we don't need to determine how to separate the items because they are already separated, we take advantage of the parentheses to *pass in* some other parameters as follows:

`myArray.slice(0,4);` <- zero is the first location to be included in the output, and 4 is the first location to be excluded.

Example:

In

```
var fruits = ["apples", "bananas", "blueberries", "grapes", "avocado", "peaches"];
```

I am going to declare a new array and use `slice()` to extract some fruit items from the `fruits` array:

```
var pickedFruits = fruits.slice(0,4);
```

Array `pickedFruits` now contains:

```
["apples", "bananas", "blueberries", "grapes"]
```

- The two parameters (0,4) have the following meaning:
 - The first parameter represents the first location to be extracted.
 - The second parameter represents the first **location** to be excluded from extraction.

A (0,1) would extract just *apples* because it excluded bananas, which is in position 1.

A (1,4) would extract *bananas*, *blueberries*, *grapes*, because it extracts from position 1 (location one) which is *bananas*, and it excludes from position 4 which is *avocado*.

For more advanced techniques on slice please refer to my eBook [JavaScript Objects Functions and Arrays Explained](#).

splice()

The Array method splice() is a very interesting one. Do you still remember *push()* and *unshift()*? They add items to the end or beginning of an array, right?

- **splice() adds items to a specific location in the array.**

Let's see how it works.

Depending on how many arguments you use in the parenthesis, you can do the following:

a) **Delete items** mode:

When you use only **two arguments** you are giving the **index positions** in the following manner:

First argument: where the action is to start (inclusively),

Second argument: How many items will be removed.

In a way, **having just two arguments** means to delete because there is nothing to add.

splice(0,3) (explanation further down the page)

b) **Add items** mode,

If you insert **more than two arguments**, the **third** and consecutive arguments are **data to be inserted**, and in that case JavaScript assumes you want to insert items into the array.

splice(2,0, "yellow")

In just add mode, the second argument is *zero*, which means that no items will be deleted in exchange for the items we are adding with the third and consecutive parameters.

or...

c) **Replace items** mode.

If the second parameter in **add mode** is anything other than zero, this number will indicate how many items will be deleted in exchange for the items being added.

splice(2,1, "yellow")

For example, **splice(0,3)** means to delete 3 items: the first, the second and the third. Number 3 represents the quantity of items to be removed. The number 3 coincides with the third item on the array but it is not necessarily so. If we had splice(2,3) it would remove the third, fourth and fifth items, which correspond to positions 2,3,4 and counting from zero.

- The 3rd and consecutive arguments represent the items to be inserted.

For example,

splice(2,0, "yellow") means to insert *yellow* in position two (third item) and *unshift* the others because the *zero* means that no items are to be deleted.

`splice(2,1, "yellow")` it means to add *yellow* to position 2 (third item) and remove whatever is there (this is a replacement due to the second argument being anything other than zero).

See the next image for a better view of these concepts.



Fig 13

- Please note that, unlike other implementations where we were displaying items, with splice() we are actually editing the original array. It is a permanent operation just like `sort()`.

We could also use `splice()` to remove and transfer some items into a new array (sort of what we did with `slice()`, except that now we are also editing the original array)

Here's an example using the following array:

```
var x =["blue","red","green","violet","brown"];
```

- a) Splitting the array by copying the first three items into **newArray** before deleting them:

```
var newArray = x.splice(0,3);
```

- b) Call array x:

x;

It returns ["violet", "brown"]

c) Call array newArray:

newArray;

It returns ["blue", "red", "green"]

We have split the contents of *x* into two different arrays. Had we used *slice()* instead of *splice()* we would still have the original array *x* intact.

In summary, *splice()* is very versatile and it can be used in place of *push()*, *pop()*, *unshift()* and *shift()* whenever we need to address a specific position in the array instead of generically using the end or the beginning of the array to insert or delete items.

If at this time your head is spinning with so much information know that you're not alone. You don't really have to memorize these methods. What you really need to do is to understand how they work. Then later when you need to use them you can always come back and review the concepts one more time. Eventually they will become part of your internal toolset. Just make sure you understand how they work.

Let's practice a bit more to make sure you get the idea!

Splice lab work

Let's try a few samples to illustrate the concept of splice().

Deleting items with splice():

- 1- On your Console declare the following array:

```
var x = ["blue", "red", "green", "violet", "brown"];
```

- 2- Delete 3 items starting at positions 0 which is *blue*.

Exclude from deletion position 3, which *violet*:

```
x.splice(0,3);
```

We have just **deleted** *blue*, *red*, *green* from array *x*.

- 3- Call in the array:

```
x;
```

It returns ["violet", "brown"] because the original items in positions 0,1,2 were deleted.

Adding items with splice():

- 4- Let's add our items back into the array, starting at position zero and without deleting any items in exchange:

```
x.splice(0,0, "blue", "red", "green");
```

- 5- Call the array:

```
x;
```

returns ["blue", "red", "green", "violet", "brown"].

We started at position zero and added all the listed items, and deleted zero items.

You may not use splice() extensively on your programming but it is a great tool to understand and call upon it when needed.

There are many more methods we can use with arrays. Learning all of them at once is just overwhelming. It may be a good idea to acquire my other [book](#) about JavaScript library methods for reference purposes and for an easy explanation on how they all work. If I thought it would help to include all the other methods here at once, I would have done it. Of course you can always search and hope not to get confused or sidetracked by some bogus explanation on the internet. Knowing these tools is an important step to become a master in the craft.

2.12 Lab work 9

NOTE: If you want to do this exercise along with me jump the next 14 steps.
Reserve this page for future practice.

- 1- Take the following array **y** as an example and sort the array:
`var y = ["grape", "apple", "banana"];`
- 2- Now assign to a new array **mySort**, the contents of the sorted array **y**.
- 3- Test **mySort** by displaying its contents on the Console.
- 4- Now, display the array **mySort** in a *string* type format and split the items with 1 blank space. Hint: use `join()`.
- 5- Now, instead of just displaying it, assign the items from **mySort** to a new variable **z**, but in a string format.
Use a combination of *space, forward slash, and space* as a separator.
- 6- Call **z**.
- 7- Use the *indexOf()* method to find the position of "banana" in array **y**.

- 8- Test for the existence of “banana” in array *y* (use an if conditional statement), and display the following message on the screen using `console.log()`:
“*banana is contained in array y*”
- 9- Using the same method as the last exercise, look for “*chestnut*“, then display two messages, one for *if()* and another for *else* as follows:
“*chestnut is contained in array y*“
else: “*chestnut is not found in array y*“.
- 10- Remove the *last* item of array ***mySort*** using *pop()*:
- 11- Remove the *first* item of array ***mySort*** using *shift()*:
- 12- The array *mySort* should now only contain “banana”.
Add “chestnut” as the last item.
Use *push()*.
mySort should now have 2 items, [“banana”, “chestnut”]

13- Using **splice()**, insert “apple” in array **mySort** at position 1 (as a second item). Do not delete any existing item.

mySort should now contain [“banana”, “apple”, “chestnut”]

14- Using **slice()**, create a new array named **newArray** with a copy of the second and third items (“apple” and “chestnut”) from *mySort* array.

The *newArray* should now contain [“apple”, “chestnut”].

(See my results on the next page.)

Answers to Lab work 9:

- 1- Take the following array *y* as an example and sort the array:

```
var y = ["grape", "apple", "banana"];  
y.sort();
```

It returns ["apple", "banana", "grape"]

- 2- Assign a new array named *mySort* to the contents of a sorted array *y*:

```
var mySort = y.sort();
```

- 3- Test *mySort*:

```
mySort;
```

It returns ["apple", "banana", "grape"]

- 4- Now display the array *mySort* is a string type format, and split the items with 1 blank space. Hint: use `join()`:

```
mySort.join(" ");
```

or you might have done this which is also correct:

```
console.log(mySort.join(" "));
```

It returns "apple banana grape" in a string format.

- 5- Now, instead of just displaying it, assign the items from *mySort* to a new variable *z*, but in a string format. Use a combination of space, forward slash, and space as a separator:

```
var z = mySort.join(" / ");
```

- 6- Call *z*:

```
z;
```

It returns "apple / banana / grape"

- 7- Use the *indexOf()* method to find the position of "banana" in array *y*:

```
y.indexOf("banana");
```

It returns 1, which means that "banana" exists and is located on position 1 (second item).

- 8- Test for the existence of “banana” in array y (use an *if* conditional statement), and display the following message on the screen using console.log():

“banana is contained in array y”:

```
if(y.indexOf("banana") >=0) {  
    console.log("banana is contained in array y");  
}
```

It returns “banana is contained in array y”

- 9- Using the same method as the last exercise, look for “chestnut”. Then display two messages, one for *if()* and another for *else* as follows:

“chestnut is contained in array y”

else: “chestnut is not found in array y”:

```
if(y.indexOf("chestnut") >=0) {  
    console.log("chestnut is contained in array y");  
} else {  
    console.log("chestnut is not found in array y");  
}
```

It returns “chestnut is not found in array y”

- 10- Remove the *last* item of array **mySort** using *pop()*:

```
mySort.pop();
```

It removes “grape”

- 11- Remove the *first* item of array **mySort** using *shift()*:

```
mySort.shift();
```

It removed “apple”

- 12- *mySort* should now only contain “banana”.

Add “chestnut” as the last item, using *push()*:

```
mySort.push("chestnut");
```

mySort should now have 2 items, [“banana”, “chestnut”]

13- Using **splice()**, insert “apple” in array **mySort** at position 1 (as a second item) and do not delete any existing item.

```
mySort.splice(1,0,"apple") ;
```

mySort should now contain [“banana”, “apple”, “chestnut”]

14- Using **slice()**, create a new array named **newArray** with a copy of the second and third items (“apple” and “chestnut”) from *mySort* array:

```
var newArray = mySort.slice(1,3) ;
```

The *newArray* now contains [“apple”, “chestnut”].

The argument 1 represents the second location from *mySort*, and argument 3 represents the exclusion which starts at location 3.

- Yes, location 3 does not exist, but we had to cover location 2 in order to include *chestnut*. The exclusion started at location3, which is item 4.

It may be overwhelming at first.

Practice, practice, practice.

END OF LAB

2.13 Lab work 10

Building a mechanism to fetch a variable from an array

This exercise will review variable assignments, and creating arrays with variables as elements, plus function declarations, reassignment of variables and the usage of indexOf.

The idea is to call a function and select a language as an argument. Then, based on the language selected, the correct sentence will be displayed. Does it sound interesting?

Example:

When calling `show(english) ;`

This sentence gets displayed:

"The quick brown fox jumped over the lazy dog".

- Notice how *english* is not wrapped in quotes (and it is in lower case). That tells us that it must be a variable name, not a string.

I have used [Google Translate](#) to convert my sentences from English to Portuguese and Spanish (and later French). My apologies to native speakers if it sounds funny, it is well intended.

Link to the convenient [jsbeautify](#).

- *Please do the exercise on your own.*
A link to my own file will be given in the end.
Interim files will also be available so that you can check your work.

1- Declare three variables named
english, portuguese, spanish.

Respectively, assign to each one of them the following sentences:

(Link to convenient copy/paste: [raw file](#) | bit.ly/1mpqCsr)

"The quick brown fox jumped over the lazy dog"

"A ligeira raposa marrom saltou sobre o cão preguiçoso"

"El rápido zorro marrón saltó sobre el perro perezoso"

- 2- Declare an array named ***langVersion*** with the following elements:
english, portuguese, spanish.

Remember, since these names are already declared as variables, we should not write them in quotes, just separate them with commas.

The purpose of this array is to save each variable in a sequence. Couldn't we just use the variables directly and forget about the array? Yes, we could. However, in a large program those variables will be saved along with many others and by creating an array we are listing the variables related to language translation, which can be further manipulated, edited, erased or even become part of a larger group of languages. Of course there are other ways to do this but right now we are practicing with arrays and it is a very important practice.

- 3- Declare one more variable named ***sentence*** with the numeric value of 0.

This variable will be used to store the location number of the language we are going to select from all the languages in the array.

The reason why we are giving this variable a value of zero is to tell JavaScript ahead of time that this variable will be numeric. In reality JavaScript will change the type of value if needs be, but it is a good practice to declare our intention, even if this intention may only help a human inspecting the code, rather than helping JavaScript to process it.

(Link to convenient copy/paste for steps 1,2,3: [raw file](#) | bit.ly/1Dt7fnS)

- 4- Below your current code, create a function named ***show***.

The purpose of this function will be to display the correct sentence based on the language we choose when we call the function. The sentence will be picked from the array of languages.

```
function show() {}
```

- 5- Introduce to the function an input parameter named ***language***.

- 6- In the function's body, line 1, reassign the already declared variable ***sentence*** to the location number of the language selected. We do this by using `indexOf()` to find the location, like this:

```
sentence = langVersion.indexOf(language) ;
```

Later, when we call the function, like for example ***show(portuguese)***, the parameter *language* will be replaced by whatever language argument we give to the function. And the **`indexOf(language)`** will assign variable *sentence* with the correct array address for that language.

- 7- Finally, on the next line and still inside of the function body, write a `console.log` to display the contents of the variable chosen when someone calls the function:

```
console.log(langVersion[sentence]) ;
```


From step 6 we know that variable *sentence* is the location number of one of the items in the array.

- 8- Paste your code on the Console and call the function ***show()*** to test each sentence (Note: if you get an error thank the gods for the opportunity to do some debugging because it is a very good practice):

```
show(english);  
show(portuguese);  
show(spanish);
```

You may see on your screen the term *undefined* after the output sentence. That is normal in test Consoles. It is just telling us that the function *show()* did not explicitly return anything and so, it automatically returns *undefined* because JavaScript functions always return something.

See the final version on the following links:

[forum](#) | [raw text](#).

bit.ly/1r0Ho2D | bit.ly/1mCJogu

Side note:

- If something is not clear, do not move forward until you consciously frame a question. Stop, step back and try to see what each line is attempting to do. Try it 100 times if you need to do so. Then, when you think you know what it is that you don't understand, move forward to see if understanding comes at the end of the assignment when you look at the script as a whole. Do not let it go if you have any doubts. Concepts need to be thought of; we can't fly over them and think that we will get them later.
- On the other hand, understanding comes in layers, but we consciously need to make sure we can continue our path even if we don't fully grasp the idea about something. Ask yourself a question and give yourself an honest answer before moving forward. They say that the act of asking is in itself half of the answer and many times the answer becomes obvious when we properly frame the question.

Adding French to the array

Suddenly someone complains because there is not French version of the Quick brown fox.

- 1- If you still have your Console open, add a variable *french* at the prompt:

```
var french =
```

Don't worry about being all the way at the bottom of the script. We are just testing stuff. Notice how *french* is in small letters.

Variables start with small letters.

- 2- Assign the following line to variable *french*:

“Le renard brun rapide saute par dessus le chien paresseux”

[raw file](#). | bit.ly/1wGvYQS.

- 3- Now, use the method **push()** to insert the variable *french* into the array *langVersion*.

```
langVersion.push(french) ;
```

- 4- Test the script by calling

```
show(french) ;
```

- See final version here:

[forum](#) | [raw text](#).

bit.ly/1r0Ho2D | bit.ly/1mCJogu

Printing out the array contents

Do you still remember how to create a *for loop*? for loops are great to traverse array elements.

In this project we are going to create a *for loop* to *console.log* the items from array *langVersion* just to illustrate a point leading us to our next topic.

If you don't have the previous script on your Console, copy it from the previous page all the way at the bottom, and paste it on the Console. Then proceed to step 1:

- 1- Create a for loop to print the array elements to screen.
I will use the common designation *i* as a variable for the counter, and *langVersion.length* as the limiter for my loop repetition.
 - 2- In the loop body, *console.log* the contents of the array.
This is accomplished by dynamically addressing *langVersion[i]*, which is the current location of each element as the loop repeats itself.
- See my script on the [forum](#) (the **fourth** script shown) | bit.ly/1r0Ho2D
or as a [raw file](#) | bit.ly/1DtOsZu

Did you notice how the variable names are not being printed?

The only thing we get displayed is the *text value* from each variable.

That's because each variable name is instantly **evaluated** by JavaScript and the value to which the variable points to is the one that JavaScript fetches and displays on the screen.

What if we just wanted a list of the variable names as shown on the array?

To get those labels (english, portuguese, etc) we would need to write **console.log(i)** by itself, instead of **console.log(langVersion[i])**. However, that would only print 0,1,2,3. Not the name of the variables, right?

I really would like to display my labels. **How can we list those?**

That will be covered on the next topic. You see, *arrays* work great and are frequently used but they are not always the best [data structure](#) for the job at hand. When it comes to labels such as the names of those variables, an ***unordered list*** may be more appropriate, as opposed to an *ordered list* like an array.

Let's move forward and talk about unordered lists. If they are unordered we don't address them numerically, right?

Let's see how we handle unordered lists. They are really popular and useful.

END OF LAB

2.14 Unordered lists

- We have seen how the numeric [data structure](#) of type *array* stores data in its ‘pockets’. The first item goes on pocket *zero*, the second item on pocket 1, the third item goes on pocket 2, etc.
- The location of the last item of an array is always the length of the array minus 1. Do not memorize this fact, just understand that locations start counting from zero. This is great to know if we ever need to manually add an item to the array. As for automatically adding new items, we may use *push()* to insert them at the end, or *unshift()* to place them at the beginning, or if we want to insert an item in a specific location we can use *splice()*. On the other hand, to delete items we use *pop()* if the item is at the end, *shift()* if at the beginning, and the versatile *splice()* if we want to delete an item from a specific location.
- As for scanning arrays, we traverse the array with a *for loop* in order to print its contents, and we also know that the JavaScript library has many other methods we can use for further manipulating arrays. Some of these methods were covered in this book, and the other more advanced methods may be read on my “[explained](#)” eBook.

In this topic we are going to work with *unordered lists*.

In unordered lists all item locations are labeled with a *string* value.

We call them **unordered lists** because each pocket is literally labeled. This label can be a single character, or a word, or even a number (however if you use numbers as labels, know that they are not sequential and they are treated as characters as opposed to arrays which are always numerically sequential).

In truth, the *unordered list* is unordered because JavaScript *maps* the label to the item. In the end, these are all arrays, even if such characteristic becomes hidden from the programmer.

In an official array we don't have to name the address or pocket label because it is numerically assumed, like for example:

```
var myArray = ["duck", "lettuce", "sapphire"];
```

For *unordered* lists we need to label the location of the item, I mean the pocket where the item is stored. That can be advantageous for certain applications:

```
var x = {"animal": "duck", "vegetable": "lettuce",  
"mineral": "sapphire"};
```

Notice how the assignment to each item is made like: "animal": "duck",

"*animal*" is the label and "*duck*" is the value.

There is a *colon* separating the two, rather than an **=**.

We use the *curly braces* to wrap the list, as opposed to the *square braces* which are reserved to *arrays*.

- **Please note:** The quotes wrapping the label are optional, but not the quotes wrapping the string value, those are mandatory because in our example we are using string values.

More about that later.

Hashes, objects, associative arrays, unordered lists

In JavaScript all these names are synonymous. Each programming language uses a different terminology to represent unordered lists.

Java uses *HashMap*, *HashTable*.

C# uses *HashTable*, *Dictionary*.

Python uses *dict*.

Ruby uses *Hash*.

Object-c uses *NSDictionary*.

C++ uses *std::unordered_map*.

And JavaScript uses object.

Did I just say object? Personally I don't like the name designation because, in JavaScript, almost everything is an object. Arrays are also objects, and so are functions. However, the name of object in JavaScript has become synonymous with *unordered lists*.

Another popular name is ***associative arrays***. Some languages actually have associative arrays. In JavaScript an associative array which is an array with **key-value** pair association, is called an *object*. However the reason why an associative array is often called an object is because objects in the sense of the term are made with associative arrays. As far as JavaScript is concerned, all these names are *unordered lists* and we call them **objects**.

Hopefully this clears up some of the confusion by knowing that **they all mean the same thing**, but please keep in mind that things are not that simple, there is a lot more to this subject than what this book calls for.

- From now on and for simplicity's sake, we are going to call any *key-value pair structure* an **object**.

Creating an object

An object is a collection of values.

These values can be a mix of anything: strings, numbers, Boolean expressions, variables, arrays, functions and (often) other similar objects.

JavaScript is very versatile in that aspect. The size of a list of values can shrink or expand at any time. We will see how to do all these, like creating, editing, adding new items, deleting items, searching and printing items in the next few pages. Most of these techniques have already been introduced to you and you will remember them as we approach each one of them.

Declaring an object and assigning values at the same time:

```
var x = {"animal": "duck", "vegetable": "lettuce",  
"mineral": "sapphire"};
```

What do we have above?

We have a variable named *x* that points to a data structure of type *object*. We know it is an object because of the curly braces wrapping the data: { }.

The variable *x* and the data structure in itself are in separate (and different) parts of the memory. They are independent of each other. This memory independence also happens for *arrays* and *functions*, but there is no separation of variable name and value when we declare a simple *string* or a *numeric* variable; those reside on the same location as the variable name. We will cover a bit more about it later.

So now we have an object assigned to variable x.

Since the *data structure* is divided into *key-value* pairs we can address the values by calling the corresponding *keys*. Think of the key as the label for the value.

Instead of `x[0]`; we now can (and must) do it this way:

```
x["animal"];
```

It displays *"duck"*. By the way, when using bracket notation, the quotes wrapping the label are mandatory. Notice how *"animal"* is wrapped in quotes. Labels are strings values. Now if you ever use a number as a key (and you can do it), you may also wrap it in quotes or you may not. JavaScript allows numbers to be written in both styles on ***bracket notation*** (but numbers are rarely used as labels).

Why should we wrap key labels in double quotes?

Normally, when we create an object we don't really have to wrap the key portion of a property in quotes and you may see it more often that way (no quotes). Example:

```
var example = {animal: "duck", vegetable: "lettuce"};
```

As you see, there are no quotes for *animal* and for *vegetable* in the object declaration.

However, wrapping key labels with **double** quotes is being used more and more in modern technology, thanks to the advent of [JSON](#) which is being applied everywhere. We will cover the basics of JSON on a later chapter.

So if you are beginning to learn object notation, you might as well develop the habit of double quoting keys when you declare an object.

The above example should then look like the following:

```
var example = {"animal": "duck", "vegetable": "lettuce"};
```

Do it as you wish but please keep this concept in mind.

Assigning new key-value pairs to objects

Let's do some lab work...

- 1- On your Console, create a new and empty object:

```
var crazyList = { };
```

Do you still remember how to check the type of a variable with *typeof* (covered on *Lab work 1*)?

- 2- Let's check the *typeof* crazyList:

```
typeof crazyList;
```

It displays “*object*“. We have an empty object.

- 3- Now we add some values by assigning them to key labels:

```
crazyList["flower"] = "daisy";
```

Object crazyList now contains *{flower: "daisy"}*

- 4- Let's add another key-value pair:

```
crazyList["drink"] = "water";
```

crazyList now has two values:

{flower: "daisy", drink: "water"}

As a curiosity, notice how JavaScript is placing the new items to the right of the last item. However, it doesn't really matter where the items are being placed since, unlike arrays, we don't need such information in order to access those items.

5- Now we're going to add some *numeric keys* just to see what happens (not a common implementation):

```
crazyList[2] = "front teeth";
```

crazyList is now holding 3 items:

```
{2: "front teeth", flower: "daisy", drink: "water"}
```

Do you see how the numeric key was placed to the left of all the *string* type keys? Again, this has no meaning to us because it is not really consistent.

6- Let's add two other numeric keys:

```
crazyList["1"] = "flying bird";
```

and

```
crazyList[3] = "is a crowd";
```

Now crazyList contains:

```
{1: "flying bird", 2: "front teeth", 3: "is a crowd", flower: "daisy", drink: "water"}
```

So we see that when the key is numeric, the item is placed on the left side and also in numeric sequence. When the key is a string value, the item is placed after the last rightmost item.

Also notice how I wrapped ["1"] in quotes. That was just to show that, when it comes to numeric keys, they can be wrapped or not. It is up to you.

- The sequential positioning of key-value pairs has no value for us at this moment. It was just mentioned as a curiosity. Please don't try to memorize such characteristic since this is an unordered list and we will pick items by calling their labels at will.

Using dot notation with objects

On the previous example we had used bracket notation to call the labels that correspond to each object property (the value).

We can also use *dot notation* to represent object properties.

This is actually a very popular way to address items of an object. In truth, we need to become very familiar with both styles of writing because, as much as dot notation is popular, sometimes it cannot be used. On the other hand, if you only use bracket notation your code will look different than the norm and someone will ask you why such style of writing. It is worth learning how to write certain expressions in dot notation and others in bracket notation. Practice will tell you which one to use and when.

Here's what you should know about the differences between these two styles of coding:

1. *Dot notation* will not work with numbers.
2. *Dot notation* will not work with *dynamic* variables (those that change dynamically such as a counter representing a key in a key-value pair when traversing an object with a loop).
3. In summary, *dot notation* can only be used when the property name is a valid identifier of a property and as long as this identifier is not numeric.

Think of **dot notation** as **hardwired** (the value must be known when dot notation is used), and **bracket notation** as **soft wired** (the value can dynamically change or be unknown at the time the code is presented to the JavaScript interpreter).

How do we use dot notation to address an item in an object?

Let's repeat our previous examples using *dot notation* instead of *bracket notation*:

- 1- Create a new and empty object:

```
var crazyList = { };
```

- 2- Let's add some values by assigning them to key labels:

```
crazyList.flower = "daisy";
```

Now crazyList contains *{flower: "daisy"}*

Notice how in dot notation a key is written without quotes. This is because *flower* is already in memory as a declared key to a property in object *crazyList*. In bracket notation however, we always include the key string names in quotes, like "flower". In dot notation quotes are not valid.

This *crazyList. "flower"* = would result in an **error**

- 3- Add another key-value pair:

```
crazyList.drink = "water";
```

Now crazyList contains:

{flower: "daisy", drink: "water"}

- 4- Add a numeric key using dot syntax:

```
crazyList.2 = "front teeth";
```

We get the following error as a result:

SyntaxError: Unexpected number

Numbers are not allowed in dot syntax. It needs to be done this way:

```
crazyList[2] = "front teeth";
```

In summary:

We can *declare an empty object* by assigning a pair of curly braces to a variable:

```
var z = { };
```

Then we can add items by providing a *key* and a *value* in either *dot notation* or *bracket notation*:

```
z.rent = 750;
```

```
z["gasBill"] = 300;
```

And now we have *{rent: 750, gasBill: 300}*

Or

We can declare an object and assign items to it all at once:

```
var b = {"color": "red", "fabric": "cotton"};
```

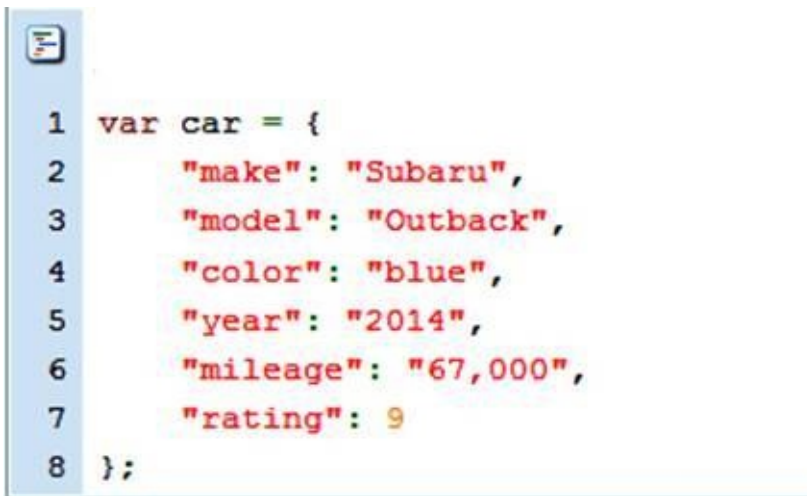
And now we have *{color: "red", fabric: "cotton"}*.

Spanning an object declaration across multiple lines

Up to this point our objects have been short and one liners. For the most part however, objects are more complex and span over several lines of code.

There is no difference between a single line object and a multiple line object. The second style of writing objects is just easier for the eye, but JavaScript does not care one way or the other.

- 1- Here's an example of a multiple line syntax style. Please write the code along with me just for practicing purposes. There is also a link to a raw file below the image:



```
1 var car = {
2     "make": "Subaru",
3     "model": "Outback",
4     "color": "blue",
5     "year": "2014",
6     "mileage": "67,000",
7     "rating": 9
8 };
```

Fig 14 see [raw file](https://bit.ly/ZbwxY3) | bit.ly/ZbwxY3.

Notice the *commas*. Just like in a single line object, all elements are separated by a comma except the last item. If you write a comma there (after the last item) you will get an error because the JavaScript interpreter will expect another element to its right (or in this case below it).

Notice the quotes on numbers **2014** and **67,000**. The reason I added quotes is because I don't plan to do any calculations with these numbers, so I've decided to treat them as strings. This is optional; you could skip the quotes and make them properties of type *number*. As for number **9**, I've decided to keep it as a numeric property in case I want to calculate the average rating for all the cars in the lot. This gives us two possible approaches on how to write number properties.

Also, as explained before, technologies using JSON (JavaScript Object Notation) want the key to be in double quotes so I have placed "make", "model", "color", etc in double quotes. In this way this script becomes machine independent, sort of like when one writes in xml notation. By writing it this way we can send the data to any machine that reads JSON and it will be understood. Quotes in keys are optional. Do it as you wish.

What is a property?

A property is what we call the data elements from an object.

We could call them variables since they are not static (their values can be changed at any time), or just object elements, but in object oriented language they are called *object properties*.

- Object properties are the data contained in an object and they are represented by a name: value pair, also known as a key: value pair.

Think of an object as the owner of its *name: value pair* components. These components are properties of the object, just like my car, bike and scuba equipment are my properties.

```
var tony = {"transportation": "car", "Computer": "desktop pc"}
```

An object may also own some methods.

Let's talk about methods next.

- Don't get wrapped into this terminology trying to memorize what is called what. That will slow you down. Just move along and let your brain assimilate these terms based on practicing, not based on memorization.

What is a method?

A method is a property that executes an action on behalf of the object. If the property is not a “thing” but an action waiting to be called upon, then it is a *method*.

Take for example a desktop computer as an object; we could say that the display, the keyboard and the mouse are some of the properties of the desktop computer. Then we could also say that Firefox and Chrome browsers, as well as Microsoft Word are methods of the desktop computer because these properties have functionality, they don't exist in the physical sense, we need to call them like we call a function, in order to use them. They represent some of the methods of a desktop computer. Their existence is based on the execution of some program routine.

When we mention “execution” we immediately think of functions, but there is a difference between what a function is and what a method is:

- A method may contain one or more functions.

Do you see the difference? A function could be a method of an object, but it is not necessarily a method in itself. A method is a collection of functionality to accomplish a task on behalf of an object. Like a method to print out a list of contents, or a method to update the contents of an object, or a method to sort the contents, or a method to identify the person asking for the contents of an object, and so on.

Let's add a method to our car object.

- 2- If you've been coding along with me, add a new comma to **rating: 9**, because we are about to add one more property, the method **printRating**. (use the [editor](#)).
- 3- Assign to label **printRating** a function that prints on screen the following message:
"The car rates " + car.rating + " out of 10"

```
8
9  "printRating": function() {
10      console.log("The car rates " + car.rating + " out of 10");
11  }
12 };
```

Fig 15 see [raw file](https://bit.ly/1C6I5d5). | bit.ly/1C6I5d5.

- 4- Let's play with this object a bit. Paste the object script from this last raw file (or your own script) on the Console.
- 5- Print the car's rating:

```
car.printRating();
```

Notice how we *call()* a method from an object. It starts like when we call any other property but then we add a pair of parentheses () which creates a function *action*. What would happen if we excluded the parentheses? It would just print the contents of key *printRating* which are the explicit description of the function. That can be handy sometimes when we want to assign the same functionality to another variable, but in order to get an action we need to use the parentheses.

- 6- What about bracket notation? How would we call the method *printRating* using **bracket notation**? Methods are best called with **dot notation** because it is an easier syntax as you will see why next, but it is good to understand how the logic goes and here it is:

```
car["printRating"]();
```

Not as pretty as in dot syntax, right? Notice how the parentheses go outside of the brackets. So this is one area where dot syntax is mostly used. Bracket syntax is reserved more for the **dynamic calling** of a property. We use bracket syntax with for loops because there, each property needs to be accessed dynamically as the loop goes around.

Let's do that!

Looping through an object

Suppose you are aware of an object existence but you don't really know what properties it contains.

You can loop through the object to find its properties.

Introducing the *for in* loop

This loop is written specifically for the object we've been discussing:

See original file here: [raw file](#) | bit.ly/1C6I5d5

The loop goes like this:

```
for(var k in car){  
  console.log(k);  
}
```

The *for in* loop is a variation of the regular *for loop* in the sense that some functionality is already preprogrammed in the JavaScript library. Let's explore it line by line:

for(var k in car) – *k* is a variable introduced by the programmer. Its purpose is to temporarily grab and hold the *key label* for each loop count (the label that matches the location number in each loop cycle).

Let me explain: objects are in a sense just like arrays. The only difference is how the keys are mapped. In arrays, JavaScript maps each location as location 0,1,2, 3, etc. But in unordered lists or objects where keys are being used, JavaScript maps the internal location (whatever location it is) to the key that was originally declared for any given data value.

So, the purpose of variable *k* is to count each location (one per cycle) and at the same time, to grab the key name that corresponds to the given count, so that we can get to the data value assigned to it. It's not *k* that does all these. It is JavaScript that provides such information and stores it in *k*.

As for the name *k*, we could use any name, such as for example *i* but *k* makes more sense because in fact, JavaScript will temporarily store the *key* values on variable *k*, one key per loop iteration. So the letter *k* reminds us of what is being stored there, a temporary location instance. You may also see it on the field or in other texts as *key*, *prop*, or *property*.

(Why am I telling you all this? because I don't want you to start reading posts on the web and think that there are a thousand different kinds of *for in* loops. There is only one *for in* loop and each developer gets creative with temporary variable names).

Going back to our "*for k in car*", in this example we have a dedicated *for in* loop because we know the name of the object, *car*. We could however make this loop *generic* and substitute *car* with another temporary variable to act as a *placeholder*, like for example a

parameter from a function. Then we could insert the real object name as an argument when we call the function, and the function would substitute the *placeholder* with the correct object name. We will try that on another exercise.

`console.log(k)` – This line is used to print all *key names* as they are passed to variable `k`. If however, we wanted to print the values instead of the keys, we need to address each key-value pair dynamically using bracket syntax:

```
console.log(car[k]);
```

JavaScript evaluates `car[k]` and replaces it with the value of the property corresponding to the *dynamic label* instance in `k`.

- Notice the lack of quotes on “`k`”. `k` is a declared variable. Variables never take quotes because that will make them a string and JavaScript will think it is a different identifier other than the one we intend to target. `k` is a variable, “make” or “color” are not variables, they are real keys and a key is a string. Those would be wrapped in quotes if used in bracket notation: `car[“color”]`.

Also, remember that *dot notation* will not work here. This is because the loop is dynamically addressing the key values by mapping `k` to the key, and *dot notation* only accepts the real key name. Remember: *hard wire* versus *soft wire*.

1- So let's paste the object on the Console, then paste the ***for in*** loop just to print the key names

- Here's the *complete* [raw file](https://bit.ly/1rjbSvB) | bit.ly/1rjbSvB.

You should see the following keys displayed:

```
make
model
color
year
mileage
rating
printRating
```

- 2- Replace **(k);** with **(car[k]) ;** on your console log. Remember, you can move up in the Console by click on the *Up Arrow*, then modify the script and press *ENTER*.

Now the printout should look like this:

```
console.log(car[k]) ;
```

And the printout should be:

```
Subaru
Outback
blue
2014
67,000
9
function () {
  console.log("The car rates " + car.rating + " out of 10");
}
```

Of course we could have made it prettier by adding the key, like for example in this way:

```
console.log(k + ": " + car[k]);
```

And that would give us:

```
make: Subaru
model: Outback
color: blue
year: 2014
mileage: 67,000
rating: 9
printRating: function () {
    console.log("The car rates " + car.rating + " out of 10");
}
```

NOTE: I will show how to exclude the *printRating* method from the printout in a future exercise.

P.S.

In case you're wondering about it, I don't really know the rating for an *Outback*, the number 9 is just for practicing purposes only.

2.15 Lab work 11 - redo lab work 10

Do you still remember when in *Lab 10* we displayed “*The quick brown fox jumps over the lazy dog*” in English, Portuguese, Spanish and French?

There were four variables: english, portuguese, spanish and french. Then we added the variables to an array of variable names.

The problem we encountered was attempting to list the array contents as variable names, or as **key-value** pairs, like for example:

english: “The quick brown fox jumps over the lazy dog”.

We could not dynamically display the variable name from the array, just the values themselves. This is because JavaScript treated the variable as an *expression* and gave us the evaluation result, which was the data inside of the variable. (Expressions will be further explained on the next topic).

Well, now we can! All we have to do is, instead of declaring separate variables, we declare them as properties of an object.

Let's do this:

- 1- Create an object named **langVersion**.
- 2- Assign to this object four property keys: english, portuguese, spanish, french.
- 3- For each key add the correct value based on the following text:

[raw file](#) | bit.ly/1rt9dzZ

Do not declare the variables, use property keys instead as seen on the raw file.

- 4- You should now have a complete object.
Next create a **for in loop** outside of the object, to print out the contents of the object.
The display of each property should look something like this:
`english: “The quick brown fox jumped over the lazy dog”;`

- See my own results here:

[forum](#) | bit.ly/1tWQddJ

or the [raw file](#) version | bit.ly/1ofRRkS.

Transforming the for in loop into an object method

5- Once you have finished the project and everything is working, reengineer the *for in* loop so it becomes an internal method of object *langVersion* as follows:

a) Add a new *comma* after the original last property in the object so that you can write another property below it.

b) Give this new property a label named ***list*** so that you can call the method later,

c) And then write a function as the value for *list*.

Inside of the function place your current *for in* loop.

6- Paste your little program on the Console and *call* the method to see how it prints:

```
langVersion.list();
```

- Compare your results with mine here:

[forum](https://bit.ly/1uUYgbd) | bit.ly/1uUYgbd

[raw file](https://bit.ly/1r124B1) | bit.ly/1r124B1

Your listing displays all the properties including the method and its internal script, right?

We can tweak the *list method* to filter out methods from printing as part of the list.

If you want to eliminate the method from the printed list you can use a *conditional statement* to filter out anything that is not of *string* type since `list()` is a function. The logic behind it goes like this: “if the type of property is a ‘string’, print it”.

Do you still remember how to use ***typeof***? We covered it on Lab work 1:

When we write:

```
typeof "tony";
```

JavaScript replies with “*string*”.

7- Reengineer the loop once again so that

```
if (typeof langVersion[k] === "string") {  
  console.log the desired output. No else is needed for this one..
```

- Compare your results with mine here:

[forum](https://bit.ly/1tWSDsR) | bit.ly/1tWSDsR

[raw file](https://bit.ly/1pqbvtZ) | bit.ly/1pqbvtZ

In summary, there are times to use an *array data structure*, and there are times when an *object data structure* is more appropriate. In the end, both arrays and objects are *lists* from the JavaScript library wearing different customs.

In arrays we use a regular *for loop* and with objects we use a *for in loop* which is a for loop preconfigured to print out label names and with no length value declaration necessary.

If you want further practice visit the new forum as I will expand the subject there. Also my other [eBook](#) has many real life exercises to drill and expand this subject.

END OF LAB

PART III: STRENGTHENING THE WARRIOR'S ARSENAL

JavaScript is an instrument that needs to be played.

3.1 Taking an inventory of what we've learned

Give me 5 minutes and I'll tell you everything I know!

Sometimes it feels like that, right?

Well, we have covered quite a lot:

The purpose of **variables**; The type of data assigned to variables; How we introduce a variable name to JavaScript by prefixing it with **var**; How a simple variable data known as a *primitive* value is stored in the same location as its variable name, as opposed to complex variables known as *reference* variables, which are stored on a separate memory location than the variable name that points to them...

We have seen how a *data structure* such as an **array** acts as a numerically indexed list of items, and how to use the **for loop** to traverse (scan) the array and manipulate each one of its elements based on the *array.length* property.

We have used **push()** and **unshift()** to insert new items into an array, and **pop()** and **shift()** to remove items from the array.

We have also experimented with **splice()** to remove, replace or add new items into any specific location of an array, and how we can copy a whole section of the array into another array by using **slice()**.

What about *sorting* arrays with **sort()**? And how to use **join()** to convert an array into a string.

Last but not least, we have seen how **indexOf()** can be useful to find the location number of a certain item, or to check if the item has already been inserted in the array.

So much we've learned in such a short time!

What about objects?

We know that almost everything in JavaScript is an object of some sort, but normally we refer to objects when we talk about **unordered lists**, those lists indexed by key labels.

The items in an object, which are called *properties*, can be addressed by either *dot* syntax or by *bracket* syntax. Some of the properties are called *methods* because they represent functionality and functionality involves at least one function mechanism.

The **for in loop** is a variation of a *for loop* specifically created to traverse over objects. The temporary counter variable serves to dynamically store the key names of each property so that we can manipulate the data, such as printing to screen.

We know that *dot syntax* does not work with **dynamic** representations of properties. Also when we print a list of properties, we can filter out the methods by creating a *condition for output* with a **typeof** verification.

In the process of learning all these we also learned about *conditional statements* by using **if**, **else if**, **else**.

Boolean **true** and **false** were introduced and we learned that, in JavaScript, every value is

considered *true* except Boolean *false*, number *zero*, or the values *undefined* and *null*.

Functions are mechanisms that save executable code. In a sense, a function does not hold a value, it holds a potential value that only materializes when we call the function. A function can have input parameters as temporary placeholders for data being passed into the function when the function is called. Of course we can also have functions that do not take any data, they just perform a certain task.

We know that the ***return*** mechanism is the only true *raw output* from a function and we need to catch this returned data if we want to reuse it somewhere else before it gets lost in cyberspace.

Another way to output data from a function is to internally reassign an outer variable, since functions have access to outer variables. As for ***console.log()***, it is just a way to display data on the screen and always in a *string type* format.

And there you have it, close to 200 pages of fun in a nutshell.

3.2 More about functions

What are expressions and statements?

We have been using the terms expression and statement throughout the book in an informal way. It is time to stop and think about these terms because they have specific meanings.

- A **statement** is a complete command which instructs the computer to carry out a specific task. It's usually written in one line (but not always) and terminated by a semicolon.

Examples of statements:

```
var z = 1 + 2 + 3;  
console.log("Hello World");  
if(z > 2) {  
    console.log("yes it is");  
}
```

In the last example, the statement starts at *if* and ends at the *semicolon*. If you place the semicolon prematurely you will get an error. That is a common mistake when we first start programming.

We have also mentioned several times “*the return statement*”. Aside from a return being only allowed in functions, it is written as a statement:

```
return 3 + 4;
```

Many times statements are compounded into a group of multiple statements in which case we use the curly braces to wrap the compound statements:

```
if(z > 2) {  
    console.log("yes it is");  
    console.log("z is greater than 2");  
    console.log("z is also the last word of the alphabet");  
}
```

Notice how the last curly brace does not terminate with a semicolon. The only time a semicolon is placed after a curly is when we assign a code block to a variable like for example when we create an object.

Then we have expressions.

An expression is anything that *evaluates to a one unit*. Like for example, when JavaScript sees **3 + 4** it actually see **7**. JavaScript looks at the expression and simplifies the expression into 1 unit. Just like statements, expressions are everywhere and they can be one and the same.

Examples:

```
var x = 4;
```

Above, `var x = 4;` is a statement and also an expression. Actually we have two expressions. JavaScript looks at the code and subdivides the expressions into:

- 1- `x`, which results in 4 when evaluated, and
- 2- `4`, which results in itself. JavaScript looks a 4 and it expresses 4 as a result.

```
if( 10 < 11) { // do something; }
```

Above, `if(10 < 11)` is an expression because JavaScript does not see the literal `10<11`, it only sees *true* or *false*, one or the other. JavaScript does not really care for `10<11` in itself, it looks at it and internally replaces it with a Boolean *true*.

- This is a very important concept in order to understand the language: We write expressions, and JavaScript stores the result of their evaluation.

Of course `10` and `11` are individual expressions in their own right, but they are not as important as the combination of `10<11`. JavaScript interprets it and saves it as just plain *true*.

Now, look at the semicolon on the previous `if()` example. That's where the statement ends, it starts on *if* and it ends at **..thing;**

What about a function? Is it an expression in itself?

Take for example the following function:

```
function x (num1, num2) {  
  return num1 + num2;  
}
```

The above function by itself does not evaluate to anything. It could potentially evaluate to the sum of `num1 + num2` but only when the function is called upon:

```
x(2,4);
```

The function when called evaluates to 6.

How does it evaluate to 6 when we call function `x` with arguments 2 and 4?

num1 becomes an expression and it evaluates to 2.

num2 becomes an expression which evaluates to 4.

num1 + num2 become an expression which evaluate to 6.

Expression evaluation is an important concept in order to truly understand JavaScript (or any other language for that matter).

In the next topic we will see how to assign a function to a variable, making the combination of function and variable, an expression.

Remember, to declare a variable we start with ***var***. To declare a function we start with ***function***:

```
var x = 33;
```

```
function y () { }
```

Next we are going to see how to assign a function as a value to a variable and answer a few of common questions you may have.

Assigning functions to variables

Up to this point we have been declaring functions in the following literal manner:

```
function x () { return 5; }
```

However, we could also assign a function to a variable like this:

```
var x = function { return 5; };
```

Notice the semicolon at the end. This is because the whole variable declaration is in itself a statement. Statements are complete command sentences given to JavaScript and they usually terminate with a semicolon which separates them from the next statement.

Wait, I still don't get it. I don't see a semicolon on a function declaration like we have done before. Why does this one get a semicolon?

The semicolon has nothing to do with the function. A function in itself is not a complete command statement until it is called in. When we call a function, like for example **x()**; we add a semicolon. The earlier semicolon in question had to do with the variable assignment. When we assign a value to a variable we terminate the assignment with a semicolon because the assignment is a command sentence, or an expression.

The same goes for any other assignment such as an array:

```
var y = [1,2,3];
```

Or as for object assignments:

```
var z = {"color": "green"};
```

Or function assignments:

```
var z = function(){ return 5; };
```

What is an anonymous function?

In the previous example, the function is known to be an *anonymous* function because it has no proper name. **z** is just some variable that points to it.

However, without **z** the function ceases to exist because the browsers will clear it from memory since it has nothing point at it. Any loose objects in memory get discarded because they cannot be identified (remember functions are also objects).

Using variables as pointers

Speaking of pointers, the previous variable **z** points to the function assigned to it. We could use more than one pointer by assigning the “contents” of **z** to another variable.

Take for example the following function assignment:

```
var z = function(){ return 7; };
```

Now assign *a* to *z* and *b* to *a* to see what happens:

```
var a = z;
```

```
var b = a;
```

Remember, assignments are from right to left.

What is the content of variable **z**?

The content of variable **z** is ***function(){ return 7; }***. That’s right, the function script (the string of words and symbols that make up the function) is the content of variable **z**, or what **z** evaluates to by itself. The only time we get the value 7 is when we call the function by post fixing a pair of parentheses to **z**: ***z()***;

Because the **value** of **z** is a function script, when we assign **z** to ***a***, or **z** to ***b***, we are assigning the script of the function. This is very powerful in JavaScript because it means that we can plug in an existing function anywhere in the program and as many times as we need to.

Now we are able to call the function with any of the three variables:

```
a(); // returns 7
```

```
b(); // returns 7
```

```
z(); // returns 7
```

In summary

Remember, an expression is anything that evaluates to a one unit. This unit is what JavaScript actually sees from the code we write, not what the code looks like to the human eye. As programmers we need to think like a computer by knowing what the computer is going to see when it encounters an expression.

1- What is the value of expression `z`?

Ans: `function(){ return 7; }`

2- What is the value of `z()`;

Ans: `7`

3- What is the value of expression `a`?

Ans: `function(){ return 7; }`

4- What is the value of expression `a()`?

Ans: `7`

5- What is the value of expression `b`?

Ans: `function(){ return 7; }`

6- What is the value of expression `b()`?

Ans: `7`

And just to solidify the concept, here's a question unrelated to our examples:

7- What is the value of expression `if(10 > 5)`?

Ans: `true`

`true` is what JavaScript sees when we write `(10 > 5)`, which by the way, an expression in an `if()` statement is actually a question as far as JavaScript is concerned.

The balloon concept

Question:

If a function was a balloon tied by three strings, z , a , b , what would happen to the balloon if we cut off string z ?

Answer:

Nothing would happen to the balloon, it would stay the same because a and b still hold the balloon.

What would happen then if we cut both a and b from the balloon? The balloon would fly away.

If the balloon was our function it would also fly away, it would be put in [garbage collection](#) mode to be wiped out from memory by the browser.

In the next few exercises we will practice with function *expressions* (those assigned to variables) instead of function declarations (the syntax style we had first learnt). Bear in mind that one function or another will yield the same result, but it is important to become with both styles.

Function parameters and arguments

A function can have *input parameters* so that when we call the function we pass in *data arguments*. Many times programmers use the same term for both meanings, either parameter or arguments and it has been accepted as such, but this is the definition: we give arguments to parameters when we call the function. A parameter is a physical thing, an interface, an input mechanism, a temporary variable that gets deleted once the function ends its execution.

The sequence of our data arguments when we call the function will always coincide with the physical sequence of the input parameter in the function parentheses.

For example:

```
var x = function(num1, num2) {  
  return num1 - num2;  
};
```

The sequence of passing data into inputs *num1* and *num2* matters because the function's internal program is wired to the layout sequence of the input parameters:

```
x(5, 3);
```

It will return the value **2** because $5 - 3 = 2$

However, if we call the function and write the arguments in reverse order:

```
x(3,5);
```

The function will return the value **-2** because $3 - 5 = -2$

In short, when we call the function, the sequence of arguments needs to be observed.

- What happens if we call the function and pass in three arguments instead of two?

```
x(5, 3, 4);
```

The result will be **2**, as of $5 - 3 = 2$. The last argument (4) is ignored because there is no provision to do anything with it within the function.

- What happens if we call the function with just one argument even though the function is hardwired with two input parameters?

```
x(5);
```

In our case it returns **NaN**, which is a JavaScript property that stands for Not-a-Number, or not a valid number. That's because the output became the subtraction of **5** - **num2** and num2 was not assigned to anything and remained "*undefined*" which is its original value.

For example, in

```
var x = function(input1, input2){
```

```
return input1 + input2;  
};
```

If I call the function with two arguments:

```
x("Tony", "deAraujo");
```

I get an output of *"TonydeAraujo"*.

If on the other hand I only use one argument, it assumes the first input parameter, and the other input parameters will be *undefined*:

```
x("Tony");
```

It outputs *"Tonyundefined"*.

- We can program as many as 256 input parameters in a function.

Do not memorize all these, juts take the time to understand it and move forward.

The object arguments

The input parameters are written into a function by the programmer because he/she wants to have some sort of sequence of data input and for the most part this works out greatly.

However, JavaScript functions have another way to accept arguments via an object that belongs to (and only to) functions: the object *arguments* (use the plural name).

The object *arguments* acts like an array in the sense that it can be addressed numerically, but it is not a full featured array because it only [inherits](#) some functionality from the array family, not all of its library.

Let's see how this object works.

```
var x = function() {  
  return arguments[0] + arguments[1].toUpperCase();  
};
```

The above function has no input parameters specified. However, it outputs the first argument we give to it when we call the function, as well as the second argument in upper case which goes on position 1 (remember, locations are zero based just like in arrays):

```
x("tony", "dearaujo");
```

It returns: "tonyDEARAUJO"

So we can still program our functions as we wish and with no specified input parameters. Just use the *arguments object* to numerically address each data element.

I go into much more detail about the arguments object on my other [book](#) which, although written earlier, it could be considered a second volume for this one.

3.3 Manipulating variables with functions

Passing data by value

Up to this point we have used *input parameters* to hardwire and interface that inserts data into functions. These interfaces serve as a bridge between the outer world and the inner works of a function.

Then, we realized that we don't have to explicitly write parameters in order to input data into the function since we can program the *object arguments* to grab any argument we include when we call the function.

Here's something new to keep in mind:

When we pass arguments to a function, like for example:

```
x(2, 3);
```

The input parameters of the function make a **copy** of the data. They don't really use the same data we write as arguments. This is because the function call resides on a separate memory area than the function itself. Later, if you read *volume two* you will see that this area of memory where we call functions or assign strings or numbers, is called the *stack*, and functions (as well as all other objects) reside in another area called the *heap*. There is no connection between these two areas. Hence the reason why the data is copied or facsimiled.

When we copy data, we are adding values to memory. This type of passing data from one mechanism to another is called **passing by value**, or by adding more value to memory sort of speaking.

Another common data transfer done *by value* is when we assign a string or numerical variable to another variable. As you know from an earlier chapter, primitive variables (those holding strings or numbers) keep the value in their own memory location. Take for example the following variables:

```
var x = 33;
```

```
var y = x;
```

x holds 33 and *y* also holds 33. But these are different 33 values. Data has been passed by value because we added new data into memory.

Passing data by reference

Complex values such as functions, arrays and objects, reside in the *heap* while the variable that points to them resides in the *stack*. Unlike simple values, these values and their variables are two separate entities.

Because they are two separate entities, when we assign a variable pointing to a complex data structure to another variable, the values are not copied and therefore this exchange is not passed by value. The only thing we are doing is to assign the reference or passing by reference so that both variables now point at the same value:

Take for example the array *a*:

```
var a = [1, 2, 3, 4];
```

Now assign the same variable to another variable:

```
var b = a;
```

We do not duplicate the contents of the array, function or object. We now have two variables pointing at the same data. Remember the *balloon* with three strings *z*, *a*, *b*?

So when we edit a value in array *a*, we are in fact also changing the value of array *b*, or vice versa since they both point to the same array.

That is called **passing data by reference**. There is no duplication; these variables just refer to the same data. If we delete one of the variables or assign it to something else, the data remains untouched because there is another variable holding it in place like it happened with the balloon.

Functions are *closed- in* structures

In JavaScript, a function is the only [data structure](#) able to protect its data from the outer world.

What does that mean?

It means that if we declare a variable inside of a function, it can't be seen from the global scope (outside of the function).

The global scope is in itself an inner scope if we compare it to outer worlds, like for example another webpage on a different tab. Everything we do on a web page is done under the **window** object which wraps the whole page. This has nothing to do with the name Windows from Microsoft. *window* here refers to the browser displaying of a page, the shell, and it is called *window* in any computer system.

The window object is, as far as a web page is concerned, the outer world or the *global scope* of the page. Anything declared under the window object is a property or method of the object window.

We could actually think of the window object as a function that creates everything else inside it.

When we declare a variable, like for example

```
var x = 33;
```

We are creating this variable in the global scope, which means that the parent of variable *x* is object *window*. To call this variable, we write:

```
x;
```

And it displays 33. But we could also call it like in the following example and it would work the same:

```
window.x;
```

By using *dot* syntax we are making sure we call the correct variable *x*, not some other *x* from the outer galaxies. For the most part window is understood and we don't have to write it down, as we have seen throughout the book.

And what are those different galaxies? You might ask.

Remember when I said that we could think of the **window** object as a *big function* that holds all the web page functionality inside of itself and it protects it from other webpages?

Well, I didn't put it quite like that but it serves the purpose which is to say that every function acts as a protective parent for its own data. In JavaScript functions are *closed in* data structures.

We could actually declare another variable `x` inside of a function because JavaScript will accept it without conflict, even knowing that there is another `x` declared as a global variable. This is because the global scope does not see the inner `x` from the function.

However, the function can see the outer variable `x`. That's right! Functions have access to global variables because functions are their siblings. A function can see the outer variable world, but the outer world cannot see what's inside of a function. The siblings of a function can see the function, but they cannot see what is inside of the function. The good news is that JavaScript always picks the closest identifier called upon. I mean, if we would call `x` from within the function, JavaScript would grab the first `x` that sees which is the one inside of the function. So, even if there was another `x` outside of the function and knowing that the function has access to it, the inner `x` would be JavaScript's first choice if we ever addressed `x` from within the function.

For example:

```
var x = 33;
var test = function() {
  var x = 5;
  return x;
};
```

Call function test:

```
test();
```

It returns **5**.

So, how would we address the outer `x` from inside of the function if we wanted to use its value in the function?

window.x would come to mind.

We will see next how to access the global `x` from within the function.

“*This*” as an object placeholder

What if we wanted to return the outer variable `x`, instead of the inner `x` like we did on the previous example?

In that case we would have to use *dot syntax* to address the global `x`:

```
var x = 33;
var test = function() {
  var x = 5;
  return window.x;
};

test();
```

Now it returns `33`, which is the outer `x`, not the inner `x` which is `5`. That's because `window.x` addresses the global variable `x` which is a child of object `window`.

Because JavaScript is very dynamic and also because we should write code as portable as possible (one that could apply to any object, not just the window object), programmers have come up with a generic placeholder to properly address the object currently calling the variable.

This placeholder is the word ***this***.

If we substitute the expression `window.x` for `this.x`, at the time of execution JavaScript will search for the object owning the property being called upon, and substitutes the placeholder ***this*** for the correct name, which in our case is object `window`.

But you might ask, isn't function `test` also an object and if so, isn't the owner of ***this.x*** the function itself, instead of object `window`?

Not really (but it is a great question). ***Who is the owner of the act of calling function test()?***

Ans: The object **`window`** owns the command `test()`.

Now if function `test(){}` was a method inside of some other object, then ***this*** would no longer represent `window`, it would represent the other object.

Just as an example please look at this sample script:

```
var x = 33;
var myObject = {
  x: 7,
  test: function() {
    var x = 5;
    return this.x;
  }
};
```

There, we have a global `x` with the value of 33, another `x` with the value of 7 that belongs to ***myObject***, and another `x` with the value of 5 inside of the function belonging to `myObject`.

When we call the method `test()`:

```
myObject.test();
```

It prints out 7 which is the second `x`, the one that belongs to ***myObject***. *window* is no longer the immediate owner of `this.x` because *myObject* is.

I have underlined earlier that *this* gets assigned “at the time of execution”, not only because the substitution happens at the time of execution and not any earlier, but also because the object owning the property may change from the time the property was declared to the time the property is being called upon. JavaScript is a dynamic language and we need to be careful with our assumption of what is going to happen dynamically. We will see ownership confusion happening and how to tackle each situation as we code more advanced scripts (for more of that please read [JavaScript Objects Functions and Arrays Explained](#)).

Here’s the original function written in a more portable format where ***this*** replaces ***window***:

```
var x = 33;
var test = function() {
  var x = 5;
  return this.x;
};
```

`test()`; (It returns 33 instead of 5).

3.4 Lab work 12

Part A: Expressions, statements, arguments

(A link to my answers will be posted at the end of this page. You may actually navigate to the forum to mentally answer the questions because the answers are not visible until you click on the individual answer button.)

Let's review some of the concepts we have covered in this section. Please try to answer all questions before you check my answers.

In general an *expression* is something JavaScript summarizes into one unit, and a *statement* is a complete sentence representing a command and terminated by a semicolon.

- 1- In the following example what are the expression(s) and what are the statement(s)?

```
if(10 < 11) {  
  console.log("I love JavaScript");  
}
```

- 2- In the following function assignment what is the value of variable *addTwo* and what is the value of *addTwo* with post-fixed parentheses *addTwo()*?

```
var addTwo = function () {  
  return 3 + 2;  
};
```

- 3- In the declaration of the above source code, which section of *addTwo* is known as an *anonymous* function?
- 4- On the function assignment from question 2, if I assign variable *addTwo* to a new variable *x*, and then later I reassign *addTwo* to the number **33**, what will the value of both variable *x* and variable *x()* become?
- 5- What are function input parameters and why do we need parameters at all?
- 6- The object *arguments* is native to functions. How would we write two *console.log statements*, one to display the first value from object *arguments*, and the other to display the third value from object *arguments* of a function?

- Answers and further explanation can be found here:

[forum](https://bit.ly/1qqmwvg) | bit.ly/1qqmwvg

Part B: Passing data by value and by reference

(A link to my answers will be posted at the end of this page. You may actually navigate to the forum to mentally answer the questions because the answers are not visible until you click on the individual answer button)

- 1- Below, I have declared two variables. Variable ***b*** which has a numeric value of 33, and variable ***a*** which is assigned to the value of variable ***b***:

```
var b = 33;  
var a = b;
```

- a) Was this an assignment *by value* or an assignment *by reference*?
- b) What happens to the value of variable ***a***, if I add 1 to ***b***?

- 2- Below I have declared and assigned two variables. The first variable, ***c***, is assigned to an array. The second variable, ***d***, is assigned to a function.

```
var c = [1, 2, 3];  
var d = function() { return 9; };
```

- a) Suddenly, I assign the value of ***c*** to ***d***, like `d = c`; What is now the value of ***d***? What happens to the previous value of ***d*** after the reassignment?
- b) What is then the value of ***d*** if I reassign ***c*** to value *null*?

```
c = null;
```

- 3- In the following script, what will be the output of function ***myCastle()*** when we call it?

```
var f = 33;  
var myCastle = function() {  
  var f = "Santarem";  
  return f;  
};  
myCastle();
```

- 4- How would you rewrite the script to output the outer variable ***f***, instead of the inner variable ***f***?
- 5- Which object acts as the parent for all global variables in a normal web page?
- 6- What data structure in JavaScript has the capability of wrapping its elements in a private manner?

- Answers and further explanation can be found here:

[forum](https://bit.ly/1rm2pUg) | bit.ly/1rm2pUg.

Part C: Find and replace a word in a string

The goal for this project is to create a mechanism that finds a word in a paragraph and replaces it with another word. The idea is to replace *brown* with *red* on the following paragraph:

"The quick **brown** fox jumps over the lazy dog."

The tools to use are as follows:

- Several variable names to be used as helpers:
 - paragraph* – assigned to the original paragraph.
 - tempArray* – assigned to the result of converting the string to array.
 - oldWord* – assigned to the word "brown", the one we search for in order to replace.
 - newWord* – assigned to the word "red", the one we want to use as a replacement.
- *split()* which converts the string into an array of separate words.
The result of splitting the string will be assigned to an array called *tempArray*.
(The reason for converting the string sentence into an array of words is so that we can target the desired word to replace).
- A *for loop*. The loop will traverse the array in order to find the targeted word and replace it.
- *join()* which will later reconvert the array back to string.

We first split into array, then search and replace, then convert back to string and assign the result back to the variable *paragraph*.

Finally we use *console.log* which prints variable *paragraph* displaying the end result.

It may sound complicated but it becomes easier after doing it the first time.

Here's a recipe guide:

Declare the original string variable.

Convert the variable into individual words.

Introduce the word that needs to be found.

Introduce the new word to replace the existing word

Iterate over the array to find and to replace the word

Reconvert the modified array to string by assigning it to the original variable

Display your results

Take your time to think about it. Should you need an editor use *jsbeautifier.org* to write your script. Then paste it onto the Console for testing purposes.

- For a quick peek or further explanation and my own final version of this project please refer to the [forum board](https://bit.ly/1pqFACc) | bit.ly/1pqFACc.
Or just the [raw file](https://bit.ly/1uVMHjS) | bit.ly/1uVMHjS.

Part D: Using a function to save and recall our code

Great job! Now let's take it one step further:

Let's save the script in a function called *findReplace*.

The idea is to convert both variables *oldWord* and *newWord* into input parameters, instead of declaring them in the script like we've done before. This makes our search and replace more flexible because from this point on we can search for any word and replace it with any other word we want.

Use either choice of function style:

```
function findReplace () { }  
or  
var findReplace = function() { };
```

I will use the first option but either one will be fine.

Don't forget to include the input parameters.

Once you're done, call the function by choosing any word and replacing it with any other word of your choice.

Examples:

```
findReplace("brown", "red");  
findReplace("quick", "slow");  
findReplace("dog", "cat"); ... you will have a problem with this one because  
of the dot at the end of the paragraph.
```

- For a quick peek or further explanation and my own final version of this project please refer to the [forum board](https://bit.ly/1rtG4EK) | bit.ly/1rtG4EK.
or just the [raw file](https://bit.ly/1ypiKg3) | bit.ly/1ypiKg3

Part E: Making the search and replace more portable

On the previous project we created an automatic find and replace script, but it only worked with variable *paragraph*. It would be nice to have a portable function so that we can apply it to other variables or even to loose strings.

At this point it should be obvious the importance of using functions to create methods that can be reused in more than one application.

In order to make the project more portable we need to add a *third input parameter*, one which represents the targeted variable or just any string value for that matter.

- 1- All we have to do is to add a third input parameter and replace all the *paragraph* instance names from inside of the function, to this new input parameter's name.

I have called my input parameter *data*.

- For a quick peek or further explanation and my own final version of this project please refer to the [forum board](https://bit.ly/1mqMRhT) | bit.ly/1mqMRhT.
Or just the [raw file](https://bit.ly/1uijwc9). | bit.ly/1uijwc9.

If you can't think of an example to test your scrip please look at the forum board for two samples of test data. You should find them right below the script we have just discussed.

p.s. *I'm so happy that you have made it this far, thank you!*

We're almost done with the JavaScript lectures and exercises. Then, JSON and AngularJS will be optional studies that you can tackle later on if you need to take a break. However, please keep this truth in check: "out of sight, out of mind". You will forget if you don't keep practicing regularly, then you will have to read the book from the beginning again.

3.5 More branching techniques

The switch

So far we have learned about the *if()*, possible *else if()* and optional *else* conditional statements, right?

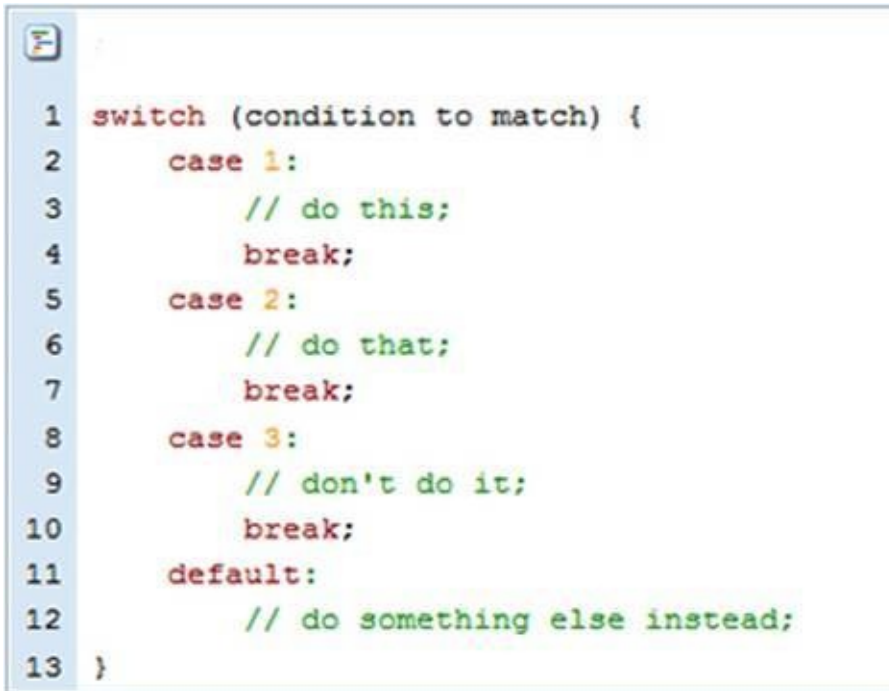
```
if(true) {  
  // do this;  
} else if(true) {  
  // do that;  
} else {  
  // do this one instead;  
}
```

You can do any possible outcome by using the above powerful combination of if, elses.

Sometimes however, a different kind of branching makes more sense. I'm talking about the one known as ***Switch*** and I just want to make sure it is explained in case you need to use it.

It is called *switch* because there are several possible outcomes from which JavaScript makes an executable decision. These possible outcomes are all in *off* state, and JavaScript will turn one *on* based on a matching Boolean condition, as we you'll see next.

Here's the layout diagram:



```
1 switch (condition to match) {
2   case 1:
3     // do this;
4     break;
5   case 2:
6     // do that;
7     break;
8   case 3:
9     // don't do it;
10    break;
11  default:
12    // do something else instead;
13 }
```

Fig 16

On line 1 we declare a switch. This switch function method takes a condition from which JavaScript will compare all the outcomes in order to select one of the possible executions.

In our generic example (based on the cases we have), the “condition to match” should be substituted by either a 1, 2, or 3, so it matches one of the cases presented. If any of these cases matches the condition, JavaScript executes the code given by the case. The Boolean *true* is what unlocks the case.

Example: if the correct case is 3, JavaScript will ask:

“is it 1? Ans: false”,

is it 2? Ans: false”,

is it 3? Ans: true”, run the code,

BREAK and END.

After the execution of a case, JavaScript exits the switch due to the *break* command that follows. If we don't include the *break* command, then JavaScript will search for more possible results and you may have several correct answers instead of just one. And even if there are no other correct answers, JavaScript still reads all the other conditions which could be a time delay if the switch has thousands of possibilities programmed in it. So the *break* is a very important implementation.

Whenever there are no conditions that match, the optional *default* takes over as the final and *true* option and its code is executed. Notice that I did not include a break after default. You could, but it is not necessary because JavaScript exits the switch at this time anyway.

For further reading please visit [this post](https://bit.ly/1r4XRCL) | bit.ly/1r4XRCL on the *forum*. Let's practice a bit with switches.

3.6 Lab work 13

13a: Switch - using Boolean matches

This script will employ a switch with 4 possible outcomes. The user will try to guess a number, and the program will display a feedback message of either, *too low*, *too high*, *right!* or *That's not a number*:

1- Create a switch inside of a function named *guessNum*.

2- This function takes one input parameter, *num* .

3- The switch should be hard wired to *true*.

Example: `switch(true) *`

4- The cases should have the following conditions:

`num <= 3` displays “too low”

`num >=5` displays “too high”

`num === 4` displays “right, the number was 4”

for **default** use “*That's not a number*”

5- Call the function by passing a numeric argument like for example:

`guessNum(2) ;`

`guessNum(7) ;`

`guessNum(4) ;`

`guessNum("a") ;`

- For a quick peek or further explanation and my own final version of this project please refer to the

[forum board](https://bit.ly/1qqwkWc) | bit.ly/1qqwkWc

or just the [raw file](https://bit.ly/1r50ODq) | bit.ly/1r50ODq

- * Think of **expressions**.

`num <= 3` will be converted to *true* or *false*. The same goes for all the other cases and therefore the switch to match should be a Boolean *true*.

13b: Switch - using literal matches

On this next project we are going to use the following:

- A function named *myFavColor*,
- A `prompt()` to get an input from the user, which will be converted to lower case so it matches one of the cases. (JavaScript is case sensitive).
- Some `confirm()` outputs so that we don't always use `console.log`,
- And some meaningful statements about *color*.

The statements are as follows (you can always make your own or copy from this

[raw file](#) | `bit.ly/lypn1A2`) :

- Green is the color of balance and growth.
- Blue is the color of trust and peace.
- Indigo is the color of intuition.
- Purple is the color of the imagination.
- Orange is the color of social communication and optimism.
- Sorry, that color is not in the system yet!

- 1- Please remember: the purpose of creating a function is to recall the program without having to retype it again. It works great for testing purposes and in real life as well. So the whole switch should be wrapped in a function. Please feel free to choose the function style you prefer, a function declaration, or a function expression which is the one assigned to a variable.

I will use the first method:

```
function myFavColor() {}
```

- 2- Inside of the function, the *prompt()* will be assigned to variable *color*.
I will convert the input from the user into all lowercased letters so that it matches my cases which will be in all in small letters.

```
var color = prompt("Enter your favorite color").toLowerCase();
```

- 3- The switch expression should be the value of the variable *color*:

```
switch (color) { }
```

- 4- Each case needs to be in small letters because the input will be converted to lowercase:

```
case "green":
```

- 5- The output from each case should be a *confirm()* followed by a *break*:

```
confirm("Green is the color of balance and growth.");  
break;
```

- 6- When no color matches, the *default* execution takes over and it could be something like the following (no break is necessary):

```
default:  
confirm("Sorry, that color is not in the system yet!");
```

- 7- Finally, don't forget the two closing braces: one for the switch and another for the function (the ones shown in red on the top of this page , steps 1 and 3).

- For a quick peek or further explanation and my own final version of this project please refer to the

[forum board](https://bit.ly/ZcaDUC) | bit.ly/ZcaDUC

or just the [raw file](https://bit.ly/1DwVBsd) | bit.ly/1DwVBsd

To test your script just call the function and follow the prompts:

```
myFavColor();
```

13c: A repeating switch routine

Sometimes we want the script to run repeatedly such as for example in a game, until we purposely decide to exit.

In order to make our previous script repeatable all we have to do is to call the function *myFavColor()* just before we close the function, and the function will trigger a new instance of the game.

When I first introduced the *confirm()* mechanism I had mentioned that *confirm()* can return a Boolean **true** if we press **OK**, or a Boolean **false** if we press **Cancel**. We can catch this returned data to trigger the new *function call*, and one way to do it is to add the following code just before closing the outer function (the variable name *replay* is arbitrary):

```
var replay = confirm("click ok to play again, or cancel to exit")
  if (replay === true) {
    myFavColor();
  }
} <— this curly brace is already there, it is the very last brace on the bottom.
```

So in a sense we grab the Boolean *true* from the *confirm* mechanism and reuse it to call *myFavColor()*;

- For a quick peek or further explanation and my own final version of this project please refer to the [forum board](https://bit.ly/1ogVffh) | bit.ly/1ogVffh
or just the [raw file](https://bit.ly/1vdaYQT) | bit.ly/1vdaYQT

13d: Using a while loop to repeat the script

We have not covered while loops. The only loops we have covered were the *for loop* for arrays and the *for in loop* for objects. There are many different styles of loops and they are all based on the for loop we first learned.

A *while loop* is one that runs indefinitely until something triggers the loop to exit.

A popular example of a while loop is the mechanism that scans the computer keyboard. The while loop keeps reading the input from the keyboard until it gets a signal from one of the keys which in turn triggers another mechanism.

The design for a while loop is as follows:

```
while(this Boolean expression is true){  
  // do this stuff;  
  // something to change the condition to false in order to  
  end the loop;  
}
```

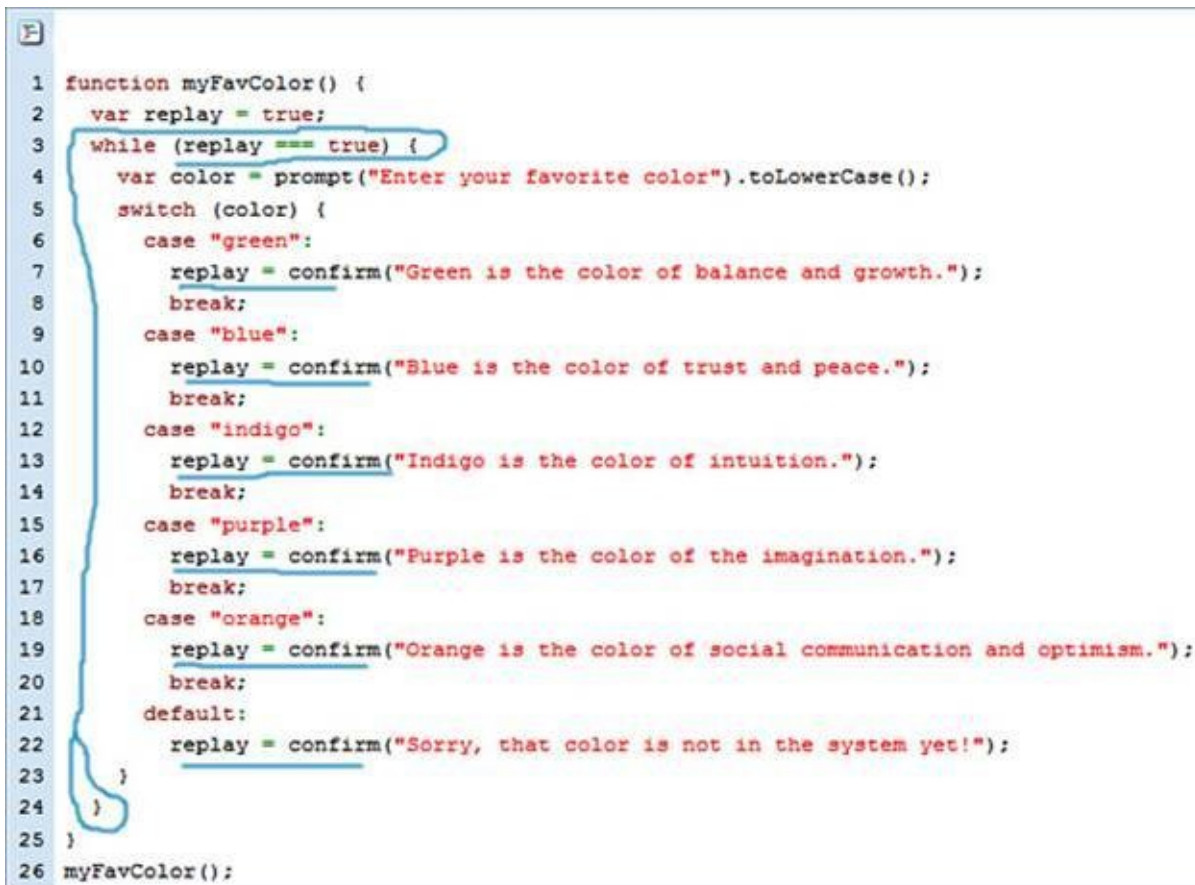
Until that something triggers a Boolean false, the *while loop* will always run.

We can use this while loop on our previous color description game to run the switch forever until the user clicks on the confirm()'s cancel button. In this way we don't have to recall the function again, all we have to do is to keep asking the user to enter his/her favorite color, until the user press the cancel button.

So where do we implement the change from Boolean true, to Boolean false?

We implement it on each case because only one case will apply. When the user confirms the output message for the case, the user also has a chance to press cancel when he/she wants to exit the game. Until the user presses *cancel* the game will go on forever.

Please look at the next image to see what I mean:



```

1 function myFavColor() {
2   var replay = true;
3   while (replay === true) {
4     var color = prompt("Enter your favorite color").toLowerCase();
5     switch (color) {
6       case "green":
7         replay = confirm("Green is the color of balance and growth.");
8         break;
9       case "blue":
10        replay = confirm("Blue is the color of trust and peace.");
11        break;
12       case "indigo":
13        replay = confirm("Indigo is the color of intuition.");
14        break;
15       case "purple":
16        replay = confirm("Purple is the color of the imagination.");
17        break;
18       case "orange":
19        replay = confirm("Orange is the color of social communication and optimism.");
20        break;
21       default:
22        replay = confirm("Sorry, that color is not in the system yet!");
23      }
24    }
25  }
26  myFavColor();

```

Fig 17

On line 3 we encapsulate the whole switch mechanism inside of a while loop.

While *replay* is *true* (which is its initial state), the switch will run and the loop will restart the switch again and again until we press Cancel. Pressing ENTER is equivalent to pressing OK and that means to run another cycle. Eventually we get tired of playing the game and press the Cancel option on the confirm box which will trigger JavaScript to exit the loop.

Notice how the value of variable *replay*, which is originally a Boolean *true*, is renewed each time an output message is prompted. This gives the user the opportunity to opt out of the game.

It looks more professional than the previous script we had done and if you are interested on creating games this can become handy.

- After you code your own version of it you can compare it with mine on the [forum board](https://bit.ly/1DwVNYo) | bit.ly/1DwVNYo.
or just the [raw file](https://bit.ly/1sWgeWf) | bit.ly/1sWgeWf

PART IV: ENTERING THE THIRD REALM

Please note:

The book is getting long! If you wish to take a break, this is the right time to do so because what we are going to cover next is supplemental, but not part of the JavaScript syntax *core*.

On the other hand, if you are in a hurry to get to AngularJS you can safely jump over JSON for the moment and go to PART V: DEFEATING THE DRAGON.

JSON is a necessary topic to know about since you will be using it sooner or later, but it is not needed for what we are covering on PART V.

I hope your time has been productive and I'm really grateful for the investment and trust you placed in this material.

[Tony de Araujo](#)

4.1 About this section

We have covered a lot of JavaScript territory and there is so much more to write about. However, it is not a good idea to do it all at once because it loses its efficiency and defeats the purpose. This is the reason why I have subdivided JavaScript into several different booklets: to give the reader time to digest all the data covered in each publication. Continual practice and repetition with conscious awareness expansion seems to be a better way to go.

[JavaScript Objects Functions and Arrays Explained](#) goes into more detail about each JavaScript Library method, as well as providing more advanced practice of concepts. If you did all the exercises covered up to this point, you should be more than ready to tackle the other volume.

Another very popular and very inexpensive eBook for practicing purposes is [Draw Six Lucky Numbers](#), which covers loops, random numbers and it reviews functions and arrays. This is a quick project for a Sunday afternoon.

Now we have come to a cross roads.

The rest of this book will be introducing two very hot technologies that every JavaScript intermediate to advanced programmer should know:

JSON (JavaScript Object Notation) is a way of formatting data to become machine independent. This format is in many ways replacing XML and being used in modern web programming. In a sense I have been introducing it from the beginning of the book so this material should look familiar to you and therefore it will be a relatively short section.

The final area of coverage is about Google's framework library **AngularJS**, which is based on JavaScript.

This library is revolutionizing the way we build websites and although still in its infancy, it has already made a permanent impact and you will see why as you get to it. By the time you finish the next few chapters you will be able to implement dynamic data into your basic webpage in an easy and comprehensive way.

Please keep in mind that I am only introducing AngularJS, this is not a complete book about it since the topic spans over several volumes. What I hope to achieve by writing this material, is to empower you to pursue further studies of AngularJS. This information will unlock what I consider the hardest part of getting into AngularJS, the very first steps. Through the exercises and explanations given you should be able to easily conquer this first barrier.

4.2 An introduction to JSON

JavaScript Object Notation, **JSON** (it is pronounced J SON), is an open standard format that uses human-readable text to transmit data objects consisting of key-value pairs (JavaScript objects). It has become an alternate technology to XML in its use for transmitting data between a server and web application. The term ‘alternate’ may be outdated because nowadays JSON is the preferred method for web services.

The reason why I am introducing JSON at this point is because at the time of this writing JSON has become the must know subset of JavaScript if one wants to build modern web pages and web apps. It should be now part the any web literacy curriculum.

JSON has achieved such popularity that other languages are now implementing it in their own vocabulary.

The good news is that you have already been prepared for this technology throughout the book and it is just a matter of becoming more aware of its existence.

JSON format is syntactically the same as when we create lists, being it numeric as an array or with key-value pairs as in an object. We can also combine the two formats to create a more complex JSON data structure.

The following is a collection of data written in the JSON format:

```
{  
  "firstName": "Tony",  
  "lastName": "deAraujo",  
  "age": 99  
}
```

Or we can also have an array:

```
[1, "hello", 33]
```

Please note: If you are planning to convert an array into a JSON format, avoid using mixed array data types like the array I’ve just shown you. This is because JSON is supposed to be a language neutral data exchange protocol and many programming language do not allow mixed arrays (those containing mixed value types) and this would cause confusion. The following would be ok because numbers are seen as strings (compare with my previous example):

```
["1", "hello", "33"]
```

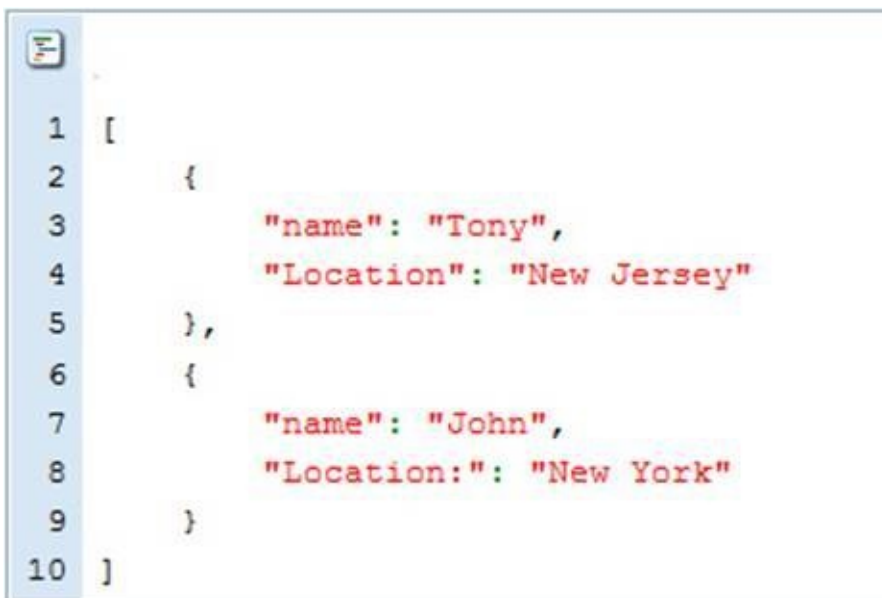
Going back to our first example on the top, notice how there is no variable assignment to the JSON object or array. The JSON part of the object is the one that gets exported into other systems in a seamless manner.

When writing in JSON format we must wrap the key in double quotes. Since we have used double quotes from the very beginning we should have no problems with that.

JSON will throw an error if keys are not wrapped in double quotes. Single quotes will not work.

Also none of these data structures terminate with a semicolon. This is not a variable assignment. JSON data is assigned to a variable at its destination so that we can manipulate the data, but the data in itself is exported without variables. The outer most tags wrapping the whole data structure must be curly braces, like in an object format.

In a more complex design you may see an array of objects. Arrays work out great to gather a bunch of objects that belong to a certain class criteria:



```
1  [  
2      {  
3          "name": "Tony",  
4          "Location": "New Jersey"  
5      },  
6      {  
7          "name": "John",  
8          "Location": "New York"  
9      }  
10 ]
```

Fig 18 (note: missing outer curly braces and a key for the array. See explanation below.)

This is the data that can be portable from machine to machine. However, **this is not yet a valid JSON object** and using it like this can lead to an unsecure transmission. It needs outer curly braces and a key name for the array in order to become a good JSON data format as we will see on the next image.

By writing it this way (plus the outer curly braces) we can feed it into any machine able to translate JSON data into their own machine format, much like XML does except that JSON is easier to implement.

For a while now, web pages have been communicating with the server by exchanging JSON data. What I mean is that the page loads on your browser and every time you click on a dynamic field, the page remains the same but the data is updated via a JSON data format exchange. This is one of the reasons why becoming aware of this technology is a must.

- **Keep this in mind:** JSON-formatted text is also syntactically legal JavaScript code. However, the opposite or other way around may not be true. This is the reason why, when we learned to create objects earlier, I chose to wrap the keys in double quotes, making it legal syntax for both JavaScript and JSON.

The next image will show a more extensive example of a JSON object. You will see the outer object subdivided into two key names: *staff* and *management*.

Each one of these keys, staff/management, will have an *array* as a value. The arrays will contain several individual records in the form of objects. Each object represents an employee from either staff or management. Let's look at the image:



```
1 {  
2   "staff": [  
3     {  
4       "firstName": "Tony",  
5       "lastName": "deAraujo"  
6     }  
7   ],  
8   "management": [  
9     {  
10      "firstName": "Judy",  
11      "lastName": "Garlad"  
12    }  
13  ]  
14 }
```

Fig 19 see [raw file](https://bit.ly/1uF1qNH) | bit.ly/1uF1qNH

Do you see how that is done? There are two properties: *staff* and *management*.

In our example, the property *staff* contains an **array** of people assigned to it, and each person is also an object with its own data. Right now we only have “Tony” as staff and we only have “Judy” as management, but we could add a comma after the closing brace from “Tony” or from “Judy”, and add another person to each array, as you will see on the picture further down the page.

Don’t get confused here. Arrays are not necessary in order to create JSON data structures. We can have arrays, strings, numbers and objects, represented in a JSON structure. Every identifier needs to be wrapped in double quotes, except for numbers in which quotes are optional. (As noted earlier, always double quote numbers if they are contained in an array, in order to comply with programming languages that forbid mixed arrays).

Use jsonlint, a tool to check JSON validation

In order to see if a code block is syntactically well written in JSON we can check it with an online tool called [jsonlint](https://jsonlint.com) (jsonlint.com). This tool becomes an important check in order to avoid errors or unsecured JSON data scripts.

Practicing exercise:

- 1- Copy the code from the previous image and paste it on the box at *jsonlint*. Then click *validate*. If everything is well written, you should see a green bar across the page.
- 2- Try removing a brace, or adding a variable name, or a semicolon just to see if it still validates.

Assigning a JSON object to a variable

Now, if we want to manipulate this data after we receive it at the destination, we need to assign it to a variable. The variable will point to the data in memory so that we can address it. That's when JavaScript comes in: We move data around in the JSON format, and at the destination we assign the data to a variable if we want to further manipulate the data.

The next extended image will show how we can assign a *variable* to the JSON object.

Remember, when writing your code, if you want to validate your JSON data with [jsonlint](#), you must do it before assigning the JSON data to a variable and before adding the semicolon at the end to terminate the statement.



```
1 var employees = {
2   "staff": [{
3     "firstName": "Tony",
4     "lastName": "deAraujo",
5     "age": 99
6   }, {
7     "firstName": "John",
8     "lastName": "Smith",
9     "age": 33
10  }, {
11    "firstName": "Mary",
12    "lastName": "Adams",
13    "age": 29
14  }],
15  "management": [{
16    "firstName": "Judy",
17    "lastName": "Garland",
18    "age": 43
19  }]
20 }; <-- added semicolon
```

Fig 20 see [raw file](#) | bit.ly/1DwXd57.

- 3- On this step extend your own object with more data, or copy it from the raw file below image 20.

If you are writing your own script and want to validate it with **jsonlint**, do it before adding the variable (ex: **var employees =**) and last semicolon, then assign the variable *employees* to the object.

Now that we have assigned the JSON object to the variable *employees*, we can **edit**, **add**, **delete** and **display** the data or part of the data because the object has a *name* we can refer to.

Editing the JSON object

The property key *staff* has an array assigned to it. The decision to have an array was so that we could include several objects as part of *staff*. Each person in the array is an object in itself. The same goes for key *management* which is also a property of the object *employees*.

Practicing exercise:

4- If you haven't done so, copy the raw file and paste it onto your JavaScript Console.

5- How would we address *staff* in order to get a list of its members?

We could use dot syntax in the following manner:

`employees.staff;`

That would give us a list of employees from the staff array. Your Console might display a short version of it, something like *object, object, object*, which represents each employee, but in real life you would get the complete list. To see the contents of each object, just click on it (see the image below). That's how the Console treats the output.



Fig 21

NOTE: The proto property is a list of all the methods from the JavaScript library that apply to this object. Just ignore it because it has nothing to do with JSON and it is not portable. I cover all these library methods on the [Volume 2](#) book.

6- How would we address the first object in the staff array?

`employees.staff[0];`

Explanation:

We start with the outer object which is *employees*, and then address its *staff* member. Then, since *staff* holds as array of inner objects, we address the first element of the array which is at location *zero*, and the result is as follows:

Object {firstName: "Tony", lastName: "deAraujo", age: 99}

7- Now try addressing the object in the array held by *staff* at the position containing *Mary*. She is the third element of the staff array, which corresponds to position 2.

See the result here: [raw file](#) | bit.ly/1sWuXAY.

8- And how would we add a new employee to the staff class?

This is the new employee we need to add:

“firstName”:“Loren”, “lastName”:“Santos”, “age”:29

Since Loren is an object that needs to be added to an array, we could use the array method **push()** to push it into staff: (see [raw file](#) | bit.ly/1wKTPPB).

```
employees.staff.push({"firstName":"Loren", "lastName":"Santos", "age":29});
```

9- Try adding a new member to employees *management*, like for example:

“firstName”:“Peter”, “lastName”:“Jones”, “age”:55

When finished, try addressing (displaying) this new member at its location in the array assigned to management.

See [raw file](#) | bit.ly/1tXHoQM

10- Call the employees property management to see what it contains:

```
> employees.management;
[▼ Object 3, ▼ Object 3]
  age: 43
  firstName: "Judy"
  lastName: "Garland"
  ▶ __proto__: Object
  age: 55
  firstName: "Peter"
  lastName: "Jones"
  ▶ __proto__: Object
```

Great! Now let's just look at two useful methods to work with JSON data structures.

JSON library methods

JavaScript has two functions to parse regular JavaScript into a JSON format, or vice versa. Once data is converted into a JSON format we can send the data across the globe into other machines.

Let's see how they basically work.

JSON.stringify

The function *stringify* compresses the object by stripping its white space and converting it into a string so it can be exported.

Taking for example our previous object *employees* (as seen on image 20), we can stringify it like this:

```
var myString = JSON.stringify(employees);
```

I have assigned the result to variable myString (any name will do) which is now a string variable and contains the following data:

```
"{"staff":[{"firstName":"Tony","lastName":"deAraujo","age":99},
{"firstName":"John","lastName":"Smith","age":33},
{"firstName":"Mary","lastName":"Adams","age":29}], "management":
[{"firstName":"Judy","lastName":"Garland","age":43}]}"
```

Now this string of data is ready to be sent via *http* or any other method to a different location or machine. Any programming language with a way to decode JSON (convert to or read to JSON) will be able to use this data.

NOTE(1): If you want to validate this data with JSONLINT.com you need to exclude the variable name and the outer quotes.

NOTE(2): When using *stringify*, functions (methods) inside of the original object will not be included. This is intentionally done. The only data that gets stringified are the string key/value pairs which are wrapped in quotes (as well as numbers).

NOTE(3): There is a way to prevent JSON from compressing the data. That implementation will be discussed when we cover the third argument for stringify, a few pages from now.

Using a second argument as a filter on stringify

NOTE: If this is too much information for now, just read it, understand it, be aware of it and return when you want to revisit the subject.

The method `stringify` can take a second argument to **filter** out everything except what we want to include during stringify. This second argument can be an array, or it can be a function, depending on what we are trying to accomplish.

(a) When using an array as a filter

When we use an array as second argument, we pick and choose the data we want to stringify.

As always, the first argument is the object name being used. When the second argument is an array, JavaScript only stringifies the key/values provided in that array. The array acts as an inclusive filter.

For example, I want to stringify just the array *staff* and only just the first name of each employee:

```
var someString = JSON.stringify(employees, ["staff",
"firstName"]);
```

JavaScript captures *staff* and grabs its *firstName* property. The result of this serialization is as follows:

```
{ "staff": [ { "firstName": "Tony" }, { "firstName": "John" }, { "firstName": "Mary" } ] }
```

Here's another example where *age* is also included (but not *lastName*):

```
var someString = JSON.stringify(employees, ["staff",
"firstName", "age"]);
```

What if we want the first name of both *staff* and *management*?

```
var someString = JSON.stringify(employees, ["staff",
"management", "firstName"]);
```

It results in:

```
{ "staff": [ { "firstName": "Tony" }, { "firstName": "John" }, { "firstName": "Mary" } ], "management": [ { "firstName": "Judy" } ] }
```

NOTE: If you try validating this result with [jsonlint](#) in its raw format, it will fail. The reason is that we need to strip the outer quotation marks (the ones making this whole script a string). Once you remove the outer quotes you will get a green light from jsonlint. There is a method to convert this string back into an object. This method will be discussed on the next chapter.

Try this one on your own:

If you want some practice, try stringifying both *staff* and *management* but only the last name and age of each employee. Use a variable name of your choice.

(b) When using a function as a filter

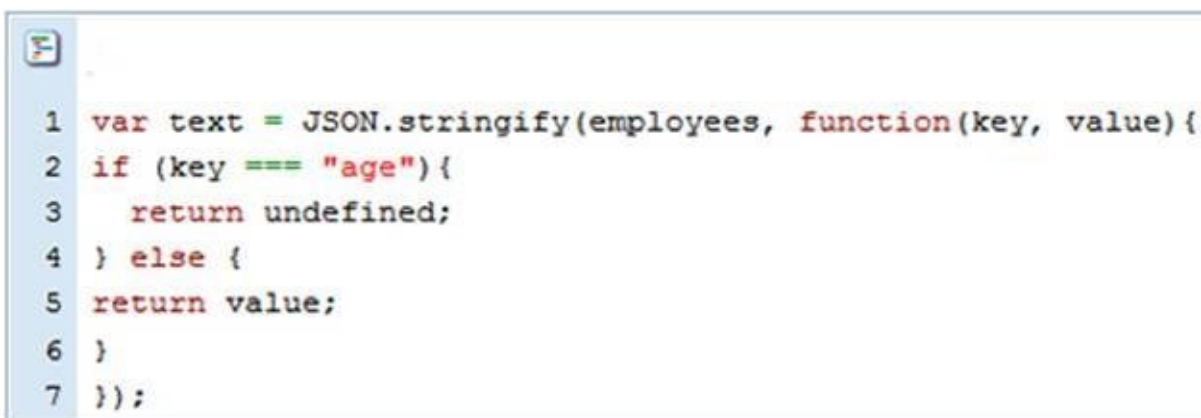
So far we have used an array as a second argument for *stringify*. That allowed us to pick and choose which elements we wanted to include in the stringification.

With a function as a second parameter we can do the opposite, **exclude elements**, or modify how certain elements will be represented.

The function takes two arguments in its input parameters. The first argument is the *key* and the second argument is the *value*. These two parameters will represent every key and every value of the JSON object. In other words, because we are grabbing each key and each value for evaluation before processing, we can write a script in the body that addresses specific keys and specific values, as the function scans each key and each value (see the image below).

When it comes to parameter names, it doesn't matter what you call them, what matters is their position which needs to be (key, value). You can call them (k,v) or (x,y), or anything else.

As an example, the following image uses the function filter to exclude the *age* from the stringification:



```
1 var text = JSON.stringify(employees, function(key, value){
2   if (key === "age"){
3     return undefined;
4   } else {
5     return value;
6   }
7 });
```

Fig 22 see [raw file](https://bit.ly/1v71zuT) | bit.ly/1v71zuT.

What happens here is that an **undefined** value will not be included as a JSON file because in JSON, undefined represents an empty key/value pair, so *age* will be stripped from the output.

As another example, let's say that you want to show the label *age* but with a value of *n/a*. You could change the return value from *undefined* which strips the key, to the following:
return "n/a";

Now "age" will show "n/a" as a value.

Great!

Moving along, what about programming the opposite, like for example when we receive JSON data and want to reconvert it back into an object?

We will look at the reconversion in a few minutes but let's first discuss how to prevent

stringify() from compressing the data (I mean how to keep the original white space), because it completes our discussion on the function *stringify()*.

Using a third argument on stringify() for controlling white space

Even if we don't want to use a second argument to filter out the data (like for example an array or a function as explained on the previous page), we could still use a third argument (or parameter) to control the white spacing when we *stringify*.

Taking for example our original *employees* object (see [raw file](#) | bit.ly/1DwXd57), instead of *stringifying* it like we did before:

```
var myString = JSON.stringify(employees) ;
```

We could instead, add a third argument, which would activate the third input parameter of function stringify:

```
var myString = JSON.stringify(employees, null, 4) ;
```

And the output would look like in this [raw file](#) | bit.ly/1oh4ts1.

First, notice how we made JavaScript read the third argument without having a second argument:

- We made the second argument *null*. That works because now we can have a third argument.

Second, the number 4 is the number of indented spaces. There is no provision for adding lines but JavaScript automatically adds lines when we ask for indentation. The maximum number is 10.

Did you notice the second sample on the raw file? It shows *dots* instead of *spaces*.

That was done by substituting the third argument from a number to a string:

```
var myString = JSON.stringify(employees, null, "...") ;
```

This last one will not validate if you test it on *jsonlint*, but it serves to illustrate the third argument in *stringify*.

JSON.parse

To convert the string back into an object just like we had it before, we use the method `JSON.parse`.

See raw file: [myString](https://bit.ly/1rtZclU) | bit.ly/1rtZclU.

```
var myEmployees = JSON.parse(myString);
```

Variable *myEmployees* is now an object with the same values as our original *employees* object.

Note: You will get an error if the data being parsed is not a valid JSON.

Properties of non-array objects are not guaranteed to be stringified in any particular order (which makes sense since key-value pairs don't really need a numeric order). Do not rely on ordering of properties within the same object when stringifying.

In summary

I hope this introduction to JSON has given you the taste for it and inspire you to start your own data import export projects.

Here are a few useful links:

- [JSON.org](https://json.org), the official website.
- [JSONLint](https://jsonlint.com), the JSON validator tool.
- [Wikipedia](https://en.wikipedia.org/wiki/JSON), an independent reference of resources.
- [Mozilla Foundation](https://developer.mozilla.org/en-US/docs/Glossary/JSON), a good glossary and resource center.
- [JSON vs XML](https://www.json.org/json-xml-comparison.html) – A comparison of these two technologies written by JSON.org.

Before we move on, let me just include a brief introduction of another technology where JSON really shines: NoSQL.

Document-oriented databases for JSON data

Modern web applications have data needs that relational databases like MySQL may have trouble delivering. A massive change is underway and is disrupting the database world as we know it. Today, three interrelated trends are Big Data, Big Users, and Cloud Computing – all pushing the adoption of NoSQL technology.

What is NoSQL?

“A NoSQL or Not Only SQL database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases.”

[Wikipedia](#).

The reason for the term “Not only SQL” database is to emphasize that they may support SQL-like query languages.

NoSQL is increasingly used in [real-time web](#) applications (example: Facebook’s news feed and Twitter).

NoSQL are document-oriented databases. Compared to relational databases, a collection of data could be considered analogous to a table and a document analogous to a record (a document could be encapsulated and encoded in one of the standard formats such as XML or JSON).

A list of NoSQL databases can be found here: [nosql-database.org](#).

A good starting point if you want to get into this technology right away is to download [CouchDB](#), which stores data as JSON and it can be queried in JavaScript . You can use it on your local computer and learn its operation from there. CouchDB is an Open Source project supported by Apache. A collection of basic lessons can be found here: [Guides](#).

PART V: DEFEATING THE DRAGON

Welcome to modern DOM scripting.

October 2015

5.3 First solid steps into AngularJS

Intro

AngularJS is a framework for developing dynamic web pages. It was created by Google to address multiple problems with conventional web design. As many other developers, I believe AngularJS is the hottest technology available and the future of web design, and because it is backed by Google, there will be plenty of resources to make it stick.

In a way, AngularJS extends HTML attributes, giving them superpowers thanks to the library provided by AngularJS. The library is connected to your webpage via a link just like you would do for CSS or JavaScript.

HTML was designed for static pages. AngularJS is what HTML could have been if it was designed for modern applications.

At this point you really need to know basic HTML because we will be working with a standard HTML page. You don't need to be an expert, the very basic stuff will do. If you need to refresh your mind or learn from scratch, give [Codecademy](#) a visit. You may see me there at times as a moderator, especially on their JavaScript exercises, but their HTML tutorial is pretty cool and free!!

Links of interest (for future reference):

[AngularJS.org](#) | [Wikipedia](#) | [Documentation](#).

Before you get yourself lost reading their material let's become familiar with the basic setup and what it can do for a web developer trying to design a more dynamic web page.

Welcome to modern web design!

Directives

How does AngularJS extend HTML capabilities?

First we provide a link on the web page to the AngularJS' library, just like we would link to a JavaScript or a CSS file.

Then the page becomes linked to this huge plethora of methods that can be used by you.

Let's suppose that we want to supercharge a DIV and nothing else on the page, like for example, you want to create a box in your web page with some dynamic data.

After linking to the library, all we have to do is to apply a *directive* to the DIV. The directive is **ng-app**, which makes this DIV an application controlled by AngularJS. The ng-app directive tells AngularJS that this DIV needs to be scanned for possible AngularJS functionality. We will see how this works in a moment.

If we want the whole page to be scanned and mapped, we apply the same directive to the HTML tag instead of applying it to a DIV. Actually, any HTML tag able to be a container can become an Angular supercharged element without even affecting the rest of the page. We could use a paragraph tag, or a form tag, or an ordered list. For the most part however, AngularJS is so friendly that we don't mind associating the whole page to it.

Please look at the following image to have a better idea of what I mean:

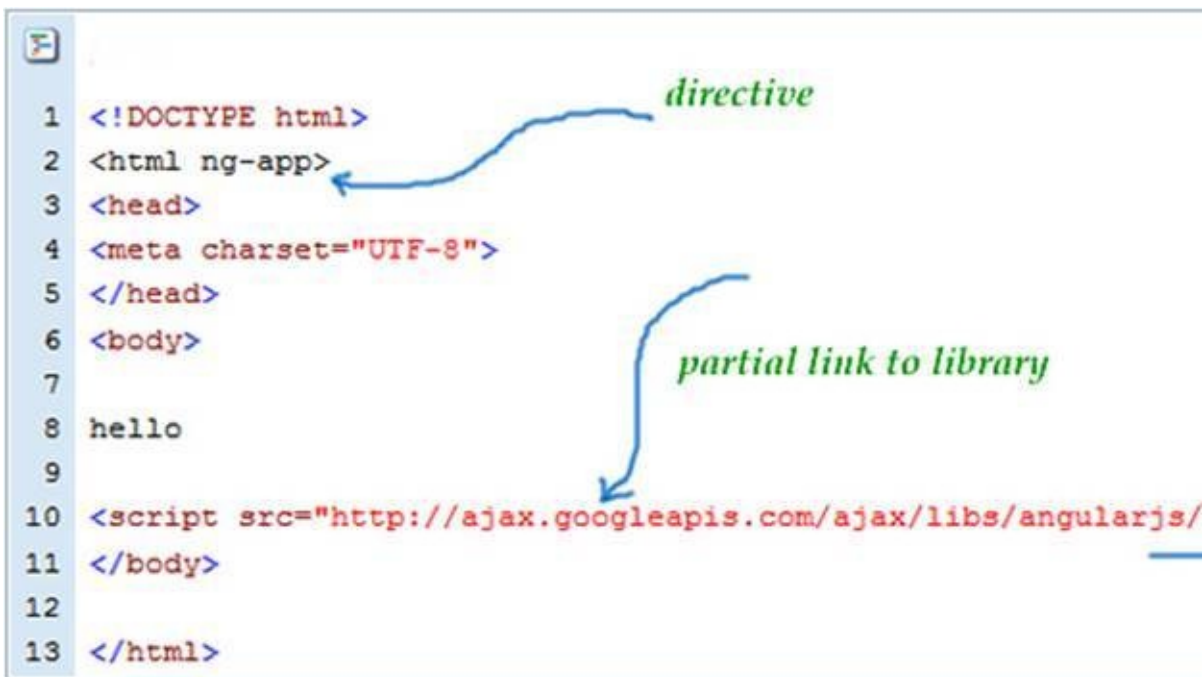


Fig 23 see [raw file](#) | [html](#)

The above image is a basic HTML page with nothing in the BODY but the word *hello*.

Well, that is not true. At the bottom of the BODY (as the last thing to be written *in* the body) we also have a link to the AngularJS library which should be (as of October 2015):

```
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.7/angular.min.js">
</script>
```

You can also substitute the *http* for an *https*.

In the future, to get the latest link version, go to [angularJS.org](#) and **click on Download**. A window will pop up with different ways of accessing the library. Copy the link on the CDN window and paste it in your web page inside of the opening script tag as shown above. For our exercises you don't have to do this step, just copy the raw file (see link below the image) as your starting HTML skeleton page.

The reason to include the link to the library at the very bottom of the BODY container is to make sure that all the HTML elements load into the browser before calling the library. As an alternative we could include the link in the HEAD section like we normally do for CSS, but then we would have to make sure that there were no side effects by calling the library prematurely.

On **line 6** I have included a directive **ng-app**. That will tell AngularJS that everything within the `<html>` and `</html>` tags can use the tools available at Google's AngularJS library. In other words, the whole HTML page has been extended with more functionality. The trick is to know what functionality is available and what we can do with it.

There is a lot to learn about it and I will provide links for further studies after we conquer the *biggest* barrier: **The first step toward AngularJS' world.**

What is a library?

A library is a collection of functions which are useful when writing web applications. They can be called when needed.

What is a framework?

A framework is a particular implementation of a web application. In a framework your code fills in the details. The framework is in charge and it calls into your code when it needs something “app” specific.

What is a directive?

A directive is something that introduces new functionality to the existing standard syntax. It expands the syntax’s capability, it’s a way of teaching an old dog new tricks. By attaching a directive to an existing HTML tag, the behavior of that tag changes by what is written on the Angular’s library under the name of such directive. In some cases directives specify global behavior, while in other cases they only affect a local section, such as a block of programming code.

AngularJS teaches the browser new syntax through constructs called directives.

If you ever built web applications before, Angular will be a total new paradigm. If you are new to web design, AngularJS will teach you modern and best practices and consider yourself lucky because you don’t have to forget the old in order to learn the new.

Yes there is a price to pay at first, you lose flexibility. What you gain in return is consistency, speed of development, and something that works today and tomorrow. You no longer need to know how to build a motor in order to drive the car, but the more you know the better you will take advantage of what the car has to offer. AngularJS is not an easy technology to learn, but if you have made it up to here in this book, then you have gotten the discipline and skill to learn and master AngularJS.

What does ng-app mean?

It just tells the browser to use AngularJS as the root of the page, or as the root of the DIV, depending where we insert the ng-app tag. In this way everything will be relative to AngularJS which provides core functionality to your page.

In this example,

```
<html ng-app>
```

The whole page uses AngularJS as its root.

In this example,

```
<div ng-app>
```

Only this DIV uses AngularJS as its root. The rest of the web is not aware of AngularJS.

We can also name the application we are building by adding the *app* name to the directive:

```
<html ng-app="myApp">
```

This becomes very useful as you will see soon.

Because the application name inclusion on the ng-app directive is very common, you will see the following syntax in many articles out there:

```
<html ng-app="">
```

That's another way of writing a directive with no name, instead of just writing **ng-app** by itself

- **Note:** Do not include a name unless such name exists as a *Module* because AngularJS will not work. We haven't covered Modules yet so, for now, we only use *ng-app* by itself, and it is perfectly fine to do so even in production.
- **Note:** Notice how *ng-app=""* does not have a blank space between quotes. A blank space would be considered a name (even if blank) and a Module under that name would have to be created. Again, let's just use *ng-app* by itself for now.

No more DOM manipulation tasks

The Document Object Module or **DOM** is the map layout of all existing HTML tags or nodes. In order to write a result of a JavaScript program in a web page we need to address the DOM location where we want to implement the scripted result. This has always been a problem due to browser inconsistencies and implementation errors. Then *jQuery* was invented. jQuery is a library of methods used to address the DOM. When properly implemented, jQuery works great. Unfortunately the usage of jQuery has become a problem because many users do not learn JavaScript, they just memorize jQuery solutions and jQuery is manually driven leading to implementation errors.

AngularJS has come to the rescue. The low level implementation will no longer be done by the programmer him/herself; it will be done by AngularJS based on the programmer's expressive desires for a certain outcome. The programmer has become a plumber; it plugs in the correct directive for the desired outcome. This will assure best practices and error free implementations. With AngularJS you are freed from low level manipulation tasks and you can use your extra time to think of solutions instead of code grammar and security implementations.

Of course, if you know jQuery you can still use it with AngularJS, but I suspect that AngularJS will develop more and more ways to avoid some of the jQuery popular implementations.

In AngularJS, the only place where an application touches the DOM is within *directives*. If however you need to access the DOM directly you can still do it through the writing of [custom directives](https://bit.ly/1tYW95Q) | bit.ly/1tYW95Q

Ok, we now know a bit about *directives*. Let's review the meaning of *expressions* so that we can expand our AngularJS topic.

What is an expression?

Do you remember from a previous chapter what expressions were?

An expression is something JavaScript evaluates to a single value.

```
2 + 5;
```

```
// evaluates to 7
```

```
"tony"
```

```
// evaluates to tony
```

```
"hello".toUpperCase();
```

```
// evaluates to "HELLO"
```

and so on...

If, on our web page we have an expression that needs to be passed into AngularJS in order to be evaluated by the JavaScript interpreter, we do so by wrapping it in double braces:

```
{{ this expression }}
```

AngularJS will write the result of a JavaScript evaluation exactly where the braces are located on the web page. This is not an AngularJS invention; it has been implemented by other libraries away before AngularJS. Just take a quick look at [Mustache](#) |

bit.ly/1DxihbF.

Let's play with a couple of examples. It will not be anything practical but it serves to familiarize us with passing expressions to AngularJS.

Note: You still can use jsbeautifier.org as an editor but unfortunately you cannot save your files unless you paste your code onto another plain text editor. NOTEPAD comes to mind. If you've done HTML before you probably already have a favorite editor that you can use.

- 1- First we fetch our sample HTML script:

[raw file](#)

- 2- Paste it onto a local editor like for example Windows NOTEPAD (I also use the free and colorful Programmer's Notepad from pnotepad.org). Save it as an HTML file. I will call my file **ang2.htm**. Make sure you create a new folder just for our examples.
- 3- To test it, go to your file list and double click on the file. Since it is an HTML file it will open in the browser and you should see the word *hello*.
- 4- Now that we know the file works, let's go back to the editor and add an expression. Replace the word hello with the following sentence and expression:

Convert the expression tony to uppercase: {{ "tony".toUpperCase() ; }}

- 5- Save the file, refresh your browser or reopen the file, and you should see the word TONY in upper case.

See the image below for an explanation:



```
1 <!DOCTYPE html>
2
3 <html ng-app>
4
5 <head>
6   <meta charset="UTF-8">
7 </head>
8
9 <body>
10
11   Convert the expression tony to uppercase: {{ 'tony'.toUpperCase(); }}
12
13   <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.2.15/angular.min.js"></script>
14 </body>
15
16 </html>
```

Fig 24 see [raw file](#) | [html](#)

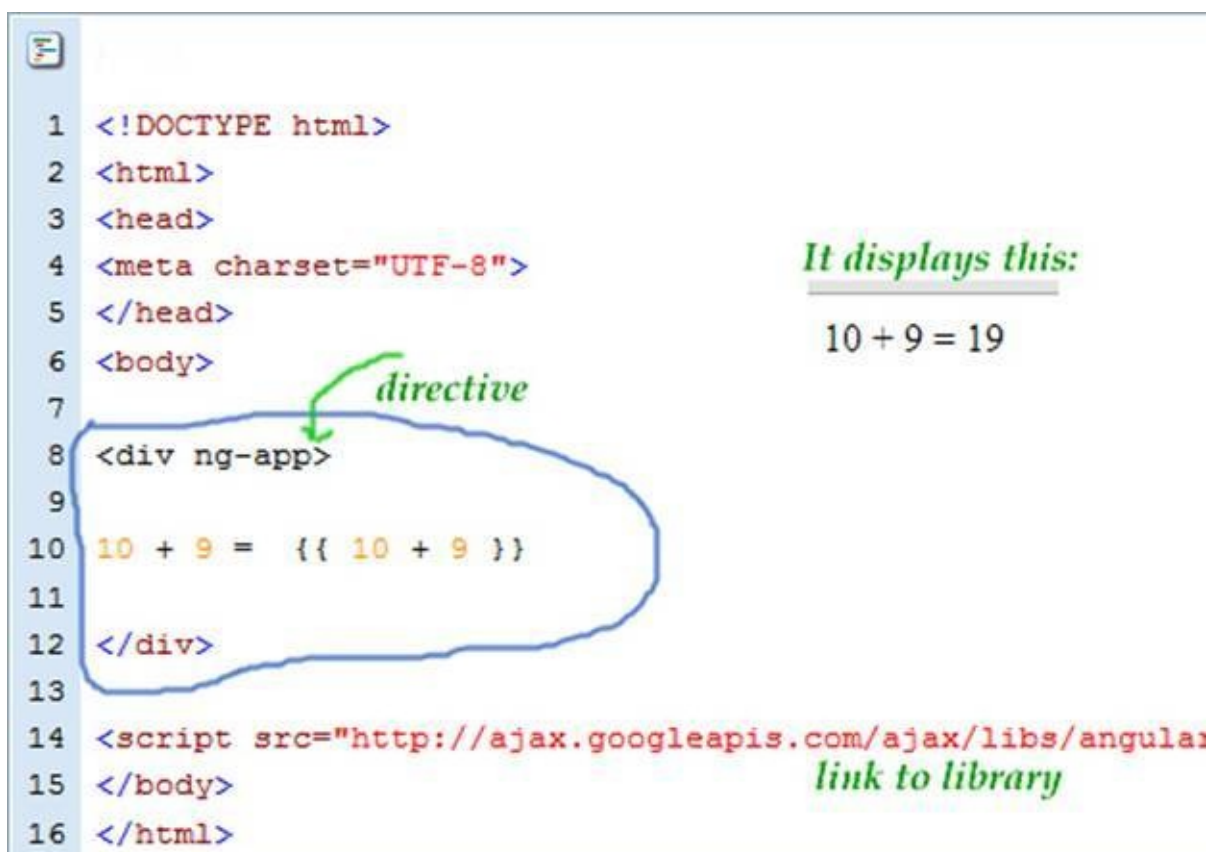
The HTML script is still the same as before. We only replaced the word *hello* with the sentence seen on line 11, followed by `{{ "tony".toUpperCase(); }}`.

Anything written inside of the double curly braces is to be evaluated by JavaScript.

Do you see yourself adding JavaScript evaluation results to your existent HTML pages?

Even if you don't make the whole page an AngularJS app, you could still reserve one paragraph or one DIV for some output you may need to ask AngularJS to process.

On the next image we have the same script but this time I only reserved one little space inside of a DIV for AngularJS processing, instead of declaring the whole web page as an AngularJS app:



```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="UTF-8">
5 </head>
6 <body>
7
8   <div ng-app>
9
10     10 + 9 = {{ 10 + 9 }}
11
12   </div>
13
14   <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.2.15/angular.min.js"></script>
15 </body>
16 </html>
```

directive

It displays this:
10 + 9 = 19

link to library

Fig 25 see [raw file](#) | [html](#)

From line 8 through 12 I declared a DIV and assigned it to AngularJS by including the directive ng-app.

On line 10, I wrote a message in plain text, and then I included $10 + 9$ inside of the double braces. This will instruct AngularJS to have this expression evaluated by JavaScript.

I didn't write any JavaScript code. AngularJS took care of the internal process. This is programming at a higher level of abstraction. Do we still need to know JavaScript? Of course we do! Your knowledge of JavaScript will become necessary when we go beyond this simple testing.

Do I always have to include my expressions on my web page?

No, we are doing it just for testing and understanding purposes before introducing the module and controller. But yes, you can always do this in production and it should work for simple stuff.

ng-init

Let's introduce another directive so that we have more tools to play with.

- An **ng-app** directive initializes the AngularJS application.
- An **ng-init** directive initializes some data to be used on the application.

Let's see how *ng-init* works in its simplest form:

- 1- Open your basic HTML script and modify the HTML tag to contain a second directive as follows:

```
<html ng-app ng-init="city='Harrisburg'; state='Pennsylvania'">
```

- 2- Then in the body write the following:

```
Do you know {{city}} is the capital of {{state}}?
```

- 3- Save it as **ang5.html** and test it.

You should see the following sentence on your browser display:

Do you know Harrisburg is the capital of Pennsylvania?

```
3 <html ng-app ng-init='city='Harrisburg';state='Pennsylvania''>
4 <head>
5 <meta charset='UTF-8'>
6 </head>
7
8 <body>
9
10 Do you know {{city}} is the capital of {{state}}?
11
```

Fig 26 [raw file](#) | [html](#)

Do you see what happened?

We initialized two variables by using **ng-init**: *city* and *state*, and then we were able to process them via two expression tags on the HTML output. There was no need to address the DOM with complicated scripting. There are better ways to do this but right now the purpose is to become familiar with the AngularJS' basic tools.

By initializing the arguments *city* and *state* on the HTML tag they became available to use on the web page. However these variables are normally placed on a separate sheet.

ng-bind

Just to illustrate the purpose of the double curly braces, let's write the same script in a different way: by using a directive called *ng-bind*.

- 6- I'm only doing it for variable *city* and not for variable *state* just to show how both will work:

Do you know `` is the capital of `{{state}}`?

And the display will be:

Do you know Harrisburg is the capital of Pennsylvania?

Actually, AngularJS always replaces the curly braces with *ng-bind* in its internal process. If you don't mind writing the extra SPAN tag this actually works a bit faster. When you refresh the page (*CTRL f5* on Windows) and focus on the output, you will see that Harrisburg shows up immediately, whereas Pennsylvania will first show the braces and then it converts to the proper word. It happens very fast but you might be able to catch it.

What does ng-bind do?

It binds *data source to target*. No need to play the `getElementById` DOM game anymore.

There is a lot more to know about AngularJS binding but we need to go slow and just grasp enough to understand what comes next. It is like that famous expression "[Rome wasn't built in a day](#)".

ng-cloak: Avoiding display flickering

When using the curly braces `{{ }}` instead of `` you may notice a quick flicker on the display because the curly braces have not yet been evaluated by AngularJS.

If that becomes a problem for you, include the link to the library in the HEAD section instead of the BODY section. This way, the curly braces will be evaluated before the browser displays the document. Including the link in the head may have some side effects, depending on the complexity of your page, hence the reason why more and more developers are moving library links to the bottom of the page.

If you are embedding AngularJS on a page but you have no access to the HEAD script, then you may use ***ng-bind*** instead of the curly braces, or you could use **ng-cloak**.

Since *ng-cloak* involves CSS, I am not going to describe here. Please check the following link for a full description of this implementation:

docs.angularjs.org/api/ng/directive/ngCloak. | bit.ly/1pcTqjF

5.4 Lab work 14

Part A: Initializing an object and outputting its contents in a paragraph

Speaking of binding, let's "bind" to memory all we have covered so far by doing a few exercises.

This is actually a familiar exercise from our JavaScript lab sessions. Here we are going to *initialize* an object called **colors**. The object contains the following properties:

a)

```
"green": "green is the color of balance and growth.",  
"blue": "blue is the color of trust and peace.",  
"indigo": "indigo is the color of intuition"
```

Then we will get the following display on the browser:

b)

The color green is the color of balance and growth.

The color blue is the color of trust and peace.

The color indigo is the color of intuition.

The directives we are going to use are:

ng-app, **ng-init=' '** and the double brace **{{ }}** expressions where the output display evaluation will take place.

Notice how ng-init is using *single quotes*. This is because the object **colors** we are going to insert inside of the quotes have in itself double quotes. By using single quotes for the *init* we assure that JavaScript does not get confused and abort the expression prematurely.

Please refer to the following link if you want to copy the opening HTML tag with both *ng-app* and *ng-init* plus the object code block: [this raw](#)

Try not to copy/paste it before you attempt to create your own version of it, or take just a quick look to refresh your mind and then try doing it yourself.

Let's start. (On the next page please find step by step instructions).

- 1- Create a basic HTML page (as we've done before).
See the starter [raw file](#)
If you are not using the starter raw file make sure you do the following:
 - a) In the HTML opening tag insert a directive to allow AngularJS to manage your web page.
 - b) On the bottom of the BODY container insert the URL for the AngularJS library.
- 2- After the *ng-app* directive, but still inside of the HTML opening tag, add a **ng-init=' '**
inside of the ' ' insert an object named **colors** with the properties described on the previous page under exhibit *a*.
For your convenience I have included a file of the raw text to be inserted as an object from which you can copy/paste:
[raw file](#)
- 3- In the BODY section add three paragraphs, one for each color.
See the first sample for the color green below:

```
<p>The color {{colors.green}}</p>
```
- 4- Save it as **ang7.html** or any other name you prefer.
Open it on your browser.
It should look like this: [html](#)
- 5- A whole script and further explanation can also be seen at the [forum](#) | bit.ly/1rk2jLx

Part B: Initializing variables and outputting a calculation

In this new exercise we want to initialize two variables,

length=9 and **width=3**

and then output the following paragraph:

If the length is 9 and the width is 3, the perimeter is 24.

Please remember that where it says 9 and 3, the dynamic markup should be written like

`{{length}}` and `{{width}}`

which is later replaced by the value assigned to each variable.

As for the total 24, it should be the perimeter of the rectangle. You can use your own formula.

I am going to use *length * 2 + width * 2* as my expression.

After finishing your script please compare it with mine here:

[forum](#) | bit.ly/1rk2KW8 | [raw file](#) | [html](#)

END OF LAB

5.5 Going a step further into modularity

Assigning a module name to ng-app

Modular programming is a design technique that emphasizes *splitting the functionality of a program into independent modules* and in such a way that each module contains everything necessary to execute only one aspect of the desired outcome.

A module is a container for the different parts of an *app* including related controllers, services, filters and directives.

Your web page *app* is a module.

In our web page when we declared *ng-app* on the HTML tag, we are actually creating a module (ours has been an *unnamed* module). The reason why we have not given a name to our test web page (or module) is because, once named, it stops working until we configured the *module interface*, which means to officially register this module with AngularJS by writing a declaration inside of a `<script></script>` block.

That's why we kept our AngularJS declaration as simple as possible by only declaring:
`<html ng-app>`

However, it is a common best practice to name the web page by giving it a module name like for example:

```
<html ng-app="nameOfTheModule">
```

`nameOfTheModule` is the identity of this web page in reference to AngularJS and you use any name you want.

Besides being a best practice to name a module, it is actually mandatory if you make your page more complex beyond what we have done so far. Of course, for quick testing purposes we can just create an unnamed ng-app like this:

```
<html ng-app="">
```

Be careful here, if you use quotes but no name, make sure there is no space between quotes because that space will confuse AngularJS to think the module is named and you will get an error because there is no such name on file.

My favorite test syntax is still *ng-app* (or *data-ng-app* for HTML5 validation compatibilities). I only use the extension when I really have a Module name to include, which is what we are going to do next.

Configuring a Module interface

Once we assign a name to our *ng-app* we have to configure the Module.

Take for example the last exercise we did, (**Part B: Initializing variables and outputting a calculation**). This exercise is simple enough to help us understand how to configure the app name module:

```
1 <!DOCTYPE html>
2
3 <html ng-app ng-init='length=9; width=3'>
4
5 <head>
6   <meta charset="UTF-8">
7 </head>
8
9
10 <body>
11
12   <p>If the length is {{length}} and the width is {{width}},
13   the perimeter is {{length * 2 + width * 2}} .</p>
14
15   <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.7/angular.min.js">
16   </script>
17 </body>
18
19 </html>
20
21 </html>
```



Fig 27 (no app module name yet)

In the picture above, the *ng-app* has no name and it works just fine.

If however we insert a name to the `<html ng-app>` such as for example

`<html ng-app="myFirstAppModule">`

the page will stop working until we configure the Module interface.

One way to configure the app module is to open a new `<script></script>` container just below the link to AngularJS at the bottom of the BODY of your HTML page.

In the script container we then assign a *function call* to some variable (like for example *x*). This particular function or method comes from the Angular library and its name is **module**. Example:

```
var x = angular.module();
```

An image of this assignment can be seen below but first let me just finish the explanation of this assignment.

After we assign the `module()` to variable *x* we then include the name of the **app** as the first argument of the method `module()`. The second argument is an empty array and it should look as follows:

```
var x = angular.module("myFirstAppModule", []);
```

Traditionally the `x` variable is named after the module and we would have two `myFirstAppModule` words in this declaration (a repetition of these terms would not conflict since they are on different areas of the code but it would help knowing what the variable `x` was used for). However, since the name match is not mandatory I have called it `x` in order to purposely illustrate where the module name goes and the fact that it has nothing to do with the variable.

This declaration is good enough to bring our ng-app back to life for now.

Please study the following image carefully.



```
1 <!DOCTYPE html>
2
3 <html ng-app="myFirstAppModule" ng-init='length=9; width=3'>
4
5 <head>
6   <meta charset="UTF-8">
7 </head>
8
9
10 <body>
11
12   <p>If the length is {{length}} and the width is {{width}},
13   the perimeter is {{length * 2 + width * 2}} .</p>
14
15 <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.7/angular.min.js">
16 </script>
17
18 <script>
19   var x = angular.module("myFirstAppModule", []);
20 </script>
21
22 </body>
23
24 </html>
```

Fig 28 (after giving ng-app a name we then register the name as a module)

Try it yourself. Get your original file ([original raw file](#)), add a name to your ng-app and register the module with Angular by writing the declaration at the bottom of the BODY. Notice how the name of the module is introduced on the function `module()` within quotes. It is passed in as a string.

If you need help please look at my own file here:
[final raw file](#).

Further explanation:

Look at the variable `x` declaration. `x` gets assigned to a function that belongs to object angular. This function which is called ***module()*** takes two arguments. The first argument is a string containing the name of the module we are declaring. Then, separated by a comma, we have an empty array. They both need to be declared.

The purpose of the array is to include future dependencies for the module being created. Sometimes we want to use other sub modules in order to enhance this particular module.

Since using other related modules is very common, the second argument was hardwired and it needs to be included when we declare a new module. In our case we just write the array as an empty array and that will work for us. If however you skip the [], the module will not work. We must place those two brackets in there.

5.6 Preparing to separate data from presentation

Creating a controller

Congratulations, we are moving steadily into a practical presentation of AngularJS. Please be patient, we're almost done.

Up to this point we have been initializing dynamic data with *ng-init*. This may come handy sometimes but it is not an ideal implementation.

```
1 <!DOCTYPE html>
2
3 <html ng-app="myFirstAppModule" ng-init='length=9; width=3'>
4
5 <head>
6   <meta charset="UTF-8">
7 </head>
8
9
10 <body>
```




Fig 29 (ng-init will be replaced by a controller at the bottom of the page)

We need to move towards a separation of files residing in properly labeled folders, but we are not there yet. I mean, we should be able to visualize the whole picture in just one page before we start splitting the code into different files, and that's what we are doing in this course: learning the basic parts of an Angular app. If we split code into folders at this level of understanding it just adds unnecessary complication and we do not want to go there right now.

What we're going to do next is to stop writing our data on the top of the page with *ng-init* and move the data object into the bottom of the BODY section (inside of the same script we inserted our module declaration). Our JavaScript code will be written inside of a **controller**, which is a special AngularJS function, (a method or property of the module we have just created).

So, right now we are going to split the body of the web page into the following conceptual areas:

- a) **The standard HTML markup** and its regular data, the one that does not need any influence from AngularJS. This data could actually be anywhere across the page because dynamic data will be wrapped into code blocks as you will see.
- b) **The area where we want AngularJS to display some dynamic data.** This area is known as a **view**. AngularJS injects into the page a number of mini dynamic *views* that blend along with the rest of the HTML presentation. It could be just one *view*, or many *views* depending on what you really want to do. Think of it as a script that we *embed* into the page, just like we do with pictures or with videos from YouTube.
- c) **The area where we create JavaScript code.** We call this area a **controller**. The area is surrounded by the tag `<script></script>` just like the link to the framework is. The idea of the controller at this point is to store data and functionality like we did with *ng-init*. The controller is a function and inside of the function we have our dynamic data. Remember, we use functions to store code that can run when called upon.

We are going to remove the *ng-init* from our previous exercise and insert the data into the controller but first let's look at the basic structure of a controller. Let's start with the image of an empty controller:



```
HTML
1 <script>
2   var x = angular.module("myFirstAppModule", []);
3   x.controller('myFirstController', function($scope) {
4       //code goes here
5   });
6 </script>
```

Fig 30 (controller is a child of module x. JavaScript code goes in the controller)

The controller function is a child of module which in our example is assigned to variable *x* in order to make it easy to implement the controller (*x* becomes a shortcut to the whole module declaration). So, the module *myFirstAppModule* is assigned to *x* and from variable *x* we extract a *controller* [line 3]. Inside of the controller method we have two parameters. The first parameter is the name we are giving to the controller and in a string type format (I'm calling it 'myFirstController'); the second parameter is an AngularJS object called *\$scope*. *\$scope* is an Angular object that keeps track of all data and properties we are going to send to the web page as we will see shortly. Notice how all this code is inside of the SCRIPT tag [lines 1 and 7] where the *ng-app="myFirstAppModule"* module was declared.

AngularJS *magically* makes available to the HTML **view** (the view assigned to this controller) whatever code and data we write inside of the controller and assign to object

\$scope.

We could say that **\$scope** is the glue between the *controller* and the page *view*. The controller wants nothing to do with the page view because it is supposed to be generically written. I mean, we could have several different views using the same data from the controller: one view for a desktop screen, another view for a mobile phone, etc. It is up to *\$scope* to interface with the external world of views and AngularJS provides such functionality in the background.

Before we do the exercise let me summarize some concepts to help you further understand what is going on.

\$scope

\$scope is an object that maps the whole page (the DOM) and creates properties that can be addressed. In a sense it is like the DOM in parallel. Later you will see that we add \$scope to our controller and then can add variables and functions into \$scope which in turn will be available to the web page app. (Note: The **\$** looks like an “S”, right? that reminds us that \$scope is an AngularJS service).

View

View is the area of the application that will be placed on display. Just like images, videos or other embedded text, we could have several views on one page and they may be hardwired or linked from a different source. Contrary to the HTML standard presentation, *view* data is not persistent which means that it varies based on the logic provided by \$scope (coming from the assigned controller), or the behavior of a user interacting with the page. A view is created by standard HTML elements in combination with AngularJS' enhancements from data binding and directives and curly braces `{{}}`.

- Any complex logic should be placed in the controller and let *\$scope* manage the feeding of this logic into the view. This does not mean that a view can't have any logic; simple expressions can be placed in the view as we have done before during our first exercises.

Controller

The controller is where we program the JavaScript code to be executed when the page runs.

The controller's purpose is to provide the logic required by AngularJS in order to initialize the object \$scope, which is the object that maps the webpage like a DOM.

Before using a controller with \$scope, we were using a simple ***ng-init*** to save and initialize our data but that was a very limited implementation. \$scope is a more powerful object and the most appropriate way to declare our properties and methods.

This next image shows how and where we implement the data we previously assigned to ng-init:

```

1 <script>
2
3   var x = angular.module("myFirstAppModule", []);
4
5   x.controller('myFirstController', function($scope) {
6       $scope.length = 9;
7       $scope.width = 3;
8   });
9
10 </script>

```

Fig 31

As you can see, all the JavaScript code is treated as a property child of \$scope. AngularJS will grab both *length* and *width* and saves them as part of \$scope (remember, \$scope will manage the DOM which is how and where things are displayed).

There is only one thing left. We now have an *app name* and we have a *controller* with some JavaScript variables to be processed. But how does AngularJS know where to display those variables in the paragraph when we open the page?

As a reminder, this is what we had in the HTML BODY of the old exercise:

```

HTML
1 <p>If the length is {{length}} and the width is {{width}},
2 the perimeter is {{length * 2 + width * 2}} .</p>

```

Fig 32

The answer is, we should wrap the whole paragraph in a DIV and assign the DIV to the controller which is called *myFirstController*:

```

1 <div ng-controller="myFirstController">
2     <p>If the length is {{length}} and the width is {{width}},
3     the perimeter is {{length * 2 + width * 2}}.</p>
4 </div>

```

Fig 33 (we wrap the paragraph in a div assigned to myFirstController)

Your turn: Add a controller to your exercise

At this time let's modify our previous exercise to add a controller to it.

Here's the original [raw file](#) as we left it after adding the module *myFirstAppModule*.

- 1- Add the controller to the script container, one line below the module declaration as seen on figure 30.
- 2- Assign length and width to *\$scope* as seen on figure 31 and then remove the *ng-init* from your page.
- 3- Wrap your paragraph in a DIV and assign the DIV to the controller as seen on figure 33.
- 4- Test your HTML Angular app. Here's my own version of it:
[raw file](#) | [html](#).

We now have a complete simple Angular app.

Alternate way to code length and width

Another way to include the *length* and *width* on our exercise is by making them properties of some object like for example *myPerimeter* and then write them inside of the controller as follows:

```
$scope.myPerimeter = {  
  "length": 9,  
  "width": 3  
};
```

Instead of using `$scope.length =9;` and `$scope.width =3;`

In the HTML **view** we would write them as

```
{{myPerimeter .length}},{{ myPerimeter .width}}  
and the formula: {{myPerimeter.length * 2 + myPerimeter.width *  
2}}
```

Let's keep practicing by creating a few cool mini projects.

Create a program to convert Fahrenheit to Celsius

We are *not* going to create this program from scratch because we have done earlier on **Lab work 6** when learning JavaScript.

See [forum](#) | bit.ly/1Dp2SKk
[raw file](#) | bit.ly/1ruKrzi.

What we are going to do is to create an *app* to display the result on an HTML page.

The idea is to practice setting up AngularJS on the page as well as practicing ways to include code implementations in the controller.

Here's the recipe menu for my sample:

Module name: *myAppModule*

Controller's name: *myController*

Variable to display the result: *result* (hint: it will be *\$scope.result* in the controller).

Sentence to be dynamically displayed:

100 degrees Fahrenheit is 37.778 Celsius.

NOTE: make sure you change the original *f2c(212)* to **f2c(100)** when calling the function.

Are you ready for the challenge?

Please do the exercise yourself. Here's a summarized guideline:

- 1- Create your HTML skeleton.
Here's our starter file: [raw file](#).
- 2- Link to AngularJS (if you're using the starter file this is already done).
- 3- Add the ng-app directive (already done) and name it myAppModule.
- 4- Below the link to AngularJS add a new Script container and declare your module in there. (I'm assigning mine to variable x).
- 5- Add your controller below the module declaration. The name of the controller is myController and it belongs to variable x. Also in its parameters it shows a string with the name of the controller and after separating with a comma, a second parameter introduces a function which passes in \$scope.
- 6- Add the Fahrenheit to Celsius conversion function to the controller.
Here we have a few alternatives. Let me explain:
 - a) First get a copy of the original JavaScript script here: [raw file](#).
 - b) We need to replace *console.log* with **return** because we are catching the result and sending it to \$scope so that it can be displayed.
 - c) Now we have two options (see steps c through f):

We could assign function **f2c** as a property of *\$scope* this way:

```
$scope.f2c = function(f){return ((f-32) * 5 / 9);}
```

- d) Then, when we get to the HTML page view, we can implement the result this way:
`{{ f2c(100) }}`
- e) **OR** (and this is a better way because it avoids using JavaScript in the HTML view), we can insert the original function inside of the controller (except for the `console.log`), and then assign to `$scope` a variable such as for example **result**, with the result of a function call.
- ```
x.controller('myController', function($scope) {
 function f2c(f) {
 return (f-32) * 5 / 9;
 }
 $scope.result = f2c(100);
});
```
- f) If we use option (e), we then apply variable result in the view like this:  
`{{ result }}`
- 7- In the BODY of the HTML page create a view by adding a DIV container.  
Link the opening DIV tag to `ng-controller="myController"`
- 8- Inside of the DIV container add the paragraph with the following message:  
`<p>100 degrees Fahrenheit is {{YOUR CODE HERE}} Celsius.</p>`
- 9- Where it says YOUR CODE HERE, replace it with option (d) or option (f) based on the way you programmed your controller's JavaScript as stated in (c) or (e).
- 10- Save and test your app. The display result should be  
100 degrees Fahrenheit is 37.77777777777778 Celsius

When you're done, compare it with my result here:

[forum](#) | [bit.ly/1ruKCKT](https://bit.ly/1ruKCKT)

[raw file](#) |

[html](#) |

Suggestion: Try to convert some of the other JavaScript exercises into Angular. You may not get a perfect solution but you will get plenty of practice.



## Placing the controller on a separate file

Now that we have logically separated the controller and view units (both part of the module), we can safely place the *controller* and *module configuration* on a separate file. This is by no means the end of the story, perhaps just the beginning.

On simple applications we can just leave it on the same page as we have done earlier. However, as versatile developers we need to practice separation of concerns in order to get ready for bigger things.

It's pretty easy. Let's take for example the exercise we just finished (Fahrenheit to Celsius):

[raw file](#) |

- 1- Save it under a different name, like for example **f2cfinal.html**

Now we can edit the file:

- 2- Copy and cut (delete) all the contents inside of the `<script></script>` where the controller and module configuration are. Just leave the SCRIPT tags by themselves so that we can add a link to the separate file.
- 3- Paste the contents on a new file and save it as **mycontroller.js** (or any other name). Notice the **.js** extension. Make sure the file is on the same folder as *f2cfinal.html*.
- 4- Go back to *f2cfinal.html* and in the opening SCRIPT **tag** that belonged to the controller, type the path to the new file which should be the following including the script tags:

```
<script src="mycontroller.js"></script>
```

- 5- Save and test.

Compare with my result here:

[raw file](#) |

[.js file](#) |

[html](#) |

## Additional information

For updates and further exploration of controllers (aside from what is covered in this eBook) please visit the following post at JSplain.com: [Notes on controllers](#).

## The ng-model

AngularJS can sometimes be really simple.

Please take a look at the following script based on the JavaScript exercise we did on Part C:

[Fahrenheit to Celsius / to Fahrenheit conversion](#) | [bit.ly/ZQgTRW](https://bit.ly/ZQgTRW)

Do you still remember how we did the JavaScript raw version of this algorithm?

To refresh your mind see the *third* example shown in the forum:

[Lab 6](#) | [bit.ly/1Dp2SKk](https://bit.ly/1Dp2SKk)

Now, back to the first link on the top, it looks pretty cool and it is being displayed on a web page. When we change the value inside of the text box, it converts the value to Celsius or Fahrenheit automatically.

Let's take a look at the script to see what it was done to it. It is much simpler than our original JavaScript version from lab 6.

In the forum you will find a **live demo** and the script right below it. Please read the explanation for each line of code:

[forum board](#) | [bit.ly/ZQieYU](https://bit.ly/ZQieYU).

After reading the description in the forum and play with the demo a bit, please return back here for some lab work. We are going to do this project in less than 10 minutes, promise!

## Lab work, creating the conversion script

Did you notice there is no *controller* or *module* declared for the temperature conversion program?

AngularJS can many times save us hours of work and it is getting better as they improve their game. For the job at hand all we need are a few tools (and an introduction to the *ng-model* directive):

- a) A link to the library
- b) An ng-app directive
- c) An ng-model directive.
- d) A few double curly braces
- e) And two HTML input boxes.

### Ready to begin?

- 1- Let's get our basic HTML script:  
[raw file](#) |
- 2- Save it as ***conversions.html***
- 3- In the BODY, replace *hello* with one DIV containing two other DIVs inside it:  

```
<div>
<div> </div>
<div> </div>
</div>
```
- 4- If you are not using my given starter file, make sure to include an ***ng-app*** directive on your opening HTML tag (or on a DIV that wraps the view).
- 5- On the top most opening DIV tag from step 3, declare an *ng-init* to initialize two variable as shown below (the *init* is because we are not going to use a controller in this project):  

```
<div ng-init="f=32;c=0">
```

At this point we should have the following on the HTML page:

- a) The *link* to AngularJS' library ( it came with the skeleton script),
  - b) The *ng-app* directive somewhere on the page,
  - c) And the *ng-init* to introduce variables *f* and *c* to AngularJS.
- 6- Just for testing purposes let's write some curly braces inside of the DIV container like this (any of the DIV containers):

```
{{f -c + f}}
```

Then save it and open it on your browser.

- If you see this: `{{f -c + f}}` on display, you have a problem. Inspect your code to see what is missing or misspelled.
- If you see the number **64** displayed, then your script is working so far.

Once your script is working properly, remove the curly braces and continue the project. Early testing is always a good idea in order to save loads of time and headaches later. We should test in gradual stages before we move on to the next implementation phase. This way we always know where the problem might reside.

- 7- The next step is to design the first **input box**.

Inside of the container created by the second pair of DIVs, write an input box of type *number*.

(About input boxes: [HTML input](http://bit.ly/1qy2lvA) | [bit.ly/1qy2lvA](http://bit.ly/1qy2lvA) )

Also include a description on the left side. Here's my example:

```
Fahrenheit to Celsius: <input type="number">
```

NOTE: You may be accustomed to use self-closing tags like this `/>`. It works either way. With the advent of HTML5 I prefer to write without self-closing.

8- Great, now let's add an **ng-model** to the *input* box and assign it to variable *f*:

Fahrenheit to Celsius: `<input type="number" ng-model="f">`

If you have read my explanation about *ng-model* on the *forum demo* you will know that the directive **ng-model** is an object that saves the data inputted by the user, and links it (or binds it) to the variable assigned to this ng-model, the `= "someName"` part of this declaration (sort of what `prompt()` does in JavaScript when is assigned to a variable). In this case it is variable *f* which although it defaults to 32, it can be changed by the input data the user enters on this text box and thanks to the binding provided by **ng-model**.

- Do not confuse these terms: *model* versus *module*.

*Module* is the complete application. *Model* is a little object that saves data and binds it to some variable. There could be hundreds of little models in one Module.

9- Let's finish this line by adding the *curly double braces* to the right of the HTML input box.

Inside of the braces include the formula to convert from Fahrenheit to Celsius:

```
{{ (f-32) * 5 / 9 }}c
```

NOTE: The little **c** on the right is just a character to tell the user that this result is in Celsius. It has nothing to do with the variable c previously declared.

The whole line should look like this:

```
Fahrenheit to Celsius: <input type="number" ng-model="f"> {{ (f-32) * 5 / 9 }}c
```

10- Save it and test it.

At this time all your DIVs should look like the following:

```
<div ng-app ng-init="f=32;c=0">
```

```
<div>
```

```
Fahrenheit to Celsius: <input type="number" ng-model="f"> {{ (f-32) * 5 / 9 }}c
```

```
</div>
```

```
<div></div>
```

```
</div>
```

11- Now, let's write the input box for the conversion of **Celsius to Fahrenheit**.

On the next DIV container write the following code:

```
<div>
Celsius to Fahrenheit: <input type="number" ng-model="c"> {{c * 9 / 5 + 32}}f
</div>
```

12- Test it, test both conversions.

The final result can be seen on the

[Forum board](https://bit.ly/ZQieYU) | [bit.ly/ZQieYU](https://bit.ly/ZQieYU).

**Would you like extra practice?**

You can add an extra box and initialize another variable like for example k:

**Here are some possible conversion formulas:**

- a) Kelvin to Fahrenheit:  $(K - 273.15) * 9/5 + 32$
- b) Kelvin to Celsius:  $K - 273.15$





## More Project Ideas

For more project ideas please visit the following link on the forum board created for this book

[Project Ideas In Angular](#)



## Directives summary and resources

As a review, here are some of the directives we have covered, along with other useful directives and their respective official page link.

I will continue to explain these and other directives by posting examples on the JSplain forum for further practicing:

[Forum thread starter](#) | [bit.ly/1roi5qc](https://bit.ly/1roi5qc)

### For Modules/Controllers:

- [ng-app](#), or *data-ng-app* or *ng-app*=“*appName*”
- [ng-controller](#)

### For Data Binding:

- [ng-init](#)
- [ng-model](#)
- [ng-bind](#)

### For DOM Traversing:

- [ng-hide](#)
- [ng-show](#)
- [ng-view](#)
- [ng-repeat](#)

### For Events:

- [ng-mouseenter](#)
- [ng-click](#)
- [ng-keypress](#)

For a quick sample of a module and controller configurations see one of our exercises here:

[Forum](#) | [bit.ly/1ve4e66](https://bit.ly/1ve4e66).



## In conclusion

Thank you for staying with me until the end of this project. I always feel a bit emotional when I get to this part.

There is so much I haven't written about JavaScript and AngularJS, but we need to take a break in order to digest all we have covered in this book.

It is my belief that I have provided the necessary boost for you to feel empowered and continue your own journey, and if you ever have a question that I may be able to help, I'm only a click away on the [new forum platform](#) purposely created for this project.

Please take the time to reread this book and practice all the exercises again.

If you feel like expanding your JavaScript understanding, my other [eBook](#) is all about the JavaScript library methods, objects, closures and memory management.

As for AngularJS, it is a whole new world and there is so much to write about it. We have covered the hardest part of learning to use AngularJS, the very initial concepts.

**Update:** There is a new supplemental exercise eBook if you want to do more AngularJS exercises. Please take a look:

[AngularJS Supplement: Easy Projects on Filters and Directives](#)

Thank you dear reader, I hope this has been time well spent.

Tony de Araujo

New Jersey, USA

[Amazon Profile](#) | [amzn.to/1wMqHrb](https://amzn.to/1wMqHrb).



## Can you do this for me?

Word-of-mouth is crucial for any author to succeed. If you enjoyed the book, please consider leaving a comment at Amazon. Even if it's only a line or two, it would be a huge help.

Here's my profile at Amazon from which you can find this book:

- US: <http://www.amazon.com/Tony-de-Araujo/e/B00D7V08WY>
- UK: <http://www.amazon.co.uk/Tony-de-Araujo/e/B00D7V08WY>
- DE: <http://www.amazon.de/Tony-de-Araujo/e/B00D7V08WY>
- FR: <http://www.amazon.fr/Tony-de-Araujo/e/B00D7V08WY>
- NL: <http://www.amazon.nl/JavaScript-Plain-Language-Self-Study-AngularJS-ebook/dp/B00NQERIEI/>
- IN: <http://www.amazon.in/gp/product/B00NQERIEI>
- AU: <http://www.amazon.com.au/gp/product/B00NQERIEI>
- MX: <http://www.amazon.com.mx/gp/product/B00NQERIEI>
- JP: <http://www.amazon.co.jp/gp/product/B00NQERIEI>
- ES: <http://www.amazon.es/gp/product/B00NQERIEI>

I may not have a profile in all global areas but you can always find my eBook at your local Amazon.

Thank you so much.

*Tony de Araujo*

Technical Instructor,

USA

.





## Errata, Contact, Updates

[JSplain.com](http://JSplain.com) is a forum dedicated to the material covered on this eBook.

In there you will find a section for announcements and updates:

[jsplain.com/javascript/index.php/Board/8-JSPL-Book-Announcements](http://jsplain.com/javascript/index.php/Board/8-JSPL-Book-Announcements)

[bit.ly/1redx6M](http://bit.ly/1redx6M)

Please register if you would like to be notified of any update or enhancements.

*(The link to login or register is located on the top left of the forum board)*

*“For the things we have to learn before we can do them, we learn by doing them.”*

— Aristotle, The Nicomachean Ethics