**Module 4: - Introduction and Learning Objectives**

So now you have a website or Cloud App with structure and content from HTML, consistent style from CSS, and a limited amount of interactivity for the user.  To enhance the user experience and create a real wow factor, you’re going to need JavaScript. With JavaScript, you can make your webpages dynamic, using features like interactive forms, picture slideshows, and sophisticated menu systems.

After completing this module, you will be able to:

* Describe the basic syntax of JavaScript.
* Explain how to work with variables in JavaScript.
* Describe how to control the flow of a JavaScript application with control statements.
* Outline the use of functions and prototypes in JavaScript.
* Explain the use of client-side JavaScript with HTML.
* Describe how JavaScript integrates with the document object model (DOM).
* List the common APIs used in JavaScript Applications.

# JavaScript Language: Overview and Syntax

Hello. Welcome to The JavaScript language, Overview and Syntax.

After watching this video, you will be able to:

Describe JavaScript primitives and objects.

JavaScript is a scripting language that is derived from the ECMAScript standard and originally

designed to run on the Netscape Navigator browser:

Virtually all browsers now support JavaScript.

Although the name “Java” appears in the word “JavaScript”, the two languages are

not related.

When a JavaScript interpreter is embedded in a browser, the result is the ability to

create dynamic web pages:

JavaScript adds behavior to otherwise static web content.

The content of a web page can be dynamically changed with the scripting capability of the

JavaScript interpreter.

JavaScript code acts on the document object model that the web browser generates.

One of the ways that server programming and browser scripting work together is in an architecture

that is called Ajax, or Asynchronous JavaScript and XML.

The term "Ajax" encompasses more than asynchronous server calls through JavaScript and XML.

Ajax represents a series of techniques that provide richer, interactive web applications

through HTML, JavaScript, Cascading style sheets, and modifying the web page through

the Document Object Model.

Nowadays, JSON is commonly used instead of XML.

In JavaScript, there are five primitive types that are associated with various primitive

values:

Number: All numbers, such as 0 or 3.1412,

String: All strings, such as “Hello World”,

Boolean: The values true or false,

Null: The value null.

Undefined: The value undefined, since a data type has not been assigned or the variable

does not exist.

All other non-primitive data types are objects.

The number primitive represents both integer and floating point values, the value NaN (not

a number), and Infinity.

Integers can be coded as base 10 (decimal), base 8 (octal), or base 16 (hexadecimal) values.

The integer literal 16 in decimal, 020 in octal, and 0x10 in hexadecimal all have the

same value.

All numbers in JavaScript are represented internally as double precision (64 bit) floating

point numbers.

JavaScript strings are delimited by either double or single quotation marks.

There is no behavior or methods that are associated with primitive data types.

The primitive types number, string, and boolean can be wrapped by their object counterparts.

Wrapper objects have the same name as the primitive type, except they start with an

uppercase letter.

Like most object-oriented programming languages, JavaScript provides built-in ways to convert

between these wrapper objects and primitive values.

The wrapper objects use special methods such as the valueOf and toString methods to convert

between objects and primitive literals.

The typeof keyword in JavaScript is used to find out the data type of the supplied operand.

Notice how strings created without the new keyword have a type of string.

The keyword new is used to create the String wrapper object.

This object can be converted to a primitive string type by calling the valueOf function

on the object wrapper class.

Arrays are specialized collection objects that aid the programmer in the storage and

retrieval of data by indexed keys.

Arrays use a zero-based indexing scheme, meaning that the first element of an array has an

index of zero.

Arrays grow or shrink dynamically by adding or removing elements.

The length property holds the number of elements that occur in the array.

Arrays can be declared by using either an array constructor or an array literal.

When declaring an array with a constructor, you use the new array keywords and specify

the array elements as parameters of the new array.

Array literals are created by declaring the array elements within square brackets.

You then assign the array to a variable, as seen in the last example on the slide.

The Date object is a specialized object that is used to hold the date and time.

The constructor for a date object is in the format: new Date ([with optional parameters]).

If you create a Date object without any parameters, JavaScript returns an object that contains

the current local date and time.

If you send this date object to the console or try to display the date object on a web

page, JavaScript automatically applies a toString method to the object.

The result that is displayed is a string version of the date as shown on this slide.

You can also create new dates by passing parameters to the new Date function.

Examples on the slide show how either string or numeric values can be used for the date

parameters when creating new date objects.

As with other object-oriented languages, JavaScript creates error object instances when an exception

occurs.

The error object instance includes two properties that contain information about the error:

The message property contains a description about the error.

The name property identifies the type of error such as a RangeError.

A RangeError is an error instance that is created when a numeric value or parameter

is outside of its valid range.

Besides a generic error, there are six other core errors types in JavaScript, three of

which are shown on the slide.

The other three are EvalError, ReferenceError, and SyntaxError.

The Error object can be extended to create custom error types.

The last line of the slide shows the creation of a generic error object with a custom error

message in the parameter field.

In this video, you learned:

JavaScript is a scripting language that adds behavior to otherwise static web content.

Primitives are values, and have no properties or methods. Examples of primitives include:

number, string, boolean, null, and undefined.

Wrapper objects allow objects of corresponding primitive data types to be created.

Wrapper objects can store a primitive value and provide methods with which to process

it.

Wrapper objects have the same name as the primitive type, but they begin with a capital

letter to differentiate them from the primitive data type itself.

Some examples of wrapper objects are array, data, and error objects.

# JavaScript: Variables and Control Statements

Hello. Welcome to JavaScript Variables and Control Statements.

After watching this video, you will be able to:

Explain how variables are declared and used in JavaScript,

Describe JavaScript control structures,

Variables are declared with the var keyword followed by the variable name, as in the example,

var age,

Variables can be declared and initialized in one step, as in the example, var age = 54,

You can assign a value to the variable at a later time, or reassign the value of a variable,

Since JavaScript is a loosely typed language, you do not need to declare the data type of

a variable. The variable assumes the data type from the value of the field during assignment

and the type of a variable can change during program execution.

Beware of not initializing variables though – the value of ‘var age’ – that has

not been assigned a value, is not zero, or an empty String, or any other useful default

value; it is ‘undefined’.

Remember that the only keyword is ‘var’, and JavaScript has no way to know whether

this variable is a numeric or a string, or anything else, until you provide a value.

It therefore cannot decide by default whether to assign a zero or an empty string.

Variable names, or identifiers, have these rules:

The name must start with a letter, underscore (\_), or a dollar sign ($),

Subsequent characters can also be digits [0-9] Identifiers are case-sensitive.

Variables also have a scope:

Variables declared within a function have a scope local to that function,

Variables declared outside of a function have a global scope.

Variables with a global scope can be used elsewhere in the JavaScript program,

Variables declared without the var keyword have a global scope.

Variables that are not initialized have a value of undefined.

You learn about functions shortly.

Conditional statements are the set of commands that are used to perform different actions

for different conditions:

In JavaScript, the IF statement is the way the program logic decides which path to take

based on the current values of variables or object properties,

The JavaScript syntax for a decision begins with the keyword IF, followed by the condition

to test; then the statements that run if the condition yields a true result.

The true processing follows immediately after the test condition and is delimited by braces,

unless it is a single statement.

If the condition resolves to false, the statements that follow the else keyword are executed.

Indentation of the statements of a compound control statement is not required in JavaScript

However, a programmer can find that indentation helps in deciphering the control statement.

Many JavaScript aware text editors automatically indent the control structure to make it more

readable.

Unlike Java, there is no block statement scope in JavaScript.

Having no block statement scope means that variables declared inside one IF condition

can be used outside the scope of that condition.

JavaScript supports the switch statement as an alternative to the IF then ELSE control

statements.

The condition that is being tested is placed in parentheses that follow the switch keyword.

The expression parameter of the switch statement can evaluate to any number or string value.

The labels in the case statement are enclosed in quotation marks when the labels represent

string values of the expression.

The program looks for a case clause with a label that matches the value of expression

and then transfers control to that clause, running the associated statements.

If no matching label is found, the program looks for the optional default clause, and

if found, transfers control to that clause, issuing the associated statements.

The break keyword is used to prevent the code from automatically falling into the next case

clause.

The FOR loop repeats a series of statements for any number of times.

The FOR loop takes three parameter arguments, namely the initial value, the condition that

is being tested, and the increment expression.

When the FOR loop runs, the following occurs:

First, the initial expression is set,

Second, the conditional expression is evaluated.

If this condition evaluates to true, the loop statements runs and the increment expression

is updated.

If the condition evaluates to false, the FOR loop terminates.

If the loop does not terminate, control returns to the second step and the conditional expression

is evaluated again.

The WHILE loop is another common loop in JavaScript,

The loop repeats while the condition remains true,

The WHILE loop assumes that the condition reaches a conclusion and then exits the loop.

Make sure that the condition evaluates to false at some point; otherwise the loop never

terminates.

In this video, you learned:

Variables are declared using the keyword var followed by the variable name.

You can initialize variables at the time of declaration or assign a value later.

You do not need to declare the data type of a variable. Variables take their data type

from the value assigned and can change type during program execution.

Variables also have a scope .

Variables declared within a function have a scope local to that function.

Variables declared outside of a function have a global scope.

Variables declared without the var keyword have a global scope.

When executing, the flow of the program is directed by control statements, including:

Conditional statements like if...then...else,

Switch statements,

Repeat statements like for loops and while loops.

# JavaScript: Functions and Prototypes

Hello. Welcome to JavaScript Functions and Prototypes.

After watching this video, you will be able to:

Describe functions in JavaScript, and

Describe prototypes in JavaScript.

A function is a block of code which can be called from any point in a script after being

declared.

A function is made up of the following parts:

The keyword function,

The name of the function,

Parentheses, with optional parameter arguments,

Curly braces, with the logic,

The last statement in a function block is the optional return statement that returns

control back to whatever called the function.

This example is a function that is named add.

This function takes two parameter arguments and returns the sum of the arguments or concatenates

the two arguments if they are strings.

Notice that you do not specify the data type for the arguments of the function.

The data types are determined by the values of the arguments that are being passed to

the function.

There is no specific return type declared – the function returns whatever type is

required.

In this case, the return is a simple addition or concatenation of the input parameters.

The action chosen depends on what data is provided to the function.

If the values can be interpreted as numbers, they are added.

If they can be interpreted as strings, they are concatenated.

This is an example of declaring a function named Car that accepts three arguments as

parameters

A little further down, you specifically ask the function to run, by declaring the statement:

var c = new Car with the arguments “meridian”, “Sabre GT” and 2012.

In the Car function, the keyword "this" refers to the current instance of the Car object

that is being created. In other words, an instance of Car that is associated with the

variable named c.

The getName function of Car returns the make, model, and year of the newly created Car object.

By using prototypes, you can easily define properties and methods for all instances of

a particular object.

Prototypes exist for all JavaScript objects that can be created with the new keyword.

All object constructors create objects that inherit the properties and methods that are

defined by the prototype for that object.

Any object that gets instantiated inherits the current state of the prototype.

Scripts can override prototype properties and functions.

If they do so, these changes affect the current working instances of objects that match the

prototype.

In the Car example that is shown on the slide, you can add another property to the Car prototype,

with the statement:

Car.prototype.property\_name as in:

Car.prototype.floor\_model = true;

Any Car object that you create, or any car object that is already created, automatically

inherits the new floor\_model property and its value.

Here is another example that uses a prototype to change Car instances.

This time, you add a method function called getName to the Car prototype.

Now when a Car object is instantiated, it also includes the getName function that returns

the make, model, and year.

All existing instances of the Car object also inherit the getName method.

Functions are usually declared first, and not run until you specifically ask them to,

as you saw in some of the previous examples.

Auto-invocation or self-executing functions start running immediately after being declared.

The functions and variables inside self-executing functions are only available to the code inside

the self-executing function.

Auto-invocation functions can also be unnamed or anonymous functions, and have the format

that is shown in the code block on the slide.

Self-executing functions are often used to initialize data or to declare DOM elements

on the page.

In this video, you learned:

A function is a block of code which can be called from any point in a script after being

declared.

Functions can take arguments passed as parameters and can return results.

By using prototypes, you can easily define properties and methods for all instances of

a specific object.

Prototypes exist for all JavaScript objects that can be created with the new keyword.

To add a new function to the template for the object, modify the prototype for the object.

Self-executing (auto-invocation) functions start running immediately after they have

been declared.

The functions and variables are isolated from the rest of the script.

# Client-Side JavaScript: with HTML

Hello. Welcome to Client-side JavaScript.

After watching this video, you will be able to:

Define a client-side script,

Give examples of when client-side script can be used,

Explain how to use the <noscript> tag,

Describe Event Binding in scripts.

A client-side script is a program that accompanies an HTML document or might be embedded directly

in the HTML document itself.

The script program runs on the client device when the document loads, or at some other

time such as when a link is activated or when a button is clicked.

Although JavaScript is widely used as a scripting language in HTML, other scripting languages

can be used instead.

Scripts offer authors a means to modify and extend HTML documents in highly interactive

ways.

Scripts can run after an HTML document is loaded.

Scripts can be used to validate forms or to process input as it is typed.

Scripts can be triggered by events that occur on a web page, such as the clicking of a button.

Scripts can be used to dynamically create document elements on an HTML page.

This slide shows two ways in which the <script> tag is used to include scripts in an HTML

document.

Example 1 demonstrates how you can include a script directly inside the HTML document.

This method is good for short scripts, but when a script is long, the method used in

example 2 is preferred.

Example 2 uses the src attribute to point to an external script file.

This method has several use cases, such as importing JavaScript libraries for complex

interactions or using the same script across several HTML documents.

Some users who visit your website might disable JavaScript from running, or they might be

using a browser that does not support scripting.

To allow for these situations, place the content for the alternative path within the <noscript>

tag.

If the browser does not support scripting, the browser runs the section of code that

is within the noscript tag.

Scripts can be run on the detection of certain events that happen when the page is running

in a browser session.

For example, the onload event can run a script when the browser finishes loading a page.

Or a function can be performed when the onclick event occurs.

This event occurs when the pointing device is clicked over an element, such as a button,

that declares a handler for the event.

The event handler is a function that declares what to do when the button is clicked.

Here the inline event handler named showAnswers runs when the button is clicked.

In this video, you learned:

A client-side script is a program that accompanies an HTML document.

It may even be embedded into the HTML itself.

Scripts offer developers ways to extend HTML documents to enhance the user experience,

especially by incorporating more interactive elements.

You can use the script tag to include a script within the HTML document, or to call a script

from an external file.

Use the noscript tag to provide an alternative when scripting is disabled.

Scripts can be bound to events so that they run automatically.

For example, the onload event can run a script when the browser finishes loading a page.

# Client-Side JavaScript: with DOM

Hello. Welcome to Client-side JavaScript.

After watching this video, you will be able to:

Describe the document object model (DOM) hierarchy.

Describe the window and document objects.

Identify the DOM objects that are commonly used in JavaScript applications for working

with HTML documents.

The document object model is the programming interface between HTML or XHTML and JavaScript.

The Document Object Model (DOM) is a browser-based interface for applications and scripts to

dynamically access and update the content, structure, and style of documents.

JavaScript uses the DOM to access and modify web page elements in the web browser.

The World Wide Web Consortium released four levels of the Document Object Model specifications.

Each successive level provides a more detailed set of features for describing structured

documents.

The different browsers have various levels of compatibility with the DOM standard.

The discussion of the DOM in this unit focuses on the DOM Level 1 Core and DOM Level 1 HTML

APIs to access HTML elements with the DOM.

The JavaScript engine in most web browsers fully supports DOM level 1.

Here is a representation of the basic DOM model for browsers:

The window object is at the top of the DOM hierarchy and controls the environment that

contains the document.

The history object keeps internal details about the recent history of pages in the browser.

The history object has methods for letting you simulate clicking the back or forward

buttons in a browser.

The location object contains information about the URL of a page.

The navigator is an object representation of the client Internet browser (user agent).

There is no standard that applies to the navigator object, so the property values returned when

running queries on the navigator object are not consistent across browsers.

The screen object is used to derive information about a user's screen, such as the dimensions

of the display screen.

The screen object is useful for determining the screen size of browser windows that run

on mobile devices.

The document object provides access to all HTML elements within a page.

Each HTML document that gets loaded into a window becomes a document object.

The window object is the outermost global container of all the objects in the DOM hierarchy.

When the browser loads a page, a window object is automatically created for you.

You can then access the window object properties and functions from your JavaScript code.

In client-side JavaScript, the Window object serves as the global object and everything

in the DOM takes place in a window.

A number of predefined methods exist for the window object.

The window.alert, window.confirm, and window.prompt dialogs that are used in web pages, come from

the global window object.

You can leave out the window prefix for methods in the DOM API. So the window.alert method

can be coded more simply as alert with a message argument.

This figure shows the object model for a simple HTML document.

Notice how the object hierarchy matches the HTML containment hierarchy on the left.

The object diagram can also be represented as a tree structure that corresponds to the

structure of the HTML document.

The branches of the tree structure are termed nodes.

There are two types of nodes in the W3C DOM, element nodes and text nodes.

All HTML tags (html, head, meta, title, and body) are element nodes.

The nodes that contain actual text that go between an element start tag and end tag,

are text nodes.

The figure shows the DOM level 2 tree for the FORM portion of the document.

The line feeds between elements are text nodes and are part of the DOM level 2 tree.

The DOM level 2 tree includes a line feed text node before the paragraph and input elements.

The input element includes a text node that contains all the text that follows the input

tag.

An additional line feed text node follows the input element.

The DOM level 0 for the form portion of the document would have only the form, p, and

input boxes.

The DOM level 2 adds the carriage returns, tabs, and spaces, sometimes referred to as

"white space".

In this video, you learned:

The document object model (DOM) is the programming interface between HTML or XHTML and JavaScript.

Each successive level of the DOM provides a more detailed set of features for describing

structured documents.

Different browsers have different levels of compatibility with the DOM standard.

The basic DOM for browsers is a hierarchy that includes objects that perform different

functions.

For example, the window object controls the environment of the document, the location

object contains information about the URL of the page, screen object derives information

about the user’s screen, and the document object provides access to all HTML elements

within a page.

DOM levels define object types, with which developers can build a variety of documents,

from plain HTML documents for web pages, to more complex forms.

# JavaScript DOM Objects

Hello. Welcome to JavaScript Dom Objects

After watching this video, you will be able to:

Identify the DOM objects that are commonly used in JavaScript applications for working

with HTML documents.

The W3C DOM level 2 defines 12 different types of nodes, seven of which have direct applicability

in HTML documents.

Node types that are not applicable to HTML are omitted from the table.

This table is of interest so that when you view a DOM tree, you understand the meaning

of the numeric node types.

Each node type is a named constant that is also represented by an integer value.

So, for example:

An ELEMENT\_NODE type is represented by the integer 1.

An ATTRIBUTE\_NODE type is represented by the integer 2.

A TEXT\_NODE type is represented by the integer 3.

A COMMENT\_NODE is represented by the integer 8.

In the DOM tree, the node name for ELEMENT\_NODE type is the name of the element or tag.

For example, if you are looking at a DIV element, the node name is DIV.

If the DIV element has an attribute like id=div123, then the attribute name is “id” and the

attribute value is “div123”, which is the name-value pair.

Another example:

If a paragraph element is followed by some text, the text string has a node name of hash-

sign-text, and the node value is the text string itself.

The table lists some of the DOM Level 2 node object properties and corresponding data types.

You see these properties when you view a DOM tree for an HTML page in a browser’s developer

tools, like Chrome’s DevTools.

How do you access the elements on an HTML page with the DOM API?

When the document is loaded, the browser creates arrays for forms, images, anchors, links,

applets, and embeds.

It then places all the objects of each type into these arrays.

The arrays are indexed as they occur in the source document. The first index of each array

starts at zero.

Each of the array types, such as forms[], contains an array of elements[] with each

index element being the fields or buttons that occur in that form.

You can reference the element named field1 in the figure by its relative position as

document.forms[0].elements[0].

You can reference the same field with named elements; for example, document.forms["form1"]

.elements["field1"], or even the shortened document.form1.field1.

Since only one document can be contained in a window, shown by the dashed line in the

figure, you can leave out the window prefix. However, you cannot omit the document prefix

from the referenced object.

The id attribute identifies an element in a document.

The id attribute of an element is used by scripts to refer to the element with a name

that matches the value of the id attribute.

In order to assign a scriptable reference name to an HTML element with the id attribute,

use the following conventions:

The id must be a unique name in the document.

The name must be in quotation marks when assigned to the id attribute.

The name must not start with a numeric digit.

The function that is used to return a node object that matches the id value is document.getElementById

with the name of the id as a parameter argument.

It is suggested that the same value is used for both the id and the name attribute (seen

earlier) when both are used.

In this video, you learned:

How to work with DOM objects in JavaScript, including:

You can access nested objects using a dot notation.

Objects can be named to make accessing them from the script easier.

# JavaScript APIs

Hello. Welcome to JavaScript APIs.

After watching this video, you will be able to:

Explain how to work with nodes,

Describe how to modify the content of an element,

Explain how to modify the inline style of an object,

Describe how to modify the attributes of an object,

Describe how to use Window object methods and events.

The DOM Level 1 core and Level 1 HTML specifications have too much detail for one unit.

Instead, the remainder of the unit focuses on script functions and properties that are

commonly used when working with HTML pages.

Some of these common APIs used in when accessing HTML DOM elements in web pages are seen on

the slide.

DOM elements are reviewed next.

To retrieve a node reference for an element of a document given an id, use the document.getElementById

function and pass the id value as an argument.

This returns one specific HTML or XML element if a matching unique id is found on the web

page.

The getElementsByTagName function retrieves a NodeList of elements with a specified tag

name.

The NodeList contains an array of elements in your document.

You provide the name of the element you are interested in and an array of all elements

with that name found in the document is returned.

For HTML elements, the tagName parameter is the literal name of the HTML tag.

If you run the function getElementsByTagName with a “p” as a parameter argument, a

NodeList of all the paragraphs in the document is returned.

This example shows how you can retrieve all the image elements from a web page by using

the getElementsByTagName function and passing “img” as the parameter argument.

The result is a NodeList that is assigned to the variable imgSet.

In the next part of the code, you loop through the nodeList and concatenate the results into

a DOM output stream.

The src attribute is a common property for the img tag. The src attribute indicates the

location of the image source.

The document.write() function adds the script-generated HTML to the document.

You can use the DOM API function document.createElement(TagName) to create an element in the current document.

After creating the element, you can use any of a number of functions to place the element

in the appropriate location within the document.

Examples of these functions include the insertBefore, appendChild, or replaceChild function that

can be used to add the newly created element into the document.

This example shows the source code for adding a node to a document.

A new paragraph element is being created that includes a text node with the string “Hello

world!”

The text node is then appended as a child of the paragraph element.

Finally, the entire paragraph with text is appended as a child node at the end of the

body node of the HTML page.

The function element.innerHTML retrieves or sets the contents of an HTML element.

The innerHTML property returns all child elements as a text string.

With the element.innerHTML function, you can change the contents of an HTML element, by

setting it to a text string that can include HTML tags.

Setting the innerHTML value of an element to a string removes all of the current child

elements. The browser then parses the string and sets the contents of the HTML element.

You can use the element.style method to retrieve or set the inline CSS style for a particular

element.

If you use element.style to set the style of an element, it overrides any setting from

a CSS style sheet with one specific style.

The way to set the style in JavaScript is with the format element.style.propertyName

= value.

For example, if you had an element <div style="color:blue">.

Here the <div> tag is used to group block-elements to format them with a color style. You can

change the style of this div tag with the JavaScript statement:

div.style.color = 'red';

In contrast, the element.setAttribute('style', …) wipes out all previously set inline CSS

styles.

The function element.setAttribute with parameters(attrName, attrValue) dynamically modifies the attribute

of an element.

In the example, the src attribute of an element with an id of theImage is set to a different

target image.

The function element.removeAttribute(attrName) removes an attribute from an element.

The function element.getAttribute(attrName) retrieves the value of the specified attribute

in the element, if it exists.

Here are some window object functions and events.

To open a new browser window, use the window.open() function.

This method returns a reference to the new window object.

You can use this reference later to close the window, with the reference\_name followed

by the close() function.

The parameters of the window.open function are:

URL - A string that indicates the location of the web page to be displayed in the new

window.

You can pass an empty string if you are going to write some script- generated content to

the new window in the current URL context.

Name - A string that specifies the name of the window.

Features - An optional string that specifies the features of the window, such as its placement

and dimensions. The features string is a comma-separated list of name- value pairs.

Replace – An optional boolean value. If true, the new location replaces the current

page in the browser history.

The window.onload function can be used to start a function after the page is loaded.

The function, window.dump("message") writes a string into the console for the web browser.

The dump() function is a less intrusive way to display diagnostic information than the

alert() method.

Finally, the window.scrollTo(x-value, y-value) scrolls the web browser to a particular set

of coordinates on a page.

The onload event handler runs in the current window after the document loads the web page.

In the example, the onload event causes an anonymous function to run. This function,

in turn, runs the function addPara().

In this video, you learned:

You can retrieve a reference to a node using:

document.getElementById(id) - Returns one specific element that is based on the id attribute.

document.getElementsByTagName(tagName) - retrieves a of elements with the specified tag.

You can create an element using:

document.createElement(TagName).

And place it using:

insertBefore, appendChild, or replaceChild.

You can modify elements using:

element.innerHTML to retrieve or set the contents of an HTML element.

element.style to retrieve or set the inline CSS style.

element.setAttribute to modify the attributes of an element.

You can manage a window object using functions including:

window.open to return reference to a new window object for the web browser.

window.dump("message") to write a string into the console for the web browser.

This is not an exhaustive list – many more functions exist that enable you to work with

HTML elements and nodes.

Window Object Methods & Events

HANDS-ON LAB: JavaScript

Use the Start Lab button below to launch the lab in a new browser tab.

[Note: In case the Start Lab button does not work, instructions to complete this lab are also available here](https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-CD0101EN-SkillsNetwork/labs/Theia%20Labs/03%20-%20Javascript/instructions.md.html).

## Hands-On Lab: JavaScript (External resource)

Clicking on the Start Lab button below will launch the cloud based SN labs virtual labs environment with instructions to complete this lab. Your username and email will be shared with SN Labs to authenticate and provision your lab environment.

**Step 1 of 1**

# Lab: Validating a JavaScript form

**Estimated time** 45 minutes

JavaScript is a client-side scripting language and is commonly used to create dynamic web pages. It helps you change web page contents dynamically, as well as enabling you to validate forms and perform other actions. In this lab, you will create an HTML form that uses JavaScript to validate input.

## Objectives

After completing this lab, you will be able to:

1. Create a basic web form
2. Add the <script> tag
3. Add a function
4. Access the form controls from JavaScript
5. Access a textbox and check if it is blank
6. Execute a set of statements based on a condition
7. Display error messages
8. Execute a function when the form is submitted

## 1. Create an HTML form

In this exercise, you will create a simple form that accepts a person's name and email ID and then performs a simple validation on the entered input.

On the window to the right, click on **File > New File**. A **New File** window opens. Enter form\_validation.html as file name and click **OK**. You are now ready to start creating the new form.

Let's start by creating a simple form designed to accept the user's name and e-mail ID. The form will have a **Submit** button and a **Reset** button.

Copy and paste the following code into your file to create the initial form without validation:

<!DOCTYPE **html**>

<**html**>

<**head**>

<**title**>Contact Details</**title**>

</**head**>

<**body**>

<**h2**>Enter your contact Details:</**h2**> <**br**>

<**form** id="form1">

<**label** for="name">Name :</**label**>

<**input** type="text" id="name" name="name">

<**br**>

<**br**>

<**label** for="email">E-mail ID :</**label**>

<**input** type="text" id="email" name="email">

<**br**>

<**br**>

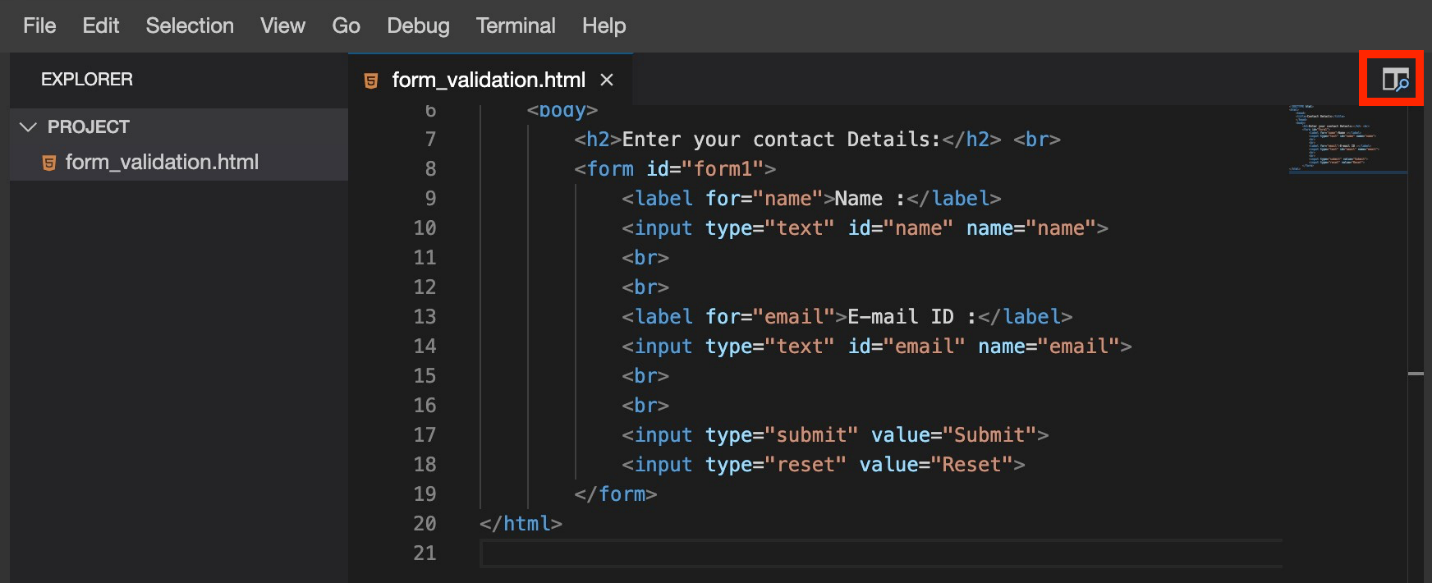
<**input** type="submit" value="Submit">

<**input** type="reset" value="Reset">

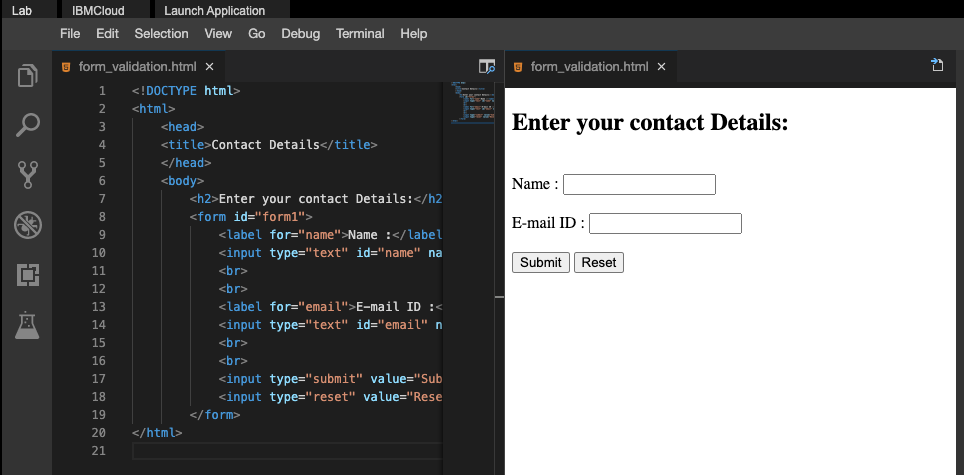
</**form**>

</**html**>

When you have pasted the code, save your file. To see how your HTML page will display, click the preview icon at the top right of the window.



Your page should look like this:



## 2. Add the <script> tag

We use the <script> tag to embed executable code, usually JavaScript, into an HTML page. The tag can contain scripting statements, or it can refer to an external script file. We use a type attribute to specify the scripting language.

Although you can put the <script> tag anywhere in your HTML document, for this lab you'll put it in the <head> section.

Replace the <head> section of your file with the following code. It tells the browser that the code we are about to put inside the <script> tag must to be executed as JavaScript.

<**head**>

<**script** type="application/javascript"></**script**>

<**title**>Contact Details</**title**>

</**head**>

## 3. Add a function

Now you'll specify what happens when a user clicks the **Submit** button. We specify this behavior with a user-defined JavaScript function, which is a block of code that is executed when it's called. A function can be called any number of times.

A function in JavaScript looks like this:

**function** **functon\_name**()

{

// code goes here

}

Let's add an empty function that has the name checkdata. Replace the <script> tags in your file with the following code:

<**script** type="application/javascript">

**function** **checkdata**()

{

}

</**script**>

## 4. Access HTML controls within JavaScript

The function you've created is intended to validate the contents of each of the input elements in the form. To access the data for an element, the script needs to refer to the correct element.

One way to identify an element is to use a method called getElementByID(elementID). The following line of code returns the element with the ID name:

document.getElementByID("name");

The following lines of code enable you to access the name and email input elements of the form. The references to the elements are stored in two JavaScript variables named username and emailid.

**var** username = document.getElementById("name");

**var** emailid = document.getElementById("email");

## 5. Access and check data

When the references to the elements are stored in the variables, the values of the elements can be retrieved using the value attribute. If username is the variable that contains the input element's reference, then its value can be accessed using

username.value

To check if this value is blank, we can use the following statement:

username.value == ""

"" indicates an empty string.

## 6. Execute a set of statements based on a condition

If the value is blank, we will print an error message and return the focus back to the empty element.

To perform this action, we use a JavaScript conditional statement called the if statement. The if conditional statement allows us to specify a block of code to be executed if a condition is true.

The syntax of the statement is as follows:

**if**(condition){

//block of code to be executed, if the condition is true.

}

Let's check if the username value is empty by using an if statement:

**if**(username.**value**==""){

**return** false;

}

If the value is blank, the return false; statement returns a boolean value false from the checkdata function that we added in step 3.

We check all input elements of the form in this way to determine whether they are empty.

## 7. Display error messages

You can display a message to a user with the help of a pop-up alert message box. To do this, you will use the alert method.

Let's use this method within the function to alert the user.

**if**(username.**value**==""){

alert("Please enter the name");

fname.focus();

**return** false;

}

The fname.focus() statement is used to bring the input focus back to the element where we found a problem, in this case, name.

We indicate that none of the elements are blank by returning true. So, we need to add a return true statement at the end of the function.

It's a good practice to include comments in your code. Comments will help you and other programmers easily debug any errors that we might encounter while running the code. In JavasScript, we add comments using two forward slashes: //

Our final checkdata function with comments added looks like:

**function** **checkdata**(){

//Create references to the input elements we wish to validate

**var** username = document.getElementById("name");

**var** emailid = document.getElementById("email");

//Check if username field is empty

**if**(username.value == ""){

alert("Please enter the name");

username.focus();

**return** false;

}

//Check if email field is empty

**if**(emailid.value == ""){

alert("Please enter the email");

emailid.focus();

**return** false;

}

//If all is well return true.

**return** true;

}

## 8. Execute a function when the form is submitted

Our final step is to ensure that the checkdata function is executed when the form is submitted. We do this using the onsubmit event. This event occurs when users click the **Submit** button.

The following code links the onsubmit event to the checkdata function:

<**form** id="form1" onsubmit="return checkdata()">

{:codeblock}

This code ensures that the checkdata function is invoked when the form is submitted.

Following is the complete code along with the HTML form and JavaScript validation function. Copy and paste the code into your file and check it to determine if it is properly validating:

<!DOCTYPE **html**>

<**html**>

<**head**>

<**title**>Contact Details</**title**>

<**script** type="application/javascript">

**function** **checkdata**(){

//create references to the input elements we wish to validate

**var** username = document.getElementById("name");

**var** emailid = document.getElementById("email");

//Check if username field is empty

**if**(username.value == ""){

alert("Please enter the name");

username.focus();

**return** false;

}

//Check if email field is empty

**if**(emailid.value == ""){

alert("Please enter the email");

emailid.focus();

**return** false;

}

//If all is well return true.

alert("Form validation is successful.")

**return** true;

}

</**script**>

</**head**>

<**body**>

<**h2**>Enter your contact Details:</**h2**> <**br**>

<**form** id="form1" onsubmit="return checkdata()">

<**label** for="name">Name :</**label**>

<**input** type="text" id="name" name="name">

<**br**>

<**br**>

<**label** for="email">E-mail ID :</**label**>

<**input** type="email" id="email" name="email">

<**br**>

<**br**>

<**input** type="submit" value="Submit">

<**input** type="reset" value="Reset">

</**form**>

</**html**>

## Summary

Congratulations! You have now learned how to create a form and validate its user inputs. As additional practice, we encourage you to add a numeric field such as "zipcode" and then validate it.

## Tutorial details

**Author:** Ramesh Sannareddy

**Change log**

| **Date** | **Version** | **Changed by** | **Change Description** |
| --- | --- | --- | --- |
| 2020-08-13 | 1.0 | Ramesh Sannareddy | Initial version created |
|  |  |  |  |
|  |  |  |  |

# Hands-on Lab: Javascript - Browser Console

Logo

Description automatically generated

# Hands-on lab on Javascript

The purpose of this lab is to javascript before you set off doing server side coding with Node JS. This lab presumes that you have completed all the other labs in the course IBM HTML CSS and JS for Web development.

#### **Duration (25 mins)**

### **Objective**

After completing this lab you will be able to:

1. Write and run Javascript on the browser console
2. Create variables, work with conditional statements, create loops and define methods in Javascript.

## **Task 1 - Open the browser console.**

In this task, we are going to run the Javascript code in the browser console. The Chrome browser has a v8, which is Google's open source high-performance JavaScript engine.

1. Open a new blank browser page clicking on Ctrl+T(Windows) or Command+T(Mac) to open a new tab.
2. Right-click anywhere on the new blank browser tab and choose **Inspect** or **Inspect Element** depending on the browser you are using. The image below is for Chrome Broswer.

Graphical user interface, application

Description automatically generated

1. Go to the **Console** tab, as shown below. You will see a command prompt. You can run the rest of the tasks there.

Graphical user interface, text

Description automatically generated

1. If your console has any logs printed, clear it by running the following command. This is not mandatory. It will just help in a fresh start.

clear()

## **Task 2 - Running JS commands**

**NOTE:** At any point of time, when you want to clear the console run the clear() command.

To run the commands we will use the command prompt on the browser control. Type or paste the command and press **enter** to run the command.

1. Let's start with a simple code to print **Hello World!** to the console. Run the following command.

console.log("Hello World!")

The output would be as below.

Graphical user interface, application

Description automatically generated

The undefined means console.log doesn't return anything.

1. Let's create some variable and print them. Run the following command.

**let** num = 5

**var** mystr = "John"

console.log(num)

console.log(mystr)

Both let and var can be used to create variables. var is used when you want the variable to have global scope and let is used when you want the variable to have scope within the block where it is created.

1. Let's create a constant and print it. Run the following command.

**const** pi\_val = 3.147

console.log(pi\_val)

Const is used to declare variable whose values can never change

1. Let's create function which prints any value that is input to it.

**function** **printMyInput**(user\_input) {

console.log("The parameter passed is "+user\_input)

}

1. Call the function you created in the previous step once with a number and once with a string.

printMyInput(9)

printMyInput("John")

1. Let's rewrite the function printMyInput according to the ES6 standard. This syntax is also called arrow functions and provide a shorthand to write functions.

**let** printMyInputES6 = (user\_input)=>{

console.log(user\_input)

}

1. Call the function you created in the previous step once with a number and once with a string.

printMyInputES6(9)

printMyInputES6("John")

Since the function is passed a single value and the body of the function is a single line, the brackets can be omitted. The code can also be written as below.

**let** printMyInputES6Short = user\_input => console.log(user\_input)

Now when we call it, the output should remain the same.

printMyInputES6(9)

printMyInputES6("John")

## **Task3 - Operators, Conditions, Loops**

In this task you will be running some javascripts from which you can learn how to use operators, controls and loops.

Ensure that you understand the code in each file. These are primitive and foundational for your understanding of JS

1. **Arithmetic operators** are operators that we use to perform arithmetic operations.
   * + (plus) operator is used to add
   * - (minus) operator is used to subtract)
   * \* (star or asterisk) operator is used to multiply
   * / (slash) operator is used to divide
   * \*\* (double star) operator is used for exponentiation/power
   * % (percentage) operator is used for modulus operation

console.log("5 + 3 = ",5+3)

console.log("7 - 3 = ",7-3)

console.log("8 \* 2 = ",8\*2)

console.log("27 / 3 = ",27/3)

console.log("4 power 3 = ",4 \*\* 3)

console.log("19 mod 4 = ",19%4)

1. **Assignment operators** are operators that are used to assign values to variables
   * = operator is used to assign value on the right to the variable on left
   * += operator is used to increment the value stored in the left operand by the value of the right operand and store it back to the left operand
   * -= operator is used to decrement the value stored in the left operand by the value of the right operand and store it back to the left operand
   * \*= operator is used to multiply the value stored in the left operand by the value of the right operand and store it back to the left operand
   * /= operator is used to divide the value stored in the left operand by the value of the right operand and store it back to the left operand
   * \*\*= operator is used to raise the value stored in the left operand to the power value of the right operand and store it back to the left operand
   * %= operator is used to get modulus of the value stored in the left operand by value of the right operand and store it back to the left operand

x = 5

console.log("Old value x ",x)

x += 3

console.log("New value x ",x)

y = 5

console.log("Old value y ",y)

y -= 3

console.log("New value y ",y)

a = 6

console.log("Old value a ",a)

a \*= 3

console.log("New value a ",a)

b = 6

console.log("Old value b ",b)

b /= 3

console.log("New value b ",b)

c = 6

console.log("Old value c ",c)

c %= 3

console.log("New value c ",c)

d = 6

console.log("Old value d ",d)

d \*\*= 3

console.log("New value d ",d)

1. **Comparison Operators** are used to compare values or variables against values or other variables

* == operator checks if the operand on the left is of equal value to the operand on right
* === operator checks if the operand on the left is of equal value and equal type to the operand on right
* != operator checks if the operand on the left is not of equal value to the operand on right
* > operator checks if the operand on the left is greater than that on the right
* < operator checks if the operand on the left is lesser than that on the right
* >= operator checks if the operand on the left is greater than or equal to that on the right
* <= operator checks if the operand on the left is lesser than or equal to that on the right

//Checking equality of 5 number type and 5 string type

console.log("5=='5' ", 5=='5')

console.log("5==='5' ", 5==='5')

console.log("5===5 ", 5===5)

console.log("5 != 5 ", 5 !== 5)

console.log("5 != 6 ", 5 != 6)

console.log("5 != '5' ", 5 !== '5')

console.log("5 > 2 ", 5 > 2)

console.log("5 > 7 ", 5 > 7)

console.log("5 > 5 ", 5 > 5)

console.log("5 < 7 ", 5 < 7)

console.log("5 < 2 ", 5 < 2)

console.log("5 < 5 ", 5 < 5)

console.log("5 >= 5 ", 5 >= 5 )

console.log("5 <= 5 ", 5 <= 5 )

1. **Logical Operators** are used to combine more than one conditions.

* && operator checks if the condition on left and right are true. Returns true only of both conditions are true. Else returns false.
* || operators checked if either the condition on the left is true or right is true. Returns true even if one of the two conditions is true.
* ! operator checks if the condition is not met.

Practice exercises for logical operators will be covered along with if-else conditions.

1. **if-else-else if** Conditional statements are very useful to control the flow of your code.

//Accept a input from the user. If it is a number print the multiplication table for the number. Else print the input and the length of the input.

**let** user\_input = prompt('Enter a value');

//check if the user input is not a number

**if**(isNaN(user\_input)) {

console.log("Your input is ",user\_input)

console.log("The length of your input is ",user\_input.length)

} **else** {

console.log(user\_input, " X 1 = ",user\_input\*1)

console.log(user\_input, " X 2 = ",user\_input\*2)

console.log(user\_input, " X 3 = ",user\_input\*3)

console.log(user\_input, " X 4 = ",user\_input\*4)

console.log(user\_input, " X 5 = ",user\_input\*5)

console.log(user\_input, " X 6 = ",user\_input\*6)

console.log(user\_input, " X 7 = ",user\_input\*7)

console.log(user\_input, " X 8 = ",user\_input\*8)

console.log(user\_input, " X 9 = ",user\_input\*9)

console.log(user\_input, " X 10 = ",user\_input\*10)

}

1. **Loops** can be used when the same block of code is to be executed many times.

**for loops** have an initial value, condition based on which the loop is executed and an incremental value.

//Accept a input from the user. If it is a number print the multiplication table for the number.

**let** user\_input = prompt('Enter a number');

//check if the user input is not a number

**if**(!isNaN(user\_input)) {

//Using for loop for the repitive statement

**for** (**let** i=0;i<10;i++) {

console.log(user\_input, " X ",i," = ",user\_input\*i)

}

}

**while loops** have just a condition based on which a block of code is executed many times.

//The code below is to find the length of the words the user is entering. The loop will go on and on until the user chooses not to continue by pressing 'n'

**let** do\_more = true

**while**(do\_more) {

//Accept a input from the user.

**let** user\_input = prompt('Enter a word');

//check if the user input is not a number and then print the length of the input

**if**(isNaN(user\_input)) {

console.log("Length of the word you entered is "+user\_input.length)

} **else** {

console.log("You entered a number. Only words are allowed")

}

**let** should\_continue = prompt("Do you want to continue. Press n to stop")

**if**(should\_continue === "n") {

do\_more = false

}

}

1. **switch-case** is used to replace multiple if-else if conditions checking the same variable. After one of the conditions is satisfied and the block of code is executed, the control should explicitly **break** out of the switch block.

**let** user\_input = prompt('Enter a number between 1 to 7');

//Using logical OR operator to check if the input is a number and it is between 1 to 7

**if**(isNaN(user\_input) || user\_input< 1 || user\_input>7) {

console.log("Invalid input")

} **else** {

user\_input = parseInt(user\_input)

**switch**(user\_input){

**case** 1: console.log("Sunday");**break**;

**case** 2: console.log("Monday");**break**;

**case** 3: console.log("Tuesday");**break**;

**case** 4: console.log("Wednesday");**break**;

**case** 5: console.log("Thursday");**break**;

**case** 6: console.log("Friday");**break**;

**case** 7: console.log("Saturday");**break**;

**default**: console.log("Invalid entry");

}

}

## **Task 4 - Collections**

1. Array is an indexed collection. The index positions are from 0. To access the element in first position, we use 0, second position will be 1 and so on. The index of the last position will always be one less than the length of the array.

**let** myArray = ["Jack","Jill",4,5,true,"John"]

console.log(myArray[0]);

console.log(myArray[5]);

1. To iterare through arrays there is a special for loop, **forEach**, which executed for each value in

**let** myArray = ["Jack","Jill",4,5,true,"John"]

myArray.forEach(x => {

console.log(x)

})

1. To find the index position and the value, we can use the generic Object.entries method, which can be used with all collection objects. This maps each index position to the value.

**let** myArray = ["Jack","Jill",4,5,true,"John"]

**for** (**const** [idx, value] **of** Object.entries(myArray)) {

console.log(idx," - ",value);

}

1. **Map** object maps a key to a value. The keys have to be unique. The values can be string, int, float or any other valid javascript datatype. An empty Map object can be create with the new keyword.

**let** myMap = **new** Map();

//Here name is key and John is the value.

myMap.set("name","John")

//Here age is the key and 22 is the value.

myMap.set("Age",22)

myMap.forEach((val,key) => {

console.log(key," - ",val)

})

## **Author(s)**

[Lavanya](https://www.linkedin.com/in/lavanya-sunderarajan-199a445/?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMDeveloperSkillsNetworkCD0101ENSkillsNetwork20336975-2021-01-01)

## **Changelog**

| **Date** | **Version** | **Changed by** | **Change Description** |
| --- | --- | --- | --- |
| 29-Sep-2021 | 1.0 | Lavanya | Created the lab |
|  |  |  |  |

# Javascript cheatsheet

|  |  |  |  |
| --- | --- | --- | --- |
| **Item** | **Syntax** | **Description** | **Example** |
| Declaring Variables var, let, const | let < var\_name > = < value > | **var** - global access, value can chage **let** - access within block where it is declared, value can change **const** - access within block where it is declared, value cannot change | let i = 5; var myStr = "John"; const pi = 3.14 |
| **Strings** | | | |
| length | *string\_obj*.length | **length** Returns the length of the string | let myStr = "Hello"; console.log(myStr.length); Output is 5 |
| split | *string\_obj*.split(*separator*) | **split** Splits the string based on the separator and returns an array. | let myStr = "Hello! How are you?"; console.log(myStr.split()) Output is [ 'Hello!', 'How', 'are', 'you?' ] |
| charAt | *string\_obj*.charAt(*index*) | **charAt** returns the character at a specified index in a string. Index starts at 0 ends at length-1 | let myStr = "Hello";< console.log(myStr.charAt(0)) Output is H |
| replace | *string\_obj*.replace(*"SearchValue","NewValue"*) | **replace** searches a string for a specified value, or a regular expression, and returns a new string where the specified values are replaced. | let myStr = "Hello User"; console.log(myStr.replace("User","World")); Output is Hello World |
| substring | *string\_obj*.substring(start, end) | **substring** is used to extract characters, between to indices from the given string, and returns the substring. It excludes the last index | let myStr="Hello"; console.log(myStr.substing(1,4)); Output is ell |
| startswith | *string\_obj*.startsWith(searchvalue) | **startsWith** returns true if a string begins with a specified string, otherwise false | let myStr="Hello from the other side"; console.log(myStr.startsWith("Hello")); Output is *true* |
| endsWith | *string\_obj*.endsWith(searchvalue)) | **endsWith** returns true if a string ends with a specified string, otherwise false | let myStr="Hello from the other side"; console.log(myStr.startsWith("side")); Output is *true* |
| toUpperCase | *string\_obj.*toUpperCase() | **toUpperCase** converts a string to uppercase letters | let myStr="hello"; console.log(myStr.toUpperCase()); Output is HELLO |
| toLowerCase | *string\_obj.*toLowerCase() | **toLowerCase** converts a string to lowercase letters | let myStr="HELLO"; console.log(myStr.toUpperCase()); Output is hello |
| concat | *string\_obj.*concat(*string1, string2,..,stringN*) | **concat** joins two or more strings. | let myStr="Hello"; let str="World"; console.log(myStr.concat(str)); Output is HelloWorld |
| **Arrays** | | | |
| push | *arr\_name.*push(*value*) | **push** adds new items to the end of an array. | let myArr=["Hello"]; myArr.push("World"); console.log(myArr); Output is ["Hello","World"] |
| pop | *arr\_name.*pop() | **pop** removes the last element of an array. | let myArr=["Hello","World"]; myArr.pop(); console.log(myArr); Output is ["Hello"] |
| length | *arr\_name.*length | **length** sets or returns the number of elements in an array. | let myArr=["Hello","World"]; console.log(myArr.length); Output is 2 |
| indexOf | *arr\_name.*indexOf(*item*) | **indexOf** searches for a specified item and returns its position. | let myArr=["Hello","World"]; console.log(myArr.indexOf("World") Output is 1 |
| lastIndexOf | *arr\_name.*lastIndexOf(*item*) | **lastIndexOf** returns the last index (position) of a specified value. | let myArr=["Hello","World","Hello"]; console.log(myArr.lastIndexOf("Hello"); Output is 2 |
| entries | *arr\_name.*entries() | **entries** Returns and Array Iterator that helps you to iterate through the array and recieve each entry as an array of two elements containing the key and the value, where in the key is the index position of the element and value is the element itself. | const hello = ["h", "e", "l", "l","o"]; console.log(hello.entries()); Output is Object [Array Iterator] {} |  |
| find | Array.find(<arrElemet>=>{ //return boolean based on a condition } | **find** Finds the first occurance of an element in the array which returns true on checking the condition | //Find the first string with s let myarr = ["Mercury","Venus","Earth","Mars"]; let found = myarr.find(val=>{ return val.includes("s"); }) console.log(found); Output Venus |  |
| filter | Array.filter(<arrElemet>=>{ //return boolean based on a condition } | **filter** Finds the all occurances of elements in the array which returns true on checking the condition | //Find the all strings with s let myarr = ["Mercury","Venus","Earth","Mars"]; let found = myarr.filter(val=>{ return val.includes("s"); }) console.log(found); Output [Venus,Mars] |  |
| map | Array.map(<arrElemet>=>{ //return processed value } | **map** Processes the all elements of the array which returns a new processed array of same size | let myarr = ["name","place","thing","animal"]; let found = myarr.map(val=>{ return val+"s"; }) console.log(found); Output [ 'names', 'places', 'things', 'animals' ] |  |
| concat | *arr\_name.*.concat(arr1.name); | **concat** concatenates (joins) two or more arrays. | let hello = ["hello", "world" ]; let lorem = ["along","lorem"] let h = hello.concat(lorem); console.log(h);  Output is ["hello", "world", "along", "lorem"] |  |
| **Map** | | | |  |
| set | mapName.set(key,value); | **set** helps you define a new element with akey and its value | var newMap = new Map(); newMap.set("h", 1); console.log(newMap); Output is {"h" => 1} |  |
| get | mapName.get(key); | **get** helps you return a value of key you are searching for | var newMap = new Map(); newMap.get("h"); console.log(newMap); Output is {"h" => 1} |  |
| keys | mapName.keys(); | **get** is used to get all of the keys associated with the mapName | var newMap = new Map(); newMap.set("h",1); newMap.set("i",2); console.log(newMap.keys()); Output is {"h", "i"} |  |
| values | mapName.values(); | **values** is used to get all of the values to the keys associated with the mapName | var newMap = new Map(); newMap.set("h",1); newMap.set("i",2); console.log(newMap.values()); Output is {1,2} |  |
| has | mapName.has(key\_name); | **has** is used to check if the key passed resides in the map or not, and returns true or false | var newMap = new Map(); newMap.set("h",1); newMap.set("i",2); console.log(newMap.has(i)); Output is true |  |
| delete | mapName.delete(key\_name); | **delete** is used to delete the key and the value from the map | var newMap = new Map(); newMap.set("h",1); newMap.set("i",2); newMap.delete("h"); console.log(newMap); Output is {"i" => 2} |  |
| **JSON** | | | |  |
| Create JSON | let varname={name1:value1,name2:values2,.....} | JSON is a dictionary Object with Key-Value pairs. | let myjson1={}; let myjson2 = {"name":"Jennifer","age":"32"} |  |
| Add entry to JSON | let jsonObj[<key>]=<value> | Adds an entry to JSON Object mapping the key to value | let myjson1 = {}; myjson1["name"]="Jason"; console.log(myjson1); |  |
| **Operators** | | | |  |
| Arithmetic | <Operand1> <Operator> <Operand2> | **+** addition **-** subtration **/** division **\*** multiplication **%** modulus(gives remainder) **++** increment by 1 **--** decrement by 1 | let num1 = 2; let num2 = 2; console.log(num1+num2); console.log(num1-num2); console.log(num1/num2); console.log(num1\*num2); console.log(num1%num2); num1++; console.log(num1); num2--; console.log(num1); Output is 4 0 1 4 0 3 3 |  |
| Logical | condition1 && condition2 condition1 || condition2 ! condition1 | **&&** (AND)is used to check if all the operand conditions are true **||** (OR)is used to check if either of the operand condition are true **!** (NOT) is used to check if the operand condition is not met | let num1 = 12, num2 = 2; console.log(num1>10 && num2>10); console.log(num1>10 || num2>10); console.log(!(num1==num2)); Output is false true true |  |
| Assignment | variable = value variable += incremental value variable -= decremental value %= modulus value /= divide value \*= multiply value | **a=b**assigns the value of b to a **a+=b** adds the value of b to a and stores it in a **a-=b** subtracts the value of b from a and stores it in a **a%=b** divides the value of a by b and stores the remainder in a **a/=b** divides the value of a to b and stores the quotient in a **a\*=b** multiplies the value of a and b and stores the value in a | let num1 = 12, num2 = 2; console.log(num1=num2); console.log(num1+=num2); console.log(num1-=num2); console.log(num1/=num2); console.log(num1\*=num2); console.log(num1%num2); console.log(num1=num2); Output is 2 14 10 6 24 0 2 |  |
| **Loops** | | | |  |
| For Loop | for(initialization;condition;increment/decrement){ //code block } | **for** loops throughout the block of code a number of times making sure the condition is satisfied | for(let num = 0 ; num <=5 ; num++){ console.log(num) } Output is 0 1 2 3 4 5 |  |
| while | while(condition){ //code block } | **while** itrates through the block of code while a specified condition is true | let num1 = 0; let num2 = 5; while(num1 < num2){ console.log(num1) num1++; }  Output is 0 1 2 3 4 |  |
| do while | do{ //code block } while(condition) | **do while** loops throughout the block once before checking condition. | let num = 5; do { console.log(num); num--; }  while(num > 0)  Output is 5 4 3 2 1 |  |
| for in | for (var in object) { //code block } | **for in** is used to itrate through the specific property/type of the object | let arr = ["a","b","c"]; for(let i in arr) { console.log(arr[i]); }  Output is a b c |  |
| **Conditional statements** | | | |  |
| if | if(*condition*){ //code Block... } | **if** a specified condition is true, a block of code will be executed | let num = 5; if(num = 5){ console.log(true); } Output is true |  |
| if-else | if(*condition*){ //Code Block... } else { //Code Block... } | **if** a specified condition is true, a block of code will be executed. in case of false, else block is executed | let num = 5; if(num = 4){ console.log(true) } else { console.log(false) } Output is false |  |
| if-else if-else | if(*condition*){ //Code Block... } else if (*condition*) { //Code Block... } else { //Code Block... } | **else if** to specify a new condition to test, if the first/previous condition is false | let num = 10; if(num < 10){ console.log("number is smaller"); } else if(num = 10) { console.log("number is equal"); } else { console.log("number is greater"); } Output is number is equal |  |
| switch | switch(expression) { case <value1>: //code break; case <value2>: //code break; . . . default: //default code block } | **switch** to select one of many blocks of code to be executed. And **break** is used to end the preocessing within the switch statement. | let num = 2; switch(num) { case 1: console.log("Hello world!"); break; case 2: console.log("Hi"); break; default: console.log("this is default"); } Output is Hi |  |
| **Other useful operations** | | | |  |
| typeof | typeof(operand) | **typeof** operator returns a string indicating the type of the unevaluated operand | console.log(typeOf("Hello")) Output is "string" |  |
| isNaN | isNaN(operand) | **isNaN** determines whether a value is anythying but a number or not. It returns false for a number | console.log(isNaN("Hello")) Output is true |  |
| parseInt | parseInt(string, radix) | **parseInt** is a function that parses a string argument and returns an integer of the specified radix.(radix is a base) | //0011 is 3 for binary, since binary only has 2 numbers 0, 1 the radix is 2 console.log(parseInt("0011", 2)); //Default parseInt takes decimal system console.log(parseInt("54")); Output is 3 54 |  |
| parseFloat | parseFloat(string) | **parseFloat** is a function that parses a string argument and returns an float | parseFloat("3.14") Output is 3.14 |  |

This cheatsheet covers the JS you will mostly use. To learn more commands you can go to this [link](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMDeveloperSkillsNetworkCD0101ENSkillsNetwork20336975-2021-01-01).