|  |  |
| --- | --- |
| [New Tools](#_New_Tools) | [Introductory Project](#_Introductory_Project) |
| [Sample Website](#_Sample_Website) | [Add JavaScript to HTML](#_Add_JavaScript_to_1) |
| [JavaScript Language](#_JavaScript_Language) | [Apply JavaScript to a Web Page](#_Apply_JavaScript_to_1) |
| [Standard Built-in Objects](#_Standard_Built-in_Objects) | [Less Disciplined JavaScript](#_Less_Disciplined_JavaScript) |

# 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.

Subsequent commands can safely be run from the built-in command prompt - <ctrl>`. To start the script from the command prompt type the command

**npm run start**

and the browser shows “Hello World”.

# Sample Website

The code for the sample website is found in the Pluralsight **Exercise Material**, which can be downloaded. The folder for section 2 of this Pluralsight course contains a set of files that are similar to those in [Introductory Project](#_First_Project).

* I chose to copy the files to a local folder **c:\users\sncole\Pluralsight\_js\demo**.
* I started a Powershell command prompt: (1) click the Windows **start** button (in lower-left corner); (2) type “powershell” into the *Search programs and files* text box followed by the **Enter** key; (3) click the program titled “Windows PowerShell”.
* I navigated to C:\users\Powershell\_js\_demo.
* I typed **npm install**.
* I started the Visual Studio Editor.
* I opened the folder C:\users\Powershell\_js\_demo.
* I started the internal command prompt via <ctrl>`.
* I typed **npm run start** to invoke the JavaScript web project.

**Use the steps above – except copying CSS files and index.html to initiate a new Website. You are welcome to create your own CSS files, and Index.html will be your creation. You will need the node\_module folder, because it seems to contain the code that drives the editor.**

# Add JavaScript to HTML

. . .

<script>

// JavaScript goes here

</script>

. . .

alert(*any\_string*); // displays *any\_string* in a pop-up window; a simple string text surrounded by

// single quotes or double quotes

for example

alert(‘Hello World’);

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

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

A recommended place to insert JavaScript is close to the end of the HTML – after the last closing </div> and before the closing </body>.

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

To modify web-site text – specified by identifier – examine the function in utils.js. The identifier in this function is “message”, and you can see how it is used by searching index.html for an instance where “message” appears as an **id** of markup.

JavaScript is case sensitive.

DEBUGGING TIPS

If you believe that you have a JavaScript error . . .

* Right click on the web page to get Chrome’s attention. Chrome displays a pop-up menu.
* Click **Inspect**. Chrome displays a pop-up dialog.
* By default the **Element** tag is selected. Instead, click the **Console** tag. You may see diagnostic text displayed with red text. Sometimes it may not be obvious where the error occurred. Look at the far-right edge of the diagnostic, where the module name and line number will probably be shown.

We have seen that alert() can be used to show diagnostic information on the web-page output. You can similarly use

console.log (*any\_string*);

to place text in the information shown on the browser’s console.

# JavaScript Language

DECLARATIONS

* “var” – to introduce a variable is deprecated. It may still be present in some old JavaScript, but it is inadvisable to use it.
* A variable is declared and set to its initial value at the same time. For example . . .

Let price = 23.45;

Let accountName = ‘Sales’;

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”.

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

OPERATORS

**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. For example

let price = 49.99;

Then the value of (typeof price) is “number”.

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

Use the keywords **“mdn operator precedence”** with Google to find an article that describes JavaScript operator precedence rules.

NUMBERS

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.

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 **“mdn string”** with Google. The **mdn String** article contains a section on **Escape notation**.

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

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

let price = 49.99; // numeric form 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 number to a string, 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

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

OBJECTS

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

let person = {

firstName: ‘John’,

lastName: ‘Adams’

};

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

alert(person.lastName); // displays Adams

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

ARRAYS

A 3-element array is declared . . .

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

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

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.

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

**array0.push(x)** appends the element x 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.

**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).

In addition refer to **indexof()**, **filter()**, **find()**, and **forEach()** for searching, replacing, etc.

CONTROL FLOW

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 !==.

The more common == and != comparison operators are 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.

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

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

If you omit the ‘+’, the truncation is still performed, but the result is a string, instead of a number. ‘+’ does not imply truncation for positive values. I tried it for a negative value, the truncation occurs as one would expect.

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.

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

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

*code\_block*

}

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 *condition* evaluates to **false**; otherwise, you get an infinite loop.

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*.

SCOPE

Variables and constants introduced by **let** and **const** are limited in scope to the code in enclosing braces (if any). This is a very good reason to use **let** and **const** instead of **var**.

FUNCTIONS

The syntax of a function declaration is

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

*body*

}

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.

A function can be declared within an object. Expanding the *person* object (introduced above), we could have

let person = {

firstName: ‘John’,

lastName: ‘Adams’,

showInfo: function() {

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

}

};

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 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).

# Apply JavaScript to a Web Page

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 its class. For example, the web page might contain

. . .

<h2 id="percent\_off">20% OFF</h2>

. . .

(We recently added **id="percent\_off"**, so that it would be possible to refer to this <h2> element in JavaScript.) We change the text with the statement

document.getElementById(‘percent\_off’).textContent = *new\_text*;

It is useful to work with the **class** or **classList** property when using a combination of JavaScript and CSS to change properties of HTML elements. Refer to the code in **home.js** that immediately follows the comment “Use JavaScript coupled with CSS to change the color of some text, and experiment with the classList object.”. Secondly refer to the code in **home.js** that immediately follows the comment “responding to a mouse button click”. (The button is toward the end with the title “SEE REVIEW”.) Finally refer to the console log after executing this code and clicking the button.

Note the syntax of adding an event to a DOM element.

*DOM\_element****.*addEventListener(***event\_name***,** **function() {**

*function\_body*

**}**

*event\_name* depends upon the type of DOM element; you can get a list and descriptions of events from Google; use the keyword “mdn” (Mozilla Development Network) plus a description of the element’s type.

Instead of getting a single HTML element (**document.getElementById()**), we can get a collection of HTML elements via

document. getElementsByClassName(*class\_name*);

This returns an object called an **HTMLCollection** – not exactly an array but similar enough, so that one can use array methods to manipulate the markup.

# 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("4/8/2020 11:35"); // or you could define it with year, month, day, etc.

showMessage (myDate.getFullYear());

Click on **String** in the left-hand column for suggested methods pertaining to strings. 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”.

# Less Disciplined JavaScript

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 string’;

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.