# JavaScript\_GettingStarted

|  |  |
| --- | --- |
| [New Tools](#_New_Tools_1) | [Introductory Project](#_Introductory_Project) |
| [Sample Website](#_Sample_Website) | [Adding JavaScript Code to a Web Page](#_Adding_JavaScript_Code) |
| [Working with JavaScript Files](#_Working_with_JavaScript) | [Formatting Code](#_Formatting_Code) |
| [Detecting and Fixing Errors](#_Detecting_and_Fixing) | [Case Sensitivity](#_Case_Sensitivity) |
| [Commenting Code](#_Commenting_Code) | [What Is a Variable?](#_What_Is_a) |
| [Declaring Variables](#_Declaring_Variables) | [Using let to Declare Variables](#_Using_let_to) |
| [Naming Variables](#_Naming_Variables) | [Common Errors Using Variables](#_Common_Errors_Using) |
| [Changing Variable Values](#_Changing_Variable_Values) | [Constants](#_Constants) |
| [The var Keyword](#_The_var_Keyword) | [Numbers](#_Numbers) |
| [Operator Precedence](#_Operator_Precedence) | [Number Precision](#_Number_Precision) |
| [Negative Numbers](#_Negative_Numbers) | [Strings](#_Strings) |
| [Manipulating Strings](#_Manipulating_Strings) | [Converting Strings and Numbers](#_Converting_Strings_and) |
| [Boolean Variables](#_Boolean_Variables) | [null and undefined](#_null_and_undefined) |
| [Objects and Symbols](#_Objects_and_Symbols) | [Conditionals Using if()](#_Conditionals_Using_if()) |
| [Truthy and Falsy](#_Truthy_and_Falsy) | [if … else](#_if_…_else) |
| [Comparing === and ==](#_Comparing_===_and) | [The Ternary Operator](#_The_Ternary_Operator) |
| [Block Scope Using let](#_Block_Scope_Using) | [Looping with for()](#_Looping_with_for()) |
| [Looping with while()](#_Looping_with_while()) | [Looping with do … while()](#_Looping_with_do) |
| [Function Basics](#_Function_Basics) | [Function Expressions](#_Function_Expressions) |
| [Function Within Object](#_Function_Within_Object) (note by Stephen Cole, not Pluralsight) | [Passing Information to Functions](#_Passing_Information_to) |
| [Passing Information to Functions by Reference or by Value](#_Passing_Information_to_1) (note by Stephen Cole, not Pluralsight) | [Function Scope](#_Function_Scope) |
| [Using Functions to Modify Web Pages](#_Using_Functions_to) | [Object Properties](#_Object_Properties) |
| [Object Methods](#_Object_Methods) | [Passing Objects to Functions](#_Passing_Objects_to) |
| [Standard Built-in Objects](#_Standard_Built-in_Objects_1) | [The Document Object Model (DOM)](#_The_Document_Object) |
| [Styling DOM Elements](#_Styling_DOM_Elements) | [Detecting Button Clicks](#_Detecting_Button_Clicks) |
| [Showing and Hiding DOM Elements](#_Showing_and_Hiding) | [Creating and Initializing Arrays](#_Creating_and_Initializing) |
| [Accessing Array Items](#_Accessing_Array_Items) | [Manipulating Arrays](#_Manipulating_Arrays) |
| [slice() and splice()](#_slice()_and_splice()) | [Array Searching and Looping](#_Array_Searching_and) |
| [Arrays in the DOM](#_Arrays_in_the) | [Global Scope](#_Global_Scope) |
| [Function Scope](#_Function_Scope_1) | [var and Hoisting](#_var_and_Hoisting) |
| [Undeclared Variables and Strict Mode](#_Undeclared_Variables_and) |  |

## New Tools

The course begins with tools to be downloaded.

* Node.js – refer to **My Documents\Current Pixel\Node\_js**. Node.js is a JavaScript runtime built on Chrome’s V8 JavaScript engine.
* Git – refer to **My Documents\Current Pixel\SourceControl**.Git is a [free and open source](https://git-scm.com/about/free-and-open-source) distributed version control system
* Editor – refer to **My Documents\Current Pixel\VisualStudioEditor**. To download this type “code.VisualStudio.com” as a browser URL

## Introductory Project

The introductory project displays “Hello, World”. The Java-Script code is retrieved from Git Hub using the command prompt. From the command prompt the command to retrieve the code is

**git clone https://github.com/pluralsight/web-dev-starter.git**

This command copies the code to a folder -- **web-dev-starter--** under the starting folder of the command prompt – in my case this is

**C:\users\sncole\source\GitRepositories\VSAndOtherPlatformProjects\JavaScript\_GettingStarted**

The next step was to start the Visual Studio Editor, which I did -- not via the command prompt -- but by clicking its button pinned to the task bar. First I navigated to **c:\users\sncole\** **Pluralsight\_js\web-dev-starter** by invoking Visual Studio Editor’s pull-down menu **File > Open Folder… > *file\_browser***. Next I shut down the welcome page by clicking the ‘x’ next to ***Getting Started*** (in the upper left corner of the large pane). The JavaScript is in **index.html**, which is accessible from Visual Studio Editor’s explorer panel on the left edge.

Visual Studio Editor has a built-in command prompt, and you can invoke it by typing <ctrl>` (the character after <ctrl> is sometimes called an “accent” or a “backward apostrophe”).

Before invoking this – and any node.js project - for the 1st time, it is necessary to install the npm packages that belong to the project. From the command prompt which has navigated to the same folder

**c:\users\sncole\Pluralsight\_js\web-dev-starter**

I used the command

**npm install**

This did not work at the outset. I noticed that the folder **node\_modules** had not been loaded. To correct the error, I started the Git Bash command-line interpreter, and I navigated to

**c:\users\sncole\Pluralsight\_js\web-dev-starter**

Then I typed

**npm install**

again, and it worked by loading the folder **node\_modules**.

The Pluralsight instructor advised that it is safer to type **npm install** from an external command prompt – not from the command prompt in the Visual Studio Editor. Another action triggered by **npm install** was to look at package.json, which is listed in **EXPLORER** - in Visual Studio Editor’s left pane.

* Open package.json by clicking the name in the left pane.
* **npm install** noted the dependency item **lite-server** (version 2.5.4); this is indicated by the text in package.json immediate after “devDependencies”:, and it did – indeed – install **lite-server**. (Open node\_modules by clicking the name in the left pane, and scroll down the list, where you will find **lite-server**.)
* You can invoke **lite-server** from **Windows PowerShell**, and you can invoke **Windows PowerShell** by typing <ctrl>`. The command prompt is “PS”, and by default the current folder is the folder, web-dev-starter, the folder that contains this JavaScript code.
* To start **lite-server** type the command at the **PS** prompt

**npm run start**

and the browser shows “Hello World”.

To make development easy to use, it is useful to have the Visual Studio Editor displayed side-by-side with the Browser (which is now displaying “Hello World!”. Resize both windows – Visual Studio Code and the Browser, and move them so that they share the screen (e.g Visual Studio Code occupying 60% of the terminal, and the Browser to the right of Visual Studio Code occupying 40% of the terminal). You can gain more real-estate in Visual Studio Code by hiding the explorer in the left pane; the icon in the upper-left corner (with 2 cascaded sheets) is a toggle switch that alternately hides and displays the explorer.

Try making a small change to “Hello World!”. For example, delete “World”. Save the text change by typing <ctrl>s. After a couple of seconds you will see the change reflected in the browser.

## Sample Website

The downloaded Exercise Files is a zip file titled javascript-getting-started.zip. When you expand javascript-getting-started.zip, you will see a list of folders titled “02”, “03”, . . . , “10”. The folder “02” for section 2 of this Pluralsight course contains the files that we need for this clip. (Do not follow the instructor’s example of expanding “05”.) When you drill down into “02”, then “demos”, then “after”, then “Js-demo”, you will see a set of files that are similar to those in [Introductory Project](#_First_Project).

* I chose to copy the files from js-demo to **c:\users\sncole\source\GitRepositories\VSAndOtherPlatformProjects\JavaScript\_GettingStarted\SectionModule2**
* I started a Git Bash command prompt, and I navigated to the folder containing the copied files.
* Type the command **npm install**. You may have to wait a few seconds to get a response. Among other things **npm** has brought several files into folder node\_modules.
* Type the command **code .** to invoke Visual Studio Code.
* Expand index.html in the **Explorer** pane, and note that this example contains considerably more code than our Hello, World example. In fact, the only two differences are this index.html file and the inclusion of main.css in the css folder.
* Start the internal command prompt by typing <ctrl>`.
* Type **npm run start** to invoke lite-server. lite-server, in turn, starts the browser, where one can view the HTML and Java Script in action.
* Resize both windows – Visual Studio Code and the Browser, and move them so that they share the screen (e.g Visual Studio Code occupying 60% of the terminal, and the Browser - to the right of Visual Studio Code - occupying 40% of the terminal). You can gain more real-estate in Visual Studio Code by hiding the explorer in the left pane; the icon in the upper-left corner (with 2 cascaded sheets) is a toggle switch that alternately hides and displays the explorer.

In subsequent sections we will be adding JavaScript to the code in index.html, and we’ll observe the effect by looking at particular portions of what is displayed by the Browser. Also, we may want to look at console output.

* Right click in the browser, and click **Inspect** in the pop-up menu. The result will be a smaller window that occupies part of the browser window. At the top of this smaller window is a horizontal list of tabs.
* Click the tab titled **Console**. The console window will (evidently) accept commands; additionally it can be used to display output.
* You can dock this smaller window in different places. The Pluralsight instructor recommends docking it at the bottom of the browser window. To do this click the icon to the right of the horizontal tabs that shows 3 vertical bullets. Then click the dock-side icon that indicates “bottom”.

## Adding JavaScript Code to a Web Page

A very common use of JavaScript is to enhance the markup in an HTML page.

The **<script>** tag separates HTML code from JavaScript code.

. . .

<script>

// JavaScript goes here

</script>

. . .

The closing tag </script> is important. (The syntax <script “src=./*filename*.js” /> may work with some browsers, but not necessarily all browsers.)

A convenient place to put JavaScript is immediately before the closing **</body>** in index.html, and the JavaScript example in this case is

alert(‘Hello World!’);

In summary the end of index.html - in our sample web site – should look like

<script>

alert(‘Hello World!’);

</script>

</body>

</html>

Before typing <ctrl>s (to save the code changes) make certain that the Browser is running. If Visual Studio Code is not showing a command prompt, type <ctrl>`. Then to start the browser type

**npm start**

(Evidently “run” is optional in **npm run start.**) Finally type <ctrl>s to start (or restart) executing the code in index.html. You will see “Hello, World!” displayed on the browser; click the **OK** command button to dismiss the alert message. You can change the JavaScript, and see the results immediately. For example, add a 2nd **alert()** function call immediately after alert(‘Hello World!’);. In response to <ctrl>s, we get 2 messages in succession (as expected).

## Working with JavaScript Files

Even though <script> can be used to set off JavaScript in an HTML file, it’s better practice to put this code into a file (suffix .js).

<script “src=./*filename*.js”></script> // ./ means that the file is in the current folder

As mentioned earlier, it is important to end these statements with “> </script>” (instead of “/>”).

Make the changes – recommended above – to folder SectionModule2.

* Remove the **alert** statements.
* Insert

src=”./home.js”

after “<script”.

* Create a new file in SectionModule2, home.js, whose content is

alert(‘Hello World!’);

alert(‘Second note’);

* Save the changes to index.html and to home.js. As expected, the execution of this code results in two alerts.

Refer to the 1st few statements in index.html - in the <head> section. The <head> section contains a few <script> . . . </script> statements. They refer to types of files (such as “jquery” and “bootstrap”), which do not rely on the web page; therefore, they can be placed in the <head> section. But the JavaScript that we write in this course will make modifications to the web page. Therefore, the <script> tag statements that we introduce need to be placed at the very end – after all of the markup is loaded.

Our next step is to add a second <script> tag at the end, a utility JavaScript file, named “util.js”. Insert

<script src=”.\util.js”></script>

immediately before the reference to home.js. Create a new file in SectionModule2, util.js, whose content is

function showMessage(message) {

document.getElementById('message').textContent = message;

}

To test showMessage() let us apply it to the markup element that displays GET A GRIP. Search index.html for “GET A GRIP”. You will find it in an <h1> tag. Add a qualifier

id=”message”

to this tag. Next modify home.js, and replace the 2 alert statements with

showMessage(“Changes...”);

Save all of the file changes ( File | SaveAll). Use the “refresh” icon (circular arrow) to induce the page to execute the changed JavaScript. The browser will show “Changes...” instead of “GET A GRIP”. Change the argument of showMessage() to “title”, and refresh again. The browser will show “title” instead of “Changes...”.

## Formatting Code

White space is permitted – similar to c, c++, and c#. But exploit this only when it improves readability.

## Detecting and Fixing Errors

Let us introduce an error (deliberately).

* Open home.js.
* Add a 2nd line of code after showMessage('Title');

showMessage('Title2');

* Introduce a new-line character between the two s’s in the 2nd line. Lines 2 and 3 are now

ShowMes

sage('Title2');

* Save the changes to home.js.
* Try to display the page. “Title” is displayed, but “Title2” is missing. This tells us that the error occurred somewhere after the 1st line of home.js executed.
* Right-click the displayed output, and click **Inspect** in the pop-up menu. A smaller window appears in the displayed-output window.
* Click the **Console** tab near the top of the smaller window. A diagnostic is displayed with red text: “Uncaught Reference Error: showMes is not defined at home.js:2”. The JavaScript interpreter does not recognize “showMes” in the context of home.js on line 2.
* Go back to home.js, and remove the new-line character that separates “showMes” from “sage”.
* Save home.js. In response, “Title2” is displayed, and the red diagnostic disappears from the console.

It can be useful to use the console to track the progress of execution.

* Replace the 2nd instance of showMessage() in home.js with

Console.log(“any message...”);

* Save home.js. In response, “Title” is displayed in the output, and “any message” is displayed in the console.

## Case Sensitivity

JavaScript is case sensitive.

Camel notation – as in **accountName** is common in JavaScript.

## Commenting Code

// This is a single-line comment.

/\* This is a multiline

comment \*/

/\* It could also create a single-line comment – maybe followed by JavaScript \*/ <script src= . /util.js>

</script>

## What Is a Variable?

A variable is a repository for data – a small amount of data that is stored in the computer’s memory. In JavaScript (like other computer languages) variables have names. For example “total” could be the name of word in memory containing the bottom line of an invoice; the value stored in that memory could be 149.99, which is understood to be a dollar amount. A description of a product could be stored in a string variable, whose value is “Hiking Boots”, and this variable’s name would be “product”. A 3rd type of JavaScript variable is a Boolean – containing either true or false.

## Declaring Variables

* The keyword **var** – to introduce a variable is deprecated. It may still be present in some old JavaScript, but it is inadvisable to use it.
* The keyword **let** is used to declare a variable and to set its initial value at the same time. For example . . .

let price = 23.45;

let accountName = ‘Sales’; // instead of single quote (‘), we could use double-quote (“)

let discounted = true;

* To change the value of a variable, the syntax is the same as the declaration but with **let** omitted.
* A constant declaration uses the keyword **const** instead of **let**.

## Using let to Declare Variables

Returning our attention to the sample code (SectionModule2), edit home.js. Insert - at the beginning of home.js

let welcome = ‘Welcome’;

In the call that invokes showMessage(), change its argument to “welcome”, I.e.

showMessage(welcome);

Save the changes, and – as a consequence – we see “Welcome” in the displayed output.

Change the statements in home.js to

let price = 49.99;

showMessage(price);

Save the changes, and – as a consequence – we see “49.99” in the displayed output.

We could add 2 additional let statements – e.g. for “name” and “discounted”, but it is good practice to combine these into a single statement, separated by commas.

let price = 49.99,

name = ‘Hiking Boots’,

discounted = false;

showMessage(name);

## Naming Variables

A variable name consists of letters, decimal digits, ‘\_’, and ‘$’. But the 1st letter of a variable must not be a decimal digit.

Do not name variables to be the same as keywords such as **let** or **const**.

## Common Errors Using Variables

The common errors are starting the name with a decimal digit, having whitespace in the variable name, and using a keyword for a variable name. A variable does not have to be initialized, but before referring to the variable, it is a good idea to supply a value. The following statements are valid.

let price;

showMessage(price);

console.log(price);

But it probably does not produce the intended results. The text from showMessage() is not shown. And the result from console.log() is “undefined”. It is good practice to supply an initial value when declaring a variable.

## Changing Variable Values

To change the value of a variable, the syntax is the same as the declaration but with **let** omitted.

let price = 49.99; // initialized value

. . . .

price = 99.99; // modified value

## Constants

If a stored value is not supposed to change during the life of a VisualStudio program, it is good practice to declare it as a constant. Instead of

let price = 49.99;

use

const price = 49.99;

Moreover, assign a value to the constant when declaring it.

## The var Keyword

**var** is similar to **let** in JavaScript. Both of these declare a variable with an optional initial value. But using **var** is no longer good practice. The Pluralsight instructor showed one example, where using **let** made it easier to find an error.

## Numbers

Navigate to home.js in SectionModule2. Get rid of old code in home.js. Add the statements

let price = 20.99;

showMessage(price);

The 1st statement specifies not only the value but the type of “price”, a number. Save the changes, and “20.99” should be displayed.

**typeof** is a unary JavaScript operator that – when applied to a variable or constant – yields the type (number, string, etc.) of that variable or constant. Change the 2nd statement to

showMessage(typeof price);

Save the change, and “number” should be displayed. Change the 1st statement to

let price = ‘20.99’; // quotation marks around the number

Save the change, and “string” should be displayed. Remove the quotation marks.

In addition to the standard arithmetic operators [+, -, \*, /], JavaScript also supports the **modulo** operator ‘%’; 4 % 3 equals 1; 15 % 5 equals 0; etc.

Operators can be combined with assignment (=) for abbreviation. After

prince += 2;

price contains 22.99. ++ and -- can be used as a prefix or as a suffix to increment or decrement by 1.

## Operator Precedence

Use the keywords **“mdn operator precedence”** with Google to find an article that describes JavaScript operator precedence rules. **mdn** is the Mozilla Developer Network, which is a great source for information related to JavaScript.

## Number Precision

Numbers in JavaScript contain floating-point values. Therefore, there is always the possibility that a number – obtained as a result of some arithmetic – could be very slightly inexact as a consequence of floating-point inaccuracies.

## Negative Numbers

Negative numbers do not introduce any unusual problems. But to improve readability when working with a negative-number literal, it is good practice to use parentheses. For example

price \* (-2)

## Strings

A string literal is enclosed in quotation marks (“) , (‘), or (`). To embed a quotation mark in the string, one could use a different quotation mark for delimiters, which would prevent the embedded quotation mark from appearing to be a delimiter. Another possibility is to use the “escape” character {\) as a prefix of the embedded quotation mark. Thus “Hello \”World\”” would produce

Hello “World”

To get more information about the escape character, keywords use the keywords **“mdn string”** with Google. Search for “escape”; you will find other characters that are available using “escape” notation.

The back-tick delimiter (`) can be used to introduces a useful feature; it allows an imbedded variable – called “interpolation”. Change home.js in SectionModule2 to contain

let name = "Andrea";

let message = "Hello ${name}";

showMessage(message);

Save the change, and “Hello Andrea” should be displayed.

## Manipulating Strings

String concatenation is expressed by the (+) binary operator.

JavaScript provides several methods that work with strings. For example change home.js in SectionModule2 to contain

let message = "Hello ";

message = message.toLowerCase(); // use intellesense by typing .to; don’t forget “()”

showMessage(message);

Save the change, and “hello” should be displayed. Another commonly used method is .substring().

In addition to string methods there are also string properties. .length is a property that is used frequently. Change home.js to

let message = "Hello ";

let leng = message.length;

showMessage(leng);

Save the change, and “5” should be displayed.

## Converting Strings and Numbers

To convert a number to a string, use the ToString() method. For example (home.js in SectionModule2)

let price = 49.99; // numeric form of “price” starts with a lower-case letter

let Price = price.toString(); // string form starts with an upper-case letter

alert('$' + Price); // $49.99 is displayed

To convert a string to a number, use the parseFloat() function. For example

let Price = “49.99”; // string form starts with an upper-case letter

let price = Number.parseFloat(Price); // numeric form starts with a lower-case letter

showMessage (price - 3); // 46.99 is displayed

## Boolean Variables

The Boolean literals are true and false. A commonly used Boolean operator is “!”. “!” is a prefix operator.

let saved = true; // value is true

saved = !saved; // value changes to false

showMessage (saved); // false is displayed

## null and undefined

null and undefined are two additional types. There is one value for the null type – null. There is one value for the undefined type – undefined.

Earlier in this course (see [Common Errors Using Variables](#_Common_Errors_Using)) we saw the consequence of declaring a variable without supplying an initial value.

Another possibility is that variable was defined, but as a consequence of processing, we decide to remove its initial value. Conventionally the programmer does this by setting the value to **null**, not to **undefined**.

let price = 12.34;

// . . . unspecified processing, which leads us to prefer to remove the value from price

price = undefined; // not good practice – conventionally JavaScript sets value to undefined

//price = null; // better practice – this indicates program’s decision to remove value

showMessage(price);

console.log(price);

Try the above – alternately commenting out the 3rd and 4th lines; check the console output each time.

## Objects and Symbols

A person usually has a first name and a last name. In JavaScript we express this by

let person = {

firstName: ‘John’,

lastName: ‘Adams’

};

Copy this declaration to home.js, and follow it with

console.log (typeof person);

After you save these changes, the display shows that the type of “person” is **object**.

**firstName** and **lastName** are properties of the object **person**. Properties are accessible via the (.) operator.

console.log (person.lastName) // displays Adams

Instead of **person.lastName** to retrieve the property, there is also the notation **person[‘lastName’]**.

The other data type is **symbol**. A **symbol** can be used as a property in an object, but it is hidden. We will visit the **symbol** later in this course when we cover objects in more detail.

## Conditionals Using if()

The **if** keyword introduces a condition. The parentheses immediately following **if** are required, and these parentheses surround the condition.

Comparison operators are similar to c, c++, and c#, except in the following cases.

* The comparison for equality is expressed by ===.
* The comparison for inequality is expressed by !==.

In other programming languages the equality/inequality comparison operators are == and !=; these are also available in JavaScript, and they permit comparisons between items (variables, constants, or literals) that are of DIFFERENT types. This can lead to confusion, and it is recommended that these operators be avoided

## Truthy and Falsy

The only number that – by itself – has a **false** value is 0. Among strings an empty string (“” or ‘’) is **false**. The values **null** and **undefined** evaluate to **false**. **NaN** (not a number) evaluates to false. All other entities evaluate to **true**.

When you are working with non integers, floating-point inaccuracies can make comparisons tricky. One way to resolve this problem is to round floating point values to a fixed number of digits. The syntax for rounding to 2 digits is

+*floating-point-value*.toFixed(2)

## if … else

The syntax for a list of several consecutive **if** statements – an alternative to **case** statements - uses **else if**.

If (*condition1*) {

*Code\_block\_1*

}

else if (*condition2*) {

*Code\_block\_2*

}

else if (*condition3*) {

*Code\_block\_3*

}

etc.

## Comparing === and ==

Repeating what was said earlier . . .

In other programming languages the equality/inequality comparison operators are == and !=; these are also available in JavaScript, and they permit comparisons between items (variables, constants, or literals) that are of DIFFERENT types. This can lead to confusion, and it is recommended that these operators be avoided.

## The Ternary Operator

This is an abbreviation for if/else. In c, c++, and c# , the 3 parts of the ternary are delimited by “?” and “:”. It is similarly available in JavaScript. For example

discount = (price > 100) ? 0.15 : 0.09;

In plain language when the price is greater than 100, the discount is 15%; otherwise, the discount is 9%.

## Block Scope Using let

Recall that variables and constants are declared via **let** and **const**. When declarations are placed within a pair of braces - “{“ and “}”, the scope of these declarations is limited to code within the braces. For example (home.js in SectionModule2)

if (true) {

let value = ‘yes’;

showMessage (value);

}

console.log (value);

When we save the changes, the word ‘yes’ appears in the display, but the console subsequently registers an error

Uncaught ReferenceError: value is not defined

and the console indicates that the error occurs in the 5th line (OUTSIDE OF THE SCOPE).

## Looping with for()

The for() statement is one way to express a loop.

for ( let i=*initial\_value*; *test\_whether\_to\_continue*; *increment\_statement*) {

*code\_block*

}

As an example (try this in home.js) . . .

for ( let j = 10; j < 15; ++j ) {

console.log ( j );

}

The console displays 10, 11, 12, 13, 14.

It’s not at all clear whether the scope of variable j persists beyond the **for** loop. To test this let’s change the requirement of the **for** loop. Specifically; *display indexes until we encounter an index that is divisible by 3. Then – after the* ***for*** *loop - display the index that is divisible by 3.*

for ( let j = 10; ( j % 3 ) !== 0; ++j ) {

console.log ( “not “ + j );

}

showMessage(j);

The console displayed “not 10” and “not 11”. But after executing the loop it displayed

Uncaught ReferenceError: j is not defined

This tells us that the scope of the for-loop index does not extend beyond the closing “}”. An easy way to get around this limitation is to use a different flavor of loop – coming up next.

## Looping with while()

Another way to express a loop uses **while**.

while ( *condition* ) {

*code\_block*

}

The loop continues as long as *condition* evaluates to **true**. Therefore, *code\_block* must eventually change some value that causes the *condition* to evaluates to **false**; otherwise, you get an infinite loop.

## Looping with do … while()

A third way to express a loop uses **do … while**.

do {

*code\_block*

}

while ( *condition* )

As in c++ and c#, (for all 3 types of loop) the **break** statement can be used to exit the loop unconditionally, and the **continue** statement can be used to jump to the end of the of the *code\_block*.

## Function Basics

The syntax of a function declaration is

function *name\_of\_function* ( *optional\_arguments* ) {

*body*

}

## Function Expressions

An alternative syntax is

let *name\_of\_function* = function *optional\_debugging\_function\_name*  ( *optional\_arguments* ) {

*body*

}

You invoke the function using *name\_of\_function*, not *optional\_debugging\_function\_name*. It is good practice to include *optional\_debugging\_name\_of\_function*, because the browser’s console can use that name to refer to the function in the event of an error.

## Function Within Object

A function can be declared within an object. Expanding the *person* object (introduced above in [Objects and Symbols](#_Objects_and_Symbols)), we could have

let person = {

firstName: ‘John’,

lastName: ‘Adams’,

showInfo: function() {

showMessage(“name is “ + this. firstName + “ “ + this.lastName);

}

};

## Passing Information to Functions

In [Function Basics](#_Function_Basics) and [Function Expressions](#_Function_Expressions) we indicated that one could include *optional\_arguments* within the parentheses - immediately after the function’s name. *optional\_arguments* consists of a series of names, separated by commas. Each of these names is a place-holder for the information that the function uses. Sometimes these names are called parameters. In JavaScript the types of parameters (number, string, Boolean, …) are not specified in function declarations.

## Passing Information to Functions by Reference or by Value

When you pass a scalar (e.g. an ordinary variable) as an argument to a function, and when the function changes the value of that variable, this has no effect on that scalar (i.e. passing by value). But when you pass an object (including an array) as an argument to a function, and when the function changes the value of one of the object’s properties, this is – indeed – a change that affects the object (i.e. passing by reference).

## Function Scope

The scope of variables (and constants) declared within the body of the function is limited to that body.

## Using Functions to Modify Web Pages

Objects on a web page have a protocol called the “Document Object Model” – or more briefly DOM. **document** is an object that refers to the entire web page. The word **element** in DOM terminology is a string of HTML code, e.g.

<h1 class=”col-sm-12”>GET A GRIP</h1>

A way that JavaScript is frequently used is to change the text of some **element**. You can refer to the element’s **identifier** or to its **class**. Recall that we changed the above **element** (in [Working with JavaScript Files](#_Working_with_JavaScript)) by inserting

id = “message”

Immediately after “<h1”. This made “message” the **identifier** of this <h1> element. Therefore, the phrase

document.getElementById('message')

(in showMessage() in utils.js) gives us a reference to this <h1> element, and we used that reference to change the <h1> text.

As a 2nd example, the element immediately following the <h1> element is

<h2>20% OFF</h2>

We can similarly insert an **identifier** for this **element**.

<h2 id="percent\_off">20% OFF</h2>

Now to change the text of this **element** we supply a 2nd function in utils.js.

function changePercentOff ( percentage ) {

Document.getElementById ( 'percent-off' ).textContent = percentage + "% OFF";

}

Test the above code by putting

changePercentOff(30);

Into home.js.

## Object Properties

This clip contains a review of the instruction in Objects and Symbols. The only information added was a brief explanation of symbols; one declares a symbol outside the object declaration, and then refers to that symbol in the object declaration. Elaborating the object example in [Objects and Symbols](#_Objects_and_Symbols)

let mySymbol = Symbol(); // declare a symbol named “mySymbol”

let person = {

firstName: ‘John’,

lastName: ‘Adams’,

[mySymbol]: ‘information that we are hiding’

};

JavaScript code that does not have access to mySymbol, can read or write only person.FirstName or person.LastName; but without access to mySymbol, the mySymbol component of person would be hidden.

## Object Methods

In addition to properties and symbols an object declaration can contain function declarations – called methods. For an example refer to [Function Within Object](#_Function_Within_Object). The function within the object can contain the **return** statement, and it can react to parameters.

## Passing Objects to Functions

Recall – in [Passing Information to Functions by Reference or by Value](#_Passing_Information_to_1) - that an object parameter to a function is passed by reference. If a property of that object changes, the change remains in effect after the function completes. Try the example`

let person = {

name: ‘John’,

age: 32,

partTime: false

};

function incrementAge ( persn ) {

persn.age++

}

incrementAge ( person );

showMessage ( person.age );

When you test this code, you can observe that the property “age” changes according to incrementAge().

## Standard Built-in Objects

Use the keywords **“mdn built-in objects”** with Google to find an article that discusses objects that are automatically available. For example **Array**, **Date**, **Math**, **String**, . . .

Click on **Date** in the left –hand column for suggested methods that one could invoke to get information about the date. These methods apply not only to the date object (which contains current date/time), but also to date objects that one might obtain from a database.

let myDate = new Date(); // gets the current date

let myDate2 = new Date('April 8, 2021'); // or you could define it with year, month, day, etc.

showMessage ( myDate.toDateString() );

console.log ( myDate2.toDateString() );

Click on **String** in the left-hand column for suggested methods pertaining to strings. If you encounter a method in the mdn documentation that is preceded by a thumb-down indication, it is good idea not to use that method, because it likely that this method will be deprecated in the near future.

myString.charAt(*position*) returns a 1-character string in the indicated position (count starting from 0).

myString.indexOf(*substring*) finds a particular substring.

myString.substring() returns the substring specified by its starting and ending positions.

Notice, also, the methods that begin with “.trim”.

The following example illustrates the use of .length, .indexOf(), and .substring() to find two successive instances of a particular substring (‘help’) that is sought.

const myString = 'Jane Phelps needs help.';

const strSought = 'help';

const firstIx = myString.indexOf ( strSought ); // 1st instance of 'help' as a substring

console.log ( '1st instance is at ' + firstIx );

// # characters through end of 1st instance

const numThroughHelp1st = firstIx + strSought.length;

// substring after 1st instance

const restOfMyString = myString.substring ( numThroughHelp1st );

showMessage ( restOfMyString );

// 2nd instance of 'help' as a substring

const secondIx = restOfMyString.indexOf ( 'help' );

console.log ( '2nd instance is at ' + (numThroughHelp1st + secondIx) );

## The Document Object Model (DOM)

The Document Object Model (DOM) pertains to browser programming, using JavaScript for web pages. The web page contains several objects, and the interaction between JavaScript and these web-page objects is called the Document Object Model (DOM). More specifically the DOM deals with modifying the web page, and responding to the events triggered by a user’s typing or mouse-clicking while the focus is on one of the web-page objects.

Refer to showMessage() and changePercentOff() in utils.js. “document” in these functions represents the web page, which is a JavaScfript object. “document” has a method, getElementById(). The return value of getElementById() is another object, which has a textContent property – the text that one can see on the web page.

Refer to the <h1> tag in index.html, whose text is “GET A GRIP”. This is called an “element”. Earlier in this course we inserted

id=”message”

which allows JavaScript to obtain access to this element.

## Styling DOM Elements

One of the properties of DOM elements is “style”. Although styles (such as color, font, . . .) are often specified via CSS, the “style” property can be used to specify styles in JavaScript. Type the following into home.js.

const header = document.getElementById (‘message’);

header.style.color = ‘red’;

When we save this. the pertinent text changes to a red color. “color” is only one example of a “style” property. The names of these properties are the names specified via CSS. Some of these names have an embedded dash, e.g. “font-weight”. But “color. font-weight” is invalid JavaScript code. The work-around is to use camel case. Try

const header = document.getElementById (‘message’);

header.style.fontWeight = 100;

When we save this, the font style changes appropriately.

## Detecting Button Clicks

There is a button in our sample web page (immediately below “Asphalt National Park”), whose text is “SEE REVIEW”. Search for “see review” in index.html, and we find in an anchor <a> element. Delete “href=#" (to prevent the attempted mouse-click response of migrating to a different web location). Now we will supply code to handle the mouse-click in JavaScript. Insert ‘id=”see-review” ’ immediately before ‘class=’. Save the change, and navigate to home.js. Delete old code in home.js, and add

const button = document.getElementById ( ‘see-review’ );

button.addEventListener ( ‘click’, function() {

console.log ( ‘click’ );

} )

Save the code, open the console, and click the button. Observe that ‘click’ is displayed in the console log; observe, also, that when you repeatedly click the button, a number is displayed before ‘click’ expressing the number of times that ‘click’ was logged.

## Showing and Hiding DOM Elements

In this section we are going to illustrate how to respond to the button clicks by alternately showing and hiding a review expressed in HTML.

Open index.html. Add the following mark-up after the 3rd </div> that follows the button’s html.

<div id=”review” class = “container d-none">

<h4>Review Title...</h4>

<p>Review text...</p>

</div>

Note that this block of mark-up has a class name “container” and a class name “d-none”. “d-none” denotes “display none”, and this is why the mark-up is not visible at the outset.

Save the changes to index.html, and navigate to home.js. Replace “console.log ( ‘click’ );” with

const review = document.getElementById ( ‘review’ );

review.classList.remove ( ‘d-none’ );

The classList property is evidently the array of class names in the element, and remove() evidently deletes the specified class name from the array. Save the changes to home.js. Click the button, and observe that the “review” is now visible.

Now, all that remains is to change the button to an alternate action switch. Instead of the single statement

review.classList.remove ( ‘d-none’ );

use some if/else code to add ‘d-none’ to the classList if it not in the list, or remove it if is in the list. At the same time change the text on the button’s face. Save and test the following code.

if ( review.classList.contains ( ‘d-none’ ) ) {

review.classList.remove ( ‘d-none’ );

button.textContent = ‘CLOSE REVIEW’;

}

else {

review.classList.add ( ‘d-none’ );

button.textContent = ‘SEE REVIEW’;

}

## Creating and Initializing Arrays

An array can be declared without initializing it.

let *array\_name* = [];

An initialized 3-element array is declared . . .

let *array\_name* = [ *value\_of\_1st\_element*, *value\_of\_2nd\_element*, *value\_of\_3rd\_element*  ];

An alternative way to do this is

let *array\_name* = Array.of ( *value\_of\_1st\_element*, *value\_of\_2nd\_element*,

*value\_of\_3rd\_element*  );

A constant array is similar, with “const” in place of “let”.

It is permitted to have elements with different data types in the array, but avoid doing this unless there is a compelling reason.

Try the following example in home.js.

const values = [ ‘a’, ‘b’, ‘c’ ];

console.log ( values );

The console displays the length of the array and the 3 elements. Try clicking the triangle on the console to the left of “(3)”; the response is the same information with more detail. Notice that the indexes start at 0; thus the 3 indexes of the array above are 0, 1, and 2.

(typeof array0)

Is always “object”. To test whether **array0** is an array use the fact that

Array.isArray(array0)

has the value **true** if and only if **array0** is an array.

## Accessing Array Items

Returning to our example in home.js, use

values[0]

to obtain access to the 1st element of this array.

Try

console.log ( values [ 3 ] );

and you will be informed that this value is **undefined**.

This same [] notation can be used to change the value of an element, e.g.

values [ 2 ] = ‘abc’;

## Manipulating Arrays

Use the keywords **“mdn array”** with Google to find an article that describes JavaScript arrays – properties, methods, etc. Some frequently used methods are push(), pop(), shift(), unshift(), slice(), and splice().

**array0.push(x, y, z, . . .)** appends the elements x, y, z, . . . to the end of **array0.**

**array0.pop()** removes the largest-indexed item from **array0** and returns that element to the calling function.

**unshift()** and **shift()** are analogous to **push()** and **pop()**, except that they work with the 0th element of the array.

The following is an example.

const values = [ 'a', 'b', 'c' ];

values.push ( 'd', 'e', 'f' );

console.log ( 'A: ' + values ); // a,b,c,d,e,f

const last = values.pop();

console.log ( 'B: ' + values, last ); // a,b,c,d,e f

const first = values.shift();

console.log ( 'C: ' + values, first ); // b,c,d,e a

values.unshift ( 'hello' );

console.log ( 'D: ' + values ); // hello,b,c,d,e

## slice() and splice()

**array0.slice(*begin, end*)** returns the sub-array starting from index *begin* and going up to but not including index *end*. a**rray0** is not affected by **slice()**.

**array0.splice(*begin, count, new\_element*)** removes ***count*** elements from **array0**, starting from index *begin*, and inserts the element ***new\_element*** in place of the deleted elements. **c*ount*** may be 0 (if you only wanted to insert, not delete). ***new\_element*** is optional and may be omitted (if you only wanted to delete, not insert).

## Array Searching and Looping

**IndexOf,** **filter**, and **find** can be used to search for specific values in an array.

**Array0.indexOf(x, fromIndex)** returns the index of the earliest occurrence of **x** as an element of **Array0**, with the search starting at index **fromIndex**. **fromIndex** is optional; if omitted, the search starts at index 0. If **x** is not present in **Array0**, **indexOf** returns –1.

**Array0.filter(*callbackFunction*)** returns an array composed of elements chosen from **Array0**. The criterion for being chosen is expressed by ***callbackFunction***. In its simplest form ***callbackFunction*** takes a single argument, whose type is the same as that of the elements of **Array0**, and it returns a Boolean value. The elements of **Array0** are passed – one at a time - to ***callbackFunction***, and if ***callbackFunction(Array0Element)*** equals **true**, the element satisfies the criterion. The order of the elements in the returned array is the same as the order in which these elements appear in **Array0**. ***callbackFunction*** can either be defined as an anonymous function in the **Array0.filter(*callbackFunction*)** statement, or it can be specified elsewhere, and its name supplied as **filter**’s argument.

The following is an example of **filter()**.

const words = ['spray', 'exuberant', 'limit', 'elite', 'destruction', 'present'];

function longWord ( word ) {

return word.length > 6;

}

const result = words.filter( longWord );

console.log ( result ); // 'exuberant', 'destruction', 'present’

You will notice that **find** and **filter** are very similar. The difference is that **find** returns the earliest element that satisfies the criterion, whereas **filter** returns an array of all elements that satisfy the criterion.

**Array0.find(*callbackFunction*)** returns the earliest element in **Array0** that satisfies a criterion. The criterion is expressed by ***callbackFunction***. In its simplest form ***callbackFunction*** takes a single argument, whose type is the same as that of the elements of **Array0**, and it returns a Boolean value. The elements of **Array0** are passed – one at a time - to ***callbackFunction***, and if ***callbackFunction(Array0Element)*** equals **true**, the element satisfies the criterion. If none of the elements in **Array0**satisfies the criterion, **find** returns **undefined**. ***callbackFunction*** can either be defined as an anonymous function in the **Array0.find(*callbackFunction*)** statement, or it can be specified elsewhere, and its name supplied as **find**’s argument.

**findIndex()** is similar to **find()**, except it returns the index of the “found” element, or it returns –1 if no element in **Array0** satisfies the criterion.

**Array0.forEach**(***callbackFunction***) performs ***callbackFunction*** once for each element in **Array0**. ***callbackFunction*** takes one argument; when it is called, the value of an element from **Array0**is assigned to that argument. ***callbackFunction*** can either be defined as an anonymous function in the **Array0.forEach(*callbackFunction*)** statement, or it can be specified elsewhere, and its name supplied as **forEach**’s argument.

## Arrays in the DOM

Navigate to index.html in our sample web site. Notice that “container” is a class name in several of the mark-up elements. We can gather all of these into an array of objects using

const containers = document.getElementsByClassName ( ‘container’ );

console.log ( containers );

Strictly speaking containers is not a JavaScript **array**. The response from console.log shows that it is, instead, an HTMLCollection; but it behaves very much like an array. Click the triangle in the console output to see a vertical display of the list of “container” elements. In particular focus on the 2nd element of the “list” - div.container.crf-cigar-banner. Scroll the display so that the boots are visible, and hover the mouse over this 2nd element. Notice that the boots become shaded, which indicates that this <div> contains the boots. Suppose we wanted to hide the boots. Insert a statement before the console.log statement yielding

const containers = document.getElementsByClassName ( ‘container’ );

containers[ 2 ].classList.add ( ‘d-none’ );

console.log ( containers );

As a consequence of the added statement, the boots become hidden.

## Global Scope

The concept of **scope** was introduced earlier – see [Block Scope Using let](#_Block_Scope_Using) and [Function Scope](#_Function_Scope). Declarations that are not within braces have global **scope**, and JavaScript statements are free to refer to such declared items from any location in JavaScript code. Recall, also, that JavaScript code is typically introduced by <script> tags; in our sample program, <script> introduces home.js and and util.js. A global-scope declaration in one of these .js files is accessible in the other .js file.

A careless programmer can get into trouble when too much use is made of global-scope declarations of data – constants and variables. A good practice is to collect all-such declarations into a single global object called “app” which serves as a repository for constants and variables that need to be accessible globally. (One would not use “app” to collect function declarations.)

const app = {

*Item1* : *initialValue1*,

*Item2* : *initialValue2*,

*. . .*

*Item\_n* : *initialValue\_n*

}

Then references to these MANAGED (almost global) items would use the prefix “app.”.

## Function Scope

The scope of variables (and constants) declared within the body of the function is limited to that body. This limitation also applies to the parameters of the function.

## var and Hoisting

With regard to constants and variables, if they are declared with **let** and **const**, these declarations must be encountered by the JavaScript run-time interpreter before any reference is made to them. Otherwise, the JavaScript interpreter tells us that they are undefined. One could say “they are not hoisted to the top of the source code”. On the other hand, functions can be invoked before they are declared; “functions are hoisted”. Another exception would be a data declaration using **var**. But it is bad practice to use **var**.

## Undeclared Variables and Strict Mode

JavaScript tends to be backward compatible with older standards. Undeclared variables or variables declared with “var” were previously permitted. If you want to detect and correct undisciplined JavaScript (possibly inherited), add the string literal

‘use strict’;

Place the literal at the top of a script module (*name.*js) to catch undisciplined code for the entire module. Place the literal within a function to catch undisciplined code for the entire function.