Introduction to JavaScript

Module Overview

This module explains about the aspect of web development and its use for full stack development

Module Objective

At the end of this module, students should be able to demonstrate appropriate knowledge, and show an understanding of the following:

- Web Technology
- HTTP Protocol
- URN
- Web Services
- Web applications
- Full Stack development
- Use of Python, Django Ruby on Rails in full stack development

Introduction to JavaScript

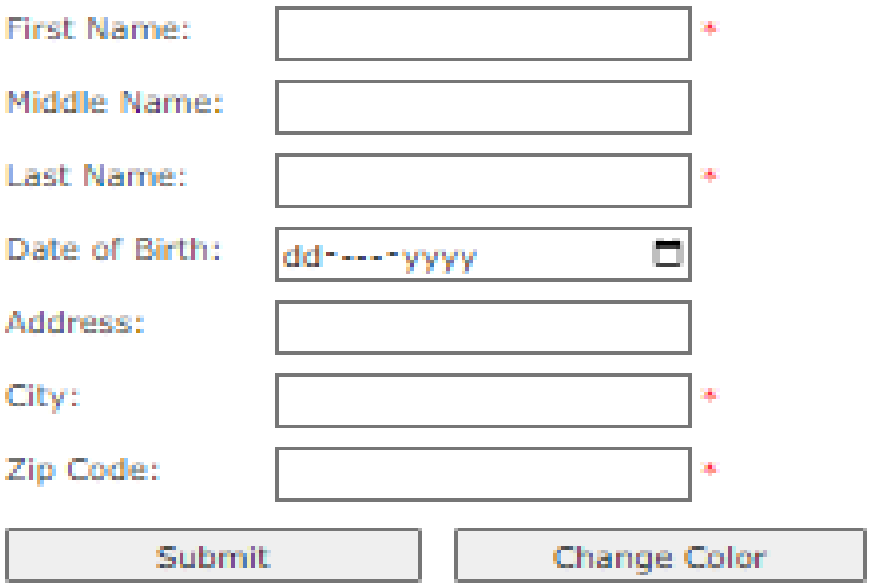
What is JavaScript

JavaScript is a programming language that executes on the browser. It turns static HTML web pages into interactive web pages by dynamically updating content, validating form data, controlling multimedia, animate images, and almost everything else on the web pages.

JavaScript is the third most important web technology after HTML and CSS. JavaScript can be used to create web and mobile applications, build web servers, create games, etc.

JavaScript Example

JavaScript can be used in various activities like data validation, displaying popup messages, handling events of HTML elements, modifying CSS, etc. The following sample form uses JavaScript to validate data and change the color of the form.



The responsive UI and menu of this website is also using JavaScript. There is no website in this world that does not use JavaScript or JavaScript-based frameworks.

JavaScript History

In early 1995, Brendan Eich from Netscape designed and implemented a new language for non-java programmers to give newly added Java support in Netscape navigator. It was initially named Mocha, then LiveScript, and finally JavaScript.

Nowadays, JavaScript can execute not only on browsers but also on the server or any device with a JavaScript Engine. For example, Node.js is a framework based on JavaScript that executes on the server.

JavaScript and ECMAScript

Often you will hear the term ECMAScript while working with JavaScript. Let's clear the confusion before it arises.

As you now know, JavaScript was primarily developed to execute on browsers. There are many different browsers from different companies. So, there was a need to standardize the execution of the JavaScript code to achieve the same functionality in all the browsers.

Ecma International is a non-profit organization that creates standards for technologies. ECMA International publishes the specification for scripting languages is called 'ECMAScript'. ECMAScript specification defined in ECMA-262 for creating a general-purpose scripting language.

JavaScript implements the ECMAScript standards, which includes features specified in ECMA-262 specification as well as other features which are not based on ECMAScript standards.

There are different editions of ECMAScript. Most browsers have implemented ECMA-262 5.1 edition.

- ECMA-262 5.1 edition, June 2011
- ECMA-262, 6th edition, June 2015
- ECMA-262, 7th edition, June 2016
- ECMA-262, 8th edition, June 2017
- ECMA-262, 9th edition, June 2018
- ECMA-262, 10th edition, June 2019

JavaScript Engine

JavaScript engine interprets, compiles, and executes JavaScript code. It also does memory management, JIT compilation, type system, etc. Different browsers use different JavaScript engines, as listed in the below table.

Browser	JavaScript Engine
Edge	Chromium with Blink and V8 engines
Chrome	V8
FireFox	Spider Monkey
Safari	Nitro

Setup JavaScript Development Environment

Here, we are going to use JavaScript in developing a web application. So, we must have at least two things, a browser, and an editor to write the JavaScript code.

Although we also need a webserver to run a web application, but we will use a single HTML web page to run our JavaScript code. So, no need to install it for now.

Browser

Mostly, you will have a browser already installed on your PC, Microsoft Edge on the Windows platform, and Safari on Mac OS.

You can also install the following browser as per your preference:

- Microsoft Edge
- Google Chrome
- Mozilla FireFox
- Safari
- Opera

IDEs for JavaScript Application Development

You can write JavaScript code using a simple editor like Notepad. However, you can install any open-sourced or licensed IDE (Integrated Development Environment) to get the advantage of IntelliSense support for JavaScript and syntax error/warning highlighter for rapid development.

The followings are some of the well-known JavaScript editors:

- Visual Studio Code (Free, cross-platform)
- Eclipse (Free, cross-platform)
- Atom (Free, cross-platform)
- Notepad++ (Free, Windows)
- Code Lobster (Free, cross-platform)
- WebStorm (Paid, cross-platform)

Online Editors for JavaScript

Use the online editor to quickly execute the JavaScript code without any installation. The followings are free online editors:

- jsfiddle.net
- jsbin.com
- playcode.io

Learn where to write JavaScript code in a web page in the next section.

HTML <script> Tag

The HTML script tag <script> is used to embed data or executable client side scripting language in an HTML page. Mostly, JavaScript or JavaScript based API code inside a <script></script> tag.

The following is an example of an HTML page that contains the JavaScript code in a <script> tag.

Example: JavaScript in a <script> Tag

```
<!DOCTYPEhtml>
<html>
<head>
```

```
</head>
<body>
<h1> JavaScript Tutorials</h1>
<script>
//write JavaScript code here..
alert('Hello, how are you?')
</script>
</body>
</html>
```

In the above example, a <script></script> tag contains the JavaScript alert('Hello, how are you?') that display a message box.
HTML v4 requires the type attribute to identify the language of script code embedded within script tag. This is specified as MIME type e.g. 'text/javascript', 'text/ecmascript', 'text/vbscript', etc.
HTML v5 page does not require the type attribute because the default script language is 'text/javascript' in a <script> tag.
An HTML page can contain multiple <script> tags in the <head> or <body> tag. The browser executes all the script tags, starting from the first script tag from the beginning.
Scripts without async, defer or type="module" attributes, as well as inline scripts, are fetched and executed immediately, before the browser continues to parse the page. Consider the following page with multiple script tags.

Example: Multiple <script> Tags

```
<!DOCTYPEhtml>
<html>
<head>
<script>
alert('Executing JavaScript 1')
</script>
</head>
<body>
<h1> JavaScript Tutorials</h1>
<script>
alert('Executing JavaScript 2')
</script>
<p>This page contains multiple script tags.</p>
<script>
alert('Executing JavaScript 3')
```

```
</script>
</body>
</html>
```

Above, the first <script> tag containing alert('Executing JavaScript 1') will be executed first, then alert('Executing JavaScript 2') will be executed, and then alert('Executing JavaScript 3') will be executed.

The browser loads all the scripts included in the <head> tag before loading and rendering the <body> tag elements. So, always include JavaScript files/code in the <head> that are going to be used while rendering the UI. All other scripts should be placed before the ending </body> tag. This way, you can increase the page loading speed.

Reference the External Script File

A <script> tag can also be used to include an external script file to an HTML web page by using the src attribute. If you don't want to write inline JavaScript code in the <script></script> tag, then you can also write JavaScript code in a separate file with .js extension and include it in a web page using <script> tag and reference the file via src attribute.

Example: JavaScript in a <script> Tag

```
<!DOCTYPEhtml>
<html>
<head>
<scriptsrc="/MyJavaScriptFile.js"></script>
</head>
<body>
<h1> JavaScript Tutorials</h1>
</body>
</html>
```

Above, the <script src="/MyJavaScriptFile.js"> points to the external JavaScript file using the src="/MyJavaScriptFile.js" attribute where the value of the src attribute is the path or url from which a file needs to be loaded in the browser. Note that you can load the files from your domain as well as other domains.

Global Attributes

The <script> can contain the following global attributes:

async	<script async> executes the script asynchronously along with the rest of the page.

crossorign	<script crossorigin="anonymous|use-credentials"> allows error logging for sites which use a separate domain for static media. Value anonymous do not send credentials, whereas use-credentials sends the credentials.
defer	<script defer> executes the script after the document is parsed and before firing DOMContentLoaded event.
src	<script src="uri\path to resource"> specifies the URI/path of an external script file;
type	<script type="text\javascript"> specifies the type of the containing script e.g. text\javascript, text\html, text\plain, application\json, application\pdf, etc.
referrerpolicy	<script referrerpolicy="no-referrer"> specifies which referrer information to send when fetching a script. Values can be no-referrer, no-referrer-when-downgrade, origin, same-origin, strict-origin, etc.
integrity	<script integrity="sha384-oqVuAfXRKap7fdgc"> specifies that a user agent can use to verify that a fetched resource has been delivered free of unexpected manipulation.
nomodule	<script nomodule> specifies that the script should not be executed in browsers supporting ES2015 modules.

JavaScript Syntax and writing code

JavaScript Syntax

Learn some important characteristics of JavaScript syntax in this section.

As mentioned in the previous chapter, JavaScript code can be written inside HTML Script Tags or in a separate file with .js extension.

Write JavaScript Code

```
<script>
//Write javascript code here...
</script>
```

Character Set

JavaScript uses the unicode character set, so allows almost all characters, punctuations, and symbols.

Case Sensitive

JavaScript is a case-sensitive scripting language. So, name of functions, variables and keywords are case sensitive. For example, myfunction and MyFunction are different, Name is not equal to nAme, etc.

Variables

In JavaScript, a variable is declared with or without the var keyword.

Example: JavaScript Statements

```
<script>
var name = "Steve";
   id = 10;
</script>
```

Semicolon

JavaScript statements are separated by a semicolon. However, it is not mandatory to end a statement with a semicolon, but it is recommended.

Example: JavaScript Statements

```
<script>
var one = 1; two = 2; three = 3; //three different statements
var four = 4; //single statement
var five = "Five" //single statement without ;
</script>
```

Whitespaces

JavaScript ignores multiple spaces and tabs. The following statements are the same.

Example: Whitespaces in JavaScript

```
<script>
var one =1;
var one   =   1;
var one       =       1;
</script>
```

Code Comments

A comment is single or multiple lines, which give some information about the current program. Comments are not for execution.

Write comment after double slashes // or write multiple lines of comments between /* and */

Example: Comment JavaScript Code

```
<script>
var one =1; // this is a single line comment
/* this
is multi line
comment*/
var two = 2;
var three = 3;
</script>
```

String

A string is a text in JavaScript. The text content must be enclosed in double or single quotation marks.

Example: String in JavaScript

```
<script>
varmsg = "Hello World"//JavaScript string in double quotes
varmsg = 'Hello World'//JavaScript string in single quotes
</script>
```

Number

JavaScript allows you to work with any type of number like integer, float, hexadecimal etc. Number must **NOT** be wrapped in quotation marks.

Example: Numbers in JavaScript

```
<script>
varnum = 100;
varflot = 10.5;
</script>
```

Boolean

As in other languages, JavaScript also includes true and false as a boolean value.

Example: Booleans in JavaScript

```
<script>
var yes = true;
var no = false;
</script>
```

Keywords

Keywords are reserved words in JavaScript, which cannot be used as variable names or function names. The following table lists some of the keywords used in JavaScript.

var	function	if
else	do	while
for	switch	break
continue	return	try
catch	finally	debugger
case	class	this
default	false	true
in	instanceOf	typeOf
new	null	throw
void	width	delete

JavaScript variables & operators

JavaScript Variables

Variable means anything that can vary. In JavaScript, a variable stores the data value that can be changed later on.

Use the reserved keyword var to declare a variable in JavaScript.

var <*variable-name*>;
var <*variable-name*> = <*value*>;

A variable must have a unique name. The following declares a variable.

Example: Variable Declaration

```
varmsg; // declaring a variable without assigning a value
```

Above, the var msg; is a variable declaration. It does not have any value yet. The default value of variables that do not have any value is undefined. You can assign a value to a variable using the = operator when you declare it or after the declaration and before accessing it.

Example: Variable Initialization

```
varmsg;
msg = "Hello JavaScript!"; // assigned a string value
alert(msg); // access a variable
//the following declares and assign a numeric value
varnum = 100;
var hundred = num;  // assigned a variable to varible
```

In the above example, the msg variable is declared first and then assigned a string value in the next statement. The num variable is declared and initialized with a numeric value in the same statement. Finally, the hundred variable is declared and initialized with another variable's value.

Multiple Variables Declaration

Multiple variables can also be declared in a single line separated by a comma.

Example: Multiple Variables in a Single Line

```
var one = 1, two = 'two', three;
```

White Spaces and Line Breaks in Variable Declaration

JavaScript allows multiple white spaces and line breaks when you declare a variable with **var** keyword.

Example: Whitespace and Line Breaks

```
var
```

```
    one
=
1,
  two
  =
  "two"
```

Please note that thesemicolon ; at the end is optional.

Javascript Operators

JavaScript includes operators same as other languages. An operator performs some operation on single or multiple operands (data value) and produces a result. For example, in 1 + 2, the + sign is an operator and 1 is left side operand and 2 is right side operand. The + operator performs the addition of two numeric values and returns a result.

<*Left operand*> operator <*right operand*>

<*Left operand*> operator

JavaScript includes following categories of operators.

- Arithmetic Operators
- Comparison Operators
- Logical Operators
- Assignment Operators
- Conditional Operators
- Ternary Operator

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations between numeric operands.

+	Adds two numeric operands.
-	Subtract right operand from left operand
*	Multiply two numeric operands.
/	Divide left operand by right operand.
%	Modulus operator. Returns remainder of two operands.
++	Increment operator. Increase operand value by one.
--	Decrement operator. Decrease value by one.

The following example demonstrates how arithmetic operators perform different tasks on operands.

Example: Arithmetic Operation

```
var x = 5, y = 10;
var z = x + y; //performs addition and returns 15
z = y - x; //performs subtraction and returns 5
z = x * y; //performs multiplication and returns 50
z = y / x; //performs division and returns 2
z = x % 2; //returns division remainder 1
```

The ++ and -- operators are unary operators. It works with either left or right operand only. When used with the left operand, e.g., x++, it will increase the value of x when the program control goes to the next statement. In the same way, when it is used with the right operand, e.g., ++x, it will increase the value of x there only. Therefore, x++ is called post-increment, and ++x is called pre-increment.

Example: Post and Pre Increment/Decrement

```
var x = 5;

x++; //post-increment, x will be 5 here and 6 in the next line
++x; //pre-increment, x will be 7 here
x--; //post-decrement, x will be 7 here and 6 in the next line
--x; //pre-decrement, x will be 5 here
```

String Concatenation

The + operator performs concatenation operation when one of the operands is of string type. The following example demonstrates string concatenation even if one of the operands is a string.

Example: + Operator with String

```
var a = 5, b = "Hello ", c = "World!", d = 10;
a + b; //returns "5Hello "
b + c; //returns "Hello World!"
a + d; //returns 15
b + true; //returns "Hello true"
c - b; //returns NaN; - operator can only used with numbers
```

Comparison Operators

JavaScript provides comparison operators that compare two operands and return a boolean value true or false.

==	Compares the equality of two operands without considering type.

===	Compares equality of two operands with type.
!=	Compares inequality of two operands.
>	Returns a boolean value true if the left-side value is greater than the right-side value; otherwise, returns false.
<	Returns a boolean value true if the left-side value is less than the right-side value; otherwise, returns false.
>=	Returns a boolean value true if the left-side value is greater than or equal to the right-side value; otherwise, returns false.
<=	Returns a boolean value true if the left-side value is less than or equal to the right-side value; otherwise, returns false.

The following example demonstrates the comparison operators.

Example: JavaScript Comparison Operators

```
var a = 5, b = 10, c = "5";
var x = a;
a == c; // returns true
a === c; // returns false
a == x; // returns true
a != b; // returns true
a > b; // returns false
a < b; // returns true
a >= b; // returns false
a <= b; // returns true
```

Logical Operators

In JavaScript, the logical operators are used to combine two or more conditions. JavaScript provides the following logical operators.

&&	&& is known as AND operator. It checks whether two operands are non-zero or not (0, false, undefined, null or "" are considered as zero). It returns 1 if they are non-zero; otherwise, returns 0.
||	|| is known as OR operator. It checks whether any one of the two operands is non-zero or not (0, false, undefined, null or "" is considered as zero). It returns 1 if any one of of them is non-zero; otherwise, returns 0.
!	! is known as NOT operator. It reverses the boolean result of the operand (or condition). !false returns true, and !true returns false.

Example: Logical Operators

```
var a = 5, b = 10;
(a != b) && (a < b); // returns true
(a > b) || (a == b); // returns false
(a < b) || (a == b); // returns true
!(a < b); // returns false
!(a > b); // returns true
```

Assignment Operators

JavaScript provides the assignment operators to assign values to variables with less key strokes.

=	Assigns right operand value to the left operand.
+=	Sums up left and right operand values and assigns the result to the left operand.
-=	Subtract right operand value from the left operand value and assigns the result to the left operand.
*=	Multiply left and right operand values and assigns the result to the left operand.
/=	Divide left operand value by right operand value and assign the result to the left operand.
%=	Get the modulus of left operand divide by right operand and assign resulted modulus to the left operand.

Example: Assignment operators

```
var x = 5, y = 10, z = 15;
x = y; //x would be 10
x += 1; //x would be 6
x -= 1; //x would be 4
x *= 5; //x would be 25
x /= 5; //x would be 1
x %= 2; //x would be 1
```

Ternary Operator

JavaScript provides a special operator called ternary operator:? that assigns a value to a variable based on some condition. This is the short form of the if else condition.

<condition> ?<value1> :<value2>;

The ternary operator starts with conditional expression followed by the? Operator. The second part (after ? and before :) will be executed if the condition turns out to be true. Suppose, the condition returns false, then the third part (after :) will be executed.

Example: Ternary operator

```
var a = 10, b = 5;
var c = a > b? a : b; // value of c would be 10
var d = a > b? b : a; // value of d would be 5
```

Points to Remember:

- JavaScript includes operators that perform some operation on single or multiple operands (data value) and produce a result.
- JavaScript includes various categories of operators: Arithmetic operators, Comparison operators, Logical operators, Assignment operators, Conditional operators.
- Ternary operator?: is a short form of if-else condition.

JavaScript Data Types

JavaScript Data Types

JavaScript includes data types similar to other programming languages like Java or C#. JavaScript is dynamic and loosely typed language. It means you don't require specifying a type of a variable. A variable in JavaScript can be assigned any type of value, as shown in the following example.

Example: Loosely Typed Variables

```
var myvariable = 1;  // numeric value
myvariable = 'one'; // string value
myvariable = 1.1; // decimal value
myvariable = true; // Boolean value
myvariable = null; // null value
```

In the above example, different types of values are assigned to the same variable to demonstrate loosely typed characteristics of JavaScript. Different values 1, 'one', 1.1, true are examples of different data types. JavaScript includes primitive and non-primitive data types as per latest ECMAScript 5.1.

Primitive Data Types

The primitive data types are the lowest level of the data value in JavaScript. The typeof operator can be used with primitive data types to know the type of a value. The followings are primitive data types in JavaScript:

Data Type	Description

String	String is a textual content wrapped inside ' ' or " " or ` ` (tick sign). Example: 'Hello World!', "This is a string", etc.
Number	Number is a numeric value. Example: 100, 4521983, etc.
BigInt	BigInt is a numeric value in the arbitrary precision format. Example: 453889879865131n, 200n, etc.
Boolean	is a logical data type that has only two values, true or false
Null	A null value denotes an absense of value Example: var str = null;
Undefined	undefined is the default value of a variable that has not been assigned any value. Example: In the variable declaration, var str;, there is no value assigned to str. So, the type of str can be check using typeof(str) which will return undefined.

Structural Data Types

The structural data types contain some kind of structure with primitive data.

Data Type	**Description**

Object	An object holds multiple values in terms of properties and methods.
	Example:
	var person = {
	firstName: "James",
	lastName: "Bond",
	age: 15
	};

Date	Date object represents date & time including days, months, years, hours, minutes, seconds and milliseconds. Example: var today = new Date("25 July 2021");
Array	An array stores multiple values using special syntax. Example: var nums = [1, 2, 3, 4];

JavaScript Array

JavaScript Array

We have learned that a variable can hold only one value, for example var i = 1, we can assign only one literal value to i. We cannot assign multiple literal values to a variable i. To overcome this problem, JavaScript

provides an array.

An array is a special type of variable, which can store multiple values using special syntax. Every value is associated with numeric index starting with 0. The following figure illustrates how an array stores values.

Value	10	“hello”	“World”	20.2	A10	A20
Index	0	1	2	3	4	5

Array Initialization

An array in JavaScript can be defined and initialized in two ways, array literal and Array constructor syntax.

Array Literal

Array literal syntax is simple. It takes a list of values separated by a comma and enclosed in square brackets.

Syntax:

var <array-name> = [element0, element1, element2,... elementN];

Example: Array Literal

The following example shows how to define and initialize an array using array literal syntax.

```
<!DOCTYPE html>
<html>
<body>
        <h1>Demo: JavaScript Array</h1>
        <p id="p1"></p>
        <p id="p2"></p>
        <p id="p3"></p>
        <p id="p4"></p>
        <p id="p5"></p>
        <script>
                var stringArray = ["one", "two", "three"];
                var numericArray = [1, 2, 3, 4];
                var decimalArray = [1.1, 1.2, 1.3];
                var booleanArray = [true, false, false, true];
                var mixedArray = [1, "two", "three", 4];
                document.getElementById("p1").innerHTML = stringArray;
                document.getElementById("p2").innerHTML = numericArray;
                document.getElementById("p3").innerHTML = decimalArray;
                document.getElementById("p4").innerHTML = booleanArray;
                document.getElementById("p5").innerHTML = mixedArray;
  </script>
```

```
</body>
</html>
```

JavaScript array can store multiple elements of different data types. It is not required to store value of same data type in an array.

Array Constructor

Initialize an array with Array constructor syntax using new keyword.

The Array constructor has following three forms.

Syntax:

```
var arrayName = new Array();
var arrayName = new Array(Number length);
var arrayName = new Array(element1, element2, element3,... elementN);
```

As you can see in the blow syntax, an array can be initialized using new keyword, in the same way as an object.

Example: Array Constructor

```
<!DOCTYPE html>
<html>
<body>
        <h1>Demo: JavaScript Array Object</h1>
        <p id="p1"></p>
        <p id="p2"></p>
        <p id="p3"></p>
        <script>
                var stringArray = new Array();
                stringArray[0] = "one";
                stringArray[1] = "two";
                stringArray[2] = "three";
                stringArray[3] = "four";
                var numericArray = new Array(3);
                numericArray[0] = 1;
                numericArray[1] = 2;
                numericArray[2] = 3;
                var mixedArray = new Array(1, "two", 3, "four");
                document.getElementById("p1").innerHTML = stringArray;
                document.getElementById("p2").innerHTML = numericArray;
                document.getElementById("p3").innerHTML = mixedArray;
   </script>
</body>
```

</html>

Array Constructor- Incorrect Initialization

Please note that array can only have numeric index (key). Index cannot be of string or any other data type. The following syntax is incorrect.

```
var stringArray = new Array();
stringArray["one"] = "one";
stringArray["two"] = "two";
stringArray["three"] = "three";
stringArray["four"] = "four";
```

Array Methods Reference

The following table lists all the Array methods.

Method	Description
concat()	Returns new array by combining values of an array that is specified as parameter with existing array values.
every()	Returns true or false if every element in the specified array satisfies a condition specified in the callback function. Returns false even if single element does not satisfy the condition.
filter()	Returns a new array with all the elements that satisfy a condition specified in the callback function.
forEach()	Executes a callback function for each elements of an array.
indexOf()	Returns the index of the first occurrence of the specified element in the array, or -1 if it is not found.
join()	Returns string of all the elements separated by the specified separator
lastIndexOf()	Returns the index of the last occurrence of the specified element in the array, or -1 if it is not found.
map()	Creates a new array with the results of calling a provided function on every element in this array.
pop()	Removes the last element from an array and returns that element.
push()	Adds one or more elements at the end of an array and returns the new length of the array.
reduce()	Pass two elements simultaneously in the callback function (till it reaches the last element) and returns a single value.
reduceRight()	Pass two elements simultaneously in the callback function from right-to-left (till it reaches the last element) and returns a single value.
reverse()	Reverses the elements of an array. Element at last index will be first and element at 0 index will be last.
shift()	Removes the first element from an array and returns that element.

slice()	Returns a new array with specified start to end elements.
some()	Returns true if at least one element in this array satisfies the condition in the callback function.
sort()	Sorts the elements of an array.
splice()	Adds and/or removes elements from an array.
toString()	Returns a string representing the array and its elements.
unshift()	Adds one or more elements to the front of an array and returns the new length of the array.

JavaScript Objects

JavaScript Objects

Here you will learn objects, object literals, Object() constructor function, and access object in JavaScript.

You learned about primitive and structured data types in JavaScript. An object is a non-primitive, structured data type in JavaScript. Objects are same as variables in JavaScript, the only difference is that an object holds multiple values in terms of properties and methods. In JavaScript, an object can be created in two ways:

1) Using Object Literal/Initializer Syntax

2) Using the Object() Constructor function with the new keyword. Objects created using any of these methods are the same. The following example demonstrates creating objects using both ways.

Example: JavaScript Objects

```
var p1 = { name:"Steve" }; // object literal syntax
var p2 = new Object(); // Object() constructor function
p2.name = "Steve"; // property
```

Above, p1 and p2 are the names of objects. Objects can be declared same as variables using var or let keywords. The p1 object is created using the object literal syntax (a short form of creating objects) with a property named name. The p2 object is created by calling the Object() constructor function with the new keyword. The p2.name = "Steve"; attach a property name to p2 object with a string value "Steve".

Create Object using Object Literal Syntax

The object literal is a short form of creating an object. Define an object in the { } brackets with key:value pairs separated by a comma. The key would be the name of the property and the value will be a literal value or a function.

Syntax:

```
var <object-name> = { key1: value1, key2: value2,...};
```

The following example demonstrates objects created using object literal syntax.

```
var emptyObject = {}; // object with no properties or methods
```

```
var person = { firstName: "John" }; // object with single property
// object with single method
var message = {
        showMessage: function (val) {
                alert(val);
        }
    };
// object with properties & method
var person = {
        firstName: "Edu",
        lastName: "Bridge",
        age: 15,
        getFullName: function () {
            return this.firstName + ' ' + this.lastName
        }
    };
```

The whole key-value pair must be declared. Declaring only a key without a value is invalid

Create Objects using Objects() Constructor

Another way of creating objects is using the Object() constructor function using the new keyword. Properties and methods can be declared using the dot notation .property-name or using the square brackets ["property-name"], as shown below.

```
<!DOCTYPE html>
<html>
<body>
    <h1>Demo: JavaScript Object</h1>
    <p id="p1"></p>
    <p id="p2"></p>
    <p id="p3"></p>
    <p id="p4"></p>
    <p id="p5"></p>
    <script>
        var person = new Object();
        // Attach properties and methods to person object
        person.firstName = "Edu";
        person["lastName"] = "Bridge";
```

```
            person.age = 25;
            person.getFullName = function () {
                        return this.firstName + ' ' + this.lastName;
                    };
            // access properties & methods
            document.getElementById("p1").innerHTML = person.firstName;
            document.getElementById("p2").innerHTML = person.lastName;
            document.getElementById("p3").innerHTML = person["firstName"];
            document.getElementById("p4").innerHTML = person["lastName"];
            document.getElementById("p5").innerHTML = person.getFullName();
  </script>
</body>
</html>
```

JavaScript Object Properties & Methods

An object's properties can be accessed using the dot notation obj.property-name or the square brackets obj["property-name"]. However, method can be invoked only using the dot notation with the parenthesis, obj.method-name(), as shown below.

```
<!DOCTYPE html>
<html>
<body>
        <h1>Demo: JavaScript Object</h1>
        <p id="p1"></p>
        <p id="p2"></p>
        <p id="p3"></p>
        <p id="p4"></p>
        <p id="p5"></p>
        <script>
                var person = {
          firstName: "Edu",
          lastName: "Bridge",
          age: 25,
          getFullName: function () {
            return this.firstName + ' ' + this.lastName
            }
        };
        document.getElementById("p1").innerHTML = person.firstName;
```

```
        document.getElementById("p2").innerHTML = person.lastName;
        document.getElementById("p3").innerHTML = person["firstName"];
        document.getElementById("p4").innerHTML = person["lastName"];
        document.getElementById("p5").innerHTML = person.getFullName();
    </script>
</body>
</html>
```

In the above example, the person.firstName access the firstName property of a person object. The person ["firstName"] is another way of accessing a property. An object's methods can be called using () operator e.g. person.getFullName(). JavaScript engine will return the function definition if accessed method without the parenthesis.

JavaScript Functions

JavaScript Functions

JavaScript provides functions similar to most of the scripting and programming languages.

In JavaScript, a function allows you to define a block of code, give it a name and then execute it as many times as you want.

A JavaScript function can be defined using function keyword.

Syntax:

```
//defining a function
function <function-name>()
{
   // code to be executed
};

//calling a function
<function-name>();
```

The following example shows how to define and call a function in JavaScript.

```
<!DOCTYPE html>
<html>
<body>
        <h1>Demo: JavaScript function</h1>
```

```
        <script>
                function ShowMessage() {
                        alert("Hello World!");
                }

                ShowMessage();
    </script>
</body>
</html>
```

In the above example, we have defined a function named ShowMessage that displays a popup message "Hello World!". This function can be execute using () operator e.g. ShowMessage().

Function Parameter

A function can have one or more parameters, which will be supplied by the calling code and can be used inside a function. JavaScript is a dynamic type scripting language, so a function parameter can have value of any data type.

```
<!DOCTYPE html>
<html>
<body>
        <h1>Demo: JavaScript function parameters</h1>

        <script>
                function ShowMessage(firstName, lastName) {
                        alert("Hello " + firstName + " " + lastName);
                }

                ShowMessage("Gaurav", "hajela");
                ShowMessage("Ashwaini", "Yadav");
                ShowMessage(100, 200);
    </script>
</body>
</html>
```

Function Arguments Object

All the functions in JavaScript can use arguments object by default. An arguments object includes value of each parameter.

The arguments object is an array like object. You can access its values using index similar to array. However, it does not support array methods.

```
<!DOCTYPE html>
<html>
<body>
	<h1>Demo: JavaScript arguments object</h1>
		<script>
		function ShowMessage(firstName, lastName) {
			alert("Hello " + arguments[0] + " " + arguments[1]);
		}
		ShowMessage("Steve", "Jobs");
		ShowMessage("Bill", "Gates");
		ShowMessage(100, 200);
  </script>
</body>
</html>
```

An arguments object is still valid even if function does not include any parameters.

```
<!DOCTYPE html>
<html>
<body>
	<h1>Demo: JavaScript arguments object</h1>

	<script>
		function ShowMessage() {
			alert("Hello " + arguments[0] + " " + arguments[1]);
		}

		ShowMessage("Steve", "Jobs");
		ShowMessage("Bill", "Gates");
		ShowMessage(100, 200);
  </script>
</body>
</html>
```

Return Value

A function can return zero or one value using return keyword.

```
<!DOCTYPE html>
<html>
<body>
        <h1>Demo: JavaScript arguments object</h1>
        <p id="p1"></p>
        <p id="p2"></p>
        <script>
                function Sum(val1, val2) {
                        return val1 + val2;
                };
                document.getElementById("p1").innerHTML = Sum(10,20);
                function Multiply(val1, val2) {
                        console.log( val1 * val2);
                };
                document.getElementById("p2").innerHTML =  Multiply(10,20);
   </script>
</body>
</html>
```

In the above example, a function named Sum adds val1 & val2 and return it. So the calling code can get the return value and assign it to a variable. The second function Multiply does not return any value, so result variable will be undefined.

A function can return another function in JavaScript.

```
<!DOCTYPE html>
<html>
<body>
        <h1>Demo: Function returing a function</h1>
        <p id="p1"></p>
        <p id="p2"></p>

        <script>
                function multiple(x) {

                        function fn(y)
                        {
                                return x * y;
```

```
            }

            return fn;
        }

        var triple = multiple(3);

        document.getElementById("p1").innerHTML = triple(2);
        document.getElementById("p2").innerHTML = triple(3);
  </script>
</body>
</html>
```

Function Expression

JavaScript allows us to assign a function to a variable and then use that variable as a function. It is called function expression.

```
<!DOCTYPE html>
<html>
<body>
    <h1>Demo: Function expression</h1>
    <p id="p1"></p>
    <p id="p2"></p>

    <script>
        var add = function sum(val1, val2) {
            return val1 + val2;
        };

        document.getElementById("p1").innerHTML = add(10,20);
        document.getElementById("p2").innerHTML = sum(10,20); // not valid
  </script>
</body>
</html>
```

Points to Remember:

1. JavaScript a function allows you to define a block of code, give it a name and then execute it as many times as you want.
2. A function can be defined using function keyword and can be executed using () operator.
3. A function can include one or more parameters. It is optional to specify function parameter values

while executing it.

4. JavaScript is a loosely-typed language. A function parameter can hold value of any data type.
5. You can specify less or more arguments while calling function.
6. All the functions can access arguments object by default instead of parameter names.
7. A function can return a literal value or another function.
8. A function can be assigned to a variable with different name.
9. JavaScript allows you to create anonymous functions that must be assigned to a variable.

JavaScript if else condition

JavaScript if else Condition

JavaScript includes if-else conditional statements to control the program flow, similar to other programming languages.

JavaScript includes following forms of if-else conditions:

- If condition
- if-else condition
- else if condition

if condition

Use if conditional statement if you want to execute something based on some condition.

Syntax:

```
if(condition expression)
{
   // code to be executed if condition is true
}
```

Example: if condition

```
<!DOCTYPE html>
<html>
<body>
        <h1>Demo: if condition</h1>

        <script>
                if( 1 > 0)
                {
                        alert("1 is greater than 0");
```

```
        }

        if( 1 < 0)
        {
                alert("1 is less than 0");
        }
  </script>
</body>
</html>
```

In the above example, the first if statement contains 1 > 0 as conditional expression, The conditional expression 1 > 0 will be evaluated to true, so an alert message "1 is greater than 0" will be displayed, whereas conditional expression in second if statement will be evaluated to false, so "1 is less than 0" alert message will not be displayed.

The same way, you can use variables in conditional expression.

```
<!DOCTYPE html>
<html>
<body>
        <h1>Demo: if condition</h1>

        <script>
                var mySal = 1000;
                var yourSal = 500;

                if( mySal > yourSal)
                {
                        alert("My Salary is greater than your salary");
                }
  </script>
</body>
</html>
```

#curly braces { } is not required when if block contains only a single line to execute

Use comparison operators carefully when writing conditional expression. For example, == and === is different.

```
<!DOCTYPE html>
<html>
```

```
<body>
	<h1>Demo: if condition</h1>
	<script>
		if(1=="1")
		{
			alert("== operator does not consider types of operands");
		}
		if(1==="1")
		{
			alert("=== operator considers types of operands");
		}
  </script>
</body>
</html>
```

else Condition

Use else statement when you want to execute the code every time when if condition evaluates to false. The else statement must follow if or else if statement. Multiple else block is NOT allowed.

Syntax:

```
	if(condition expression)
	{
	  //Execute this code..
	}
	else{
	  //Execute this code..
	}
```

Example: else condition

```
<!DOCTYPE html>
<html>
<body>
	<h1>Demo: if-else condition</h1>
		<script>
		var mySal = 500;
		var yourSal = 1000;
		if( mySal > yourSal)
		{
			alert("My Salary is greater than your salary");
```

```
        }
        else
        {
            alert("My Salary is less than or equal to your salary");
        }
  </script>
</body>
</html>
```

else if Condition

Use "else if" condition when you want to apply second level condition after if statement.

Syntax:

```
if(condition expression)
{
   //Execute this code block
}
else if(condition expression){
   //Execute this code block
}
```

Example: else if condition

```
<!DOCTYPE html>
<html>
<body>
    <h1>Demo: if-else condition</h1>

    <script>
        var mySal = 500;
        var yourSal = 1000;

        if( mySal > yourSal)
        {
            alert("My Salary is greater than your salary");
        }
        else if(mySal < yourSal)
        {
            alert("My Salary is less than your salary");
```

```
        }

    </script>
</body>
</html>
```

JavaScript allows multiple else if statements also

```
<!DOCTYPE html>
<html>
<body>
    <h1>Demo: if-else condition</h1>

    <script>
        var mySal = 500;
        var yourSal = 1000;

        if( mySal > yourSal)
        {
            alert("My Salary is greater than your salary");
        }
        else if(mySal < yourSal)
        {
            alert("My Salary is less than your salary");
        }
        else if(mySal == yourSal)
        {
            alert("My Salary is equal to your salary");
        }

    </script>
</body>
</html>
```

Points to Remember:

1. Use if-else conditional statements to control the program flow.
2. JavaScript includes three forms of if condition: if condition, if else condition and else if condition.
3. The if condition must have conditional expression in brackets () followed by single statement or code block wrapped with { }.

4. 'else if' statement must be placed after if condition. It can be used multiple times.
5. 'else' condition must be placed only once at the end. It must come after if or else if statement.

JavaScript Switch

JavaScript switch

The switch is a conditional statement like if statement. Switch is useful when you want to execute one of the multiple code blocks based on the return value of a specified expression.

Syntax:

```
switch(expression or literal value){
  case 1:
    //code to be executed
   break;
  case 2:
    //code to be executed
    break;
  case n:
    //code to be executed
    break;
  default:
    //default code to be executed
    //if none of the above case executed
}
```

As per the above syntax, switch statement contains an expression or literal value. An expression will return a value when evaluated. The switch can includes multiple cases where each case represents a particular value. Code under particular case will be executed when case value is equal to the return value of switch expression. If none of the cases match with switch expression value then the default case will be executed.

Example: switch Statement

```
<!DOCTYPE html>
<html>
<body>
        <h1>Demo: switch case</h1>

        <script>
```

```
var a = 3;

switch (a) {
        case 1:
                alert('case 1 executed');
        case 2:
                alert("case 2 executed");
                break;
  case 3:
                alert("case 3 executed");
                break;
        case 4:
                alert("case 4 executed");
                break;
        default:
                alert("default case executed");
}

   </script>
</body>
</html>
```

In the above example, switch statement contains a literal value as expression. So, the case that matches a literal value will be executed, case 3 in the above example.

The switch statement can also include an expression. A case that matches the result of an expression will be executed.

```
<!DOCTYPE html>
<html>
<body>
        <h1>Demo: switch statement</h1>

        <script>
                var a = 3;

                switch (a/3) {
                        case 1:
                                alert("case 1 executed");
                                break;
```

```
            case 2:
                alert("case 2 executed");
                break;
            case 3:
                alert("case 3 executed");
                break;
            case 4:
                alert("case 4 executed");
                break;
            default:
                alert("default case executed");
        }

    </script>
</body>
</html>
```

In the above example, switch statement includes an expression a/3, which will return 1 (because a = 3). So, case 1 will be executed in the above example.

The switch can also contain string type expression.

```
        <!DOCTYPE html>
        <html>
        <body>
            <h1>Demo: switch with string type</h1>

            <script>
                var str = "bill";

                switch (str)
                {
                    case "steve":
                        alert("This is Steve");
                    case "bill":
                        alert("This is Bill");
                        break;
                    case "john":
                        alert("This is John");
```

```
                        break;
                default:
                        alert("Unknown Person");
                        break;
        }
    </script>
</body>
</html>
```

Multiple cases can be combined in a switch statement.

```
<!DOCTYPE html>
<html>
<body>
        <h1>Demo: combined switch cases</h1>

        <script>
                var a = 2;

                switch (a) {
                        case 1:
                        case 2:
                        case 3:
                                alert("case 1, 2, 3 executed");
                                break;
                        case 4:
                                alert("case 4 executed");
                                break;
                        default:
                                alert("default case executed");
                }
    </script>
</body>
</html>
```

Points to Remember:

The switch is a conditional statement like if statement.

1. A switch statement includes literal value or is expression based

2. A switch statement includes multiple cases that include code blocks to execute.
3. A break keyword is used to stop the execution of case block.
4. A switch case can be combined to execute same code block for multiple cases.

JavaScript Loop

JavaScript for Loop

JavaScript includes for loop like Java or C#. Use for loop to execute code repeatedly.

Syntax:

```
for(initializer; condition; iteration)
{
   // Code to be executed
}
```

The for loop requires following three parts.

Initializer: Initialize a counter variable to start with
Condition: specify a condition that must evaluate to true for next iteration
Iteration: increase or decrease counter
Example: for loop

```
for (var i = 0; i < 5; i++)
{
   console.log(i);
}
```

Output:
0 1 2 3 4

In the above example, var i = 0 is an initializer statement where we declare a variable i with value 0. The second part, i < 5 is a condition where it checks whether i is less than 5 or not. The third part, i++ is iteration statement where we use ++ operator to increase the value of i to 1. All these three parts are separated by semicolon;.

The for loop can also be used to get the values for an array.

Example: for loop

```
var arr = [10, 11, 12, 13, 14];

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

```
{
   console.log(arr[i]);
}
```

Output:

10 11 12 13 14

Please note that it is not mandatory to specify an initializer, condition and increment expression into bracket. You can specify initializer before starting for loop. The condition and increment statements can be included inside the block.

Example: for loop

```
var arr = [10, 11, 12, 13, 14];
var i = 0;

for (; ;) {

   if (i >= 5)
   break;

   console.log(arr[i]);

   i++;
}
```

Output:

10 11 12 13 14

Learn about while loop in the next section.

Points to Remember:

1. JavaScript for loop is used to execute code repeatedly.
2. for loop includes three parts: initialization, condition and iteration. e.g.for(initializer; condition; iteration){ ... }
3. The code block can be wrapped with { } brackets.
4. An initializer can be specified before starting for loop. The condition and increment statements can be included inside the block.

JavaScript While Loop

JavaScript - While Loop

JavaScript includes while loop to execute code repeatedly till it satisfies a specified condition. Unlike for loop, while loop only requires condition expression.

Syntax:

```
while(condition expression)
{
   /* code to be executed
   till the specified condition is true */
}
```

Example: while loop

```
var i =0;

while(i < 5)
{
   console.log(i);
   i++;
}
```

Output:

0 1 2 3 4

Make sure condition expression is appropriate and include increment or decrement counter variables inside the while block to avoid infinite loop.

As you can see in the above example, while loop will execute the code block till i < 5 condition turns out to be false. Initialization statement for a counter variable must be specified before starting while loop and increment of counter must be inside while block.

do while

JavaScript includes another flavour of while loop, that is do-while loop. The do-while loop is similar to while loop the only difference is it evaluates condition expression after the execution of code block. So do-while loop will execute the code block at least once.

Syntax:

```
do{

   //code to be executed

}while(condition expression)
```

Example: do-while loop

```
var i = 0;
```

```
do{

    alert(i);
   i++;

} while(i < 5)
```

Output:

0 1 2 3 4

The following example shows that do-while loop will execute a code block even if the condition turns out to be false in the first iteration.

Example: do-while loop

```
var i =0;

do{

   alert(i);
   i++;

} while(i > 1)
```

Output:

0

Points to Remember:

1. JavaScript while loop & do-while loop execute the code block repeatedly till conditional expression returns true.
2. do-while loop executes the code at least once even if condition returns false.

Error handling in JavaScript

Error handling in JavaScript

JavaScript is a loosely-typed language. It does not give compile-time errors. So some times you will get a runtime error for accessing an undefined variable or calling undefined function etc.

try catch block does not handle syntax errors.
JavaScript provides error-handling mechanism to catch runtime errors using try-catch-finally block, similar to other languages like Java or C#.

Syntax:

```
try
{
   // code that may throw an error
}
catch(ex)
{
   // code to be executed if an error occurs
}
finally{
   // code to be executed regardless of an error occurs or not
}
```

try: wrap suspicious code that may throw an error in try block.

catch: write code to do something in catch block when an error occurs. The catch block can have parameters that will give you error information. Generally catch block is used to log an error or display specific messages to the user.

finally: code in the finally block will always be executed regardless of the occurrence of an error. The finally block can be used to complete the remaining task or reset variables that might have changed before error occurred in try block.

Let's look at simple error handling examples.

Example: Error Handling in JS

```
try
{
   var result  =  Sum(10, 20); // Sum is not defined yet
}
catch(ex)
{
   document.getElementById("errorMessage").innerHTML = ex;
}
```

In the above example, we are calling function Sum, which is not defined yet. So, try block will throw an error

which will be handled by catch block. Ex includes error message that can be displayed.

The finally block executes regardless of whatever happens.

Example: finally Block

```
try
{
    var result  =  Sum(10, 20); // Sum is not defined yet
}
catch(ex)
{
   document.getElementById("errorMessage").innerHTML = ex;
}
finally{
   document.getElementById("message").innerHTML = "finally block executed";
}
```

throw

Use throw keyword to raise a custom error.

Example: throw Error

```
try
{
   throw "Error occurred";
}
catch(ex)
{
   alert(ex);
}
```

You can use JavaScript object for more information about an error.

Example: throw error with error info

```
try
{
   throw {
      number: 101,
      message: "Error occurred"
   };
}
```

```
catch (ex) {
   alert(ex.number + "- " + ex.message);
}
```

OOPs in JavaScript

Define Class in JavaScript

JavaScript ECMAScript 5, does not have class type. So it does not support full object oriented programming concept as other languages like Java or C#. However, you can create a function in such a way so that it will act as a class.

The following example demonstrates how a function can be used like a class in JavaScript.

Example: Class in JavaScript

```
function Person() {
   this.firstName = "unknown";
   this.lastName = "unknown";
}

var person1 = new Person();
person1.firstName = "Steve";
person1.lastName = "Jobs";

alert(person1.firstName + " " + person1.lastName);

var person2 = new Person();
person2.firstName = "Bill";
person2.lastName = "Gates";

alert(person2.firstName + " " + person2.lastName );
```

In the above example, a Person() function includes firstName, lastName & age variables using this keyword. These variables will act like properties. As you know, we can create an object of any function using new keyword, so person1 object is created with new keyword. So now, Person will act as a class and person1 & person2 will be its objects (instances). Each object will hold their values separately because all the variables are defined with this keyword which binds them to particular object when we create an object using new keyword.

So this is how a function can be used like a class in the JavaScript.

Add Methods in a Class

We can add a function expression as a member variable in a function in JavaScript. This function expression will act like a method of class.

Example: Method in Class

```
function Person() {
        this.firstName = "unknown";
        this.lastName = "unknown";
        this.getFullName = function(){
           return this.firstName + " " + this.lastName;
        }
    };

var person1 = new Person();
person1.firstName = "Steve";
person1.lastName = "Jobs";

alert(person1.getFullName());

var person2 = new Person();
person2.firstName = "Bill";
person2.lastName = "Gates";

alert(person2.getFullName());
```

In the above example, the Person function includes function expression that is assigned to a member variable getFullName. So now, getFullName() will act like a method of the Person class. It can be called using dot notation e.g. person1.getFullName().

Constructor

In the other programming languages like Java or C#, a class can have one or more constructors. In JavaScript, a function can have one or more parameters. So, a function with one or more parameters can be used like a constructor where you can pass parameter values at the time or creating an object with new keyword.

Example: Constructor

```
function Person(FirstName, LastName, Age) {
    this.firstName = FirstName || "unknown";
    this.lastName = LastName || "unknown";
```

```
    this.age = Age || 25;
    this.getFullName = function () {
        return this.firstName + " " + this.lastName;
    }
};

var person1 = new Person("James","Bond",50);
alert(person1.getFullName());

var person2 = new Person("Tom","Paul");
alert(person2.getFullName());
```

In the above example, the Person function includes three parameters FirstName, LastName and Age. These parameters are used to set the values of a respective property.

Note:

Please notice that parameter assigned to a property, if parameter value is not passed while creating an object using new then they will be undefined.

Properties with Getters and Setters

As you learned in the previous section, Object.defineProperty() method can be used to define a property with getter & setter.

The following example shows how to create a property with getter & setter.

Example: Property

```
function Person() {
    var _firstName = "unknown";

    Object.defineProperties(this, {
        "FirstName": {
            get: function () {
                return _firstName;
            },
            set: function (value) {
                _firstName = value;
            }
        }
    });
};
```

```
var person1 = new Person();
person1.FirstName = "Steve";
alert(person1.FirstName );

var person2 = new Person();
person2.FirstName = "Bill";
alert(person2.FirstName );
```

In the above example, the Person() function creates a FirstName property by using Object.defineProperties() method. The first argument is this, which binds FirstName property to calling object. Second argument is an object that includes list of properties to be created. We have specified FirstName property with get & set function. You can then use this property using dot notation as shown above.

Read-only Property

Do not specify set function in order to create read-only property as shown below.

Example: Read-only Property

```
function Person(firstName) {

   var _firstName = firstName || "unknown";

      Object.defineProperties(this, {
         "FirstName": {
            get: function () {
               return _firstName;
            }
         }
      });
   };
var person1 = new Person("Steve");
//person1.FirstName = "Steve"; -- will not work
alert(person1.FirstName );

var person2 = new Person("Bill");
//person2.FirstName = "Bill"; -- will not work
alert(person2.FirstName );
```

Multiple Properties

Specify more than one property in defineProperties() method as shown below.

Example: Multiple Properties

```
function Person(firstName, lastName, age) {
   var _firstName = firstName || "unknown";
   var _lastName = lastName || "unknown";
   var _age = age || 25;

   Object.defineProperties(this, {
      "FirstName": {
         get: function () { return _firstName },
         set: function (value) { _firstName = value }
      },
      "LastName": {
         get: function () { return _lastName },
         set: function (value) { _lastName = value }
      },
      "Age": {
         get: function () { return _age },
         set: function (value) { _age = value }
      }
   });

   this.getFullName = function () {
         return this.FirstName + " " + this.LastName;
   }
};
var person1 = new Person();
person1.FirstName = "John";
person1.LastName = "Bond";
alert(person1.getFullName());
```

JavaScript Object in Depth

You have already learned about JavaScript object in the JavaScript Basics section. Here, you will learn about object in detail.

As you know, object in JavaScript can be created using object literal, object constructor or constructor function. Object includes properties. Each property can either be assigned a literal value or a function.

Consider the following example of objects created using object literal and constructor function.

Example: JavaScript Object

```
// object literal
var person = {
  firstName:'Steve',
  lastName:'Jobs'
};

// Constructor function
function Student(){
  this.name = "John";
  this.gender = "Male";
  this.sayHi = function(){
    alert('Hi');
  }
}
var student1 = new Student();
console.log(student1.name);
console.log(student1.gender);
student1.sayHi();
```

In the above example, person object is created using object literal that includes firstName and lastName properties and student1 object is created using constructor function Student that includes name, gender and sayHi properties where function is assigned to sayHi property.

Note:
Any javascript function using which object is created is called constructor function.
Use Object.keys() method to retrieve all the properties name for the specified object as a string array.

Example: Edit Property Descriptor

```
function Student(){
  this.title = "Mr.";
  this.name = "Steve";
  this.gender = "Male";
  this.sayHi = function(){
```

```
    alert('Hi');
  }
}
var student1 = new Student();

Object.keys(student1);
```

Output:

["title", "name", "gender", "sayHi"]

Use for-in loop to retrieve all the properties of an object as shown below.

Example: Enumerable Properties

```
function Student(){
  this.title = "Mr.";
  this.name = "Steve";
  this.gender = "Male";
  this.sayHi = function(){
    alert('Hi');
  }
}
var student1 = new Student();

//enumerate properties of student1
for(var prop in student1){
  console.log(prop);
}
```

Output:

title name gender sayHi

Property Descriptor

In JavaScript, each property of an object has property descriptor which describes the nature of a property. Property descriptor for a particular object's property can be retrieved using Object.getOwnPropertyDescriptor() method.

Syntax:

```
Object.getOwnPropertyDescriptor(object, 'property name')
```

The getOwnPropertyDescriptor method returns a property descriptor for a property that directly defined in the specified object but not inherited from object's prototytpe.

The following example display property descriptor to the console.

Example: Property Descriptor

```
var person = {
  firstName:'Steve',
  lastName:'Jobs'
};

function Student(){
  this.name = "John";
  this.gender = "Male";
  this.sayHi = function(){
    alert('Hi');
  }
}

var student1 = new Student();

console.log(Object.getOwnPropertyDescriptor(person,'firstName'));
console.log(Object.getOwnPropertyDescriptor(student1,'name'));
console.log(Object.getOwnPropertyDescriptor(student1,'sayHi'));
```

Output:

Object {value: "Steve", writable: true, enumerable: true, configurable: true}
Object {value: "John", writable: true, enumerable: true, configurable: true}
Object {value: function, writable: true, enumerable: true, configurable: true}

As you can see in the above output, the property descriptor includes the following 4 important attributes.

Attribute	Description
value	Contains an actual value of a property.
writable	Indicates that whether a property is writable or read-only. If true than value can be changed and if false then value cannot be changed and will throw an exception in strict mode
enumerable	Indicates whether a property would show up during the enumeration using for-in loop or Object.keys() method.

configurable	Indicates whether a property descriptor for the specified property can be changed or not. If true then any of this 4 attribute of a property can be changed using Object.defineProperty() method.

Object.defineProperty()

The Object.defineProperty() method defines a new property on the specified object or modifies an existing property or property descriptor.

Syntax:

Object.defineProperty(object, 'property name', descriptor)

The following example demonstrates modifying property descriptor.

Example: Edit Property Descriptor

```
'use strict'

function Student(){
  this.name = "Steve";
  this.gender = "Male";

}

var student1 = new Student();

Object.defineProperty(student1,'name', { writable:false} );

try
{
   student1.name = "James";
   console.log(student1.name);
}
catch(ex)
{
   console.log(ex.message);
}
```

The above example, it modifies writable attribute of name property of student1 object using Object.defineProperty(). So, name property can not be changed. If you try to change the value of name property then it would throw an exception in strict mode. In non-strict mode, it won't throw an exception but it also won't change a value of name property either.

The same way, you can change enumerable property descriptor as shown below.

Example: Edit Property Descriptor

```
function Student(){
  this.name = "Steve";
  this.gender = "Male";
}

var student1 = new Student();

//enumerate properties of student1
for(var prop in student1){
  console.log(prop);
}

//edit enumerable attributes of name property to false
Object.defineProperty(student1,'name',{ enumerable:false });

console.log('After setting enumerable to false:');

for(var prop in student1){
  console.log(prop);
}
```

Output:

name
gender

After setting enumerable to false:
gender

In the above example, it display all the properties using for-in loop. But, once you change the enumerable attribute to false then it won't display name property using for-in loop. As you can see in the output, after setting enumerable attributes of name property to false, it will not be enumerated using for-in loop or even Object.keys() method.

The Object.defineProperty() method can also be used to modify configurable attribute of a property which restrict changing any property descriptor attributes further.

The following example demonstrates changing configurable attribute.

Example: Edit Property Descriptor

```
'use strict';

function Student(){
  this.name = "Steve";
  this.gender = "Male";

}
var student1 = new Student();

Object.defineProperty(student1,'name',{configurable:false});// set configurable to false

try
{
   Object.defineProperty(student1,'name',{writable:false}); // change writable attribute
}
catch(ex)
{
   console.log(ex.message);
}
```

In the above example, Object.defineProperty(student1,'name',{configurable:false}); sets configurable attribute to false which make student1 object non configurable after that. Object.defineProperty(student1,'name',{writable:false}); sets writable to false which will throw an exception in strict mode because we already set configurable to false.

Define New Property

The Object.defineProperty() method can also be used to define a new properties with getters and setters on an object as shown below.

Example: Define New Property

```
function Student(){
  this.title = "Mr.";
  this.name = "Steve";
}
var student1 = new Student();
Object.defineProperty(student1,'fullName',{
```

```
  get:function(){
    return this.title + ' ' + this.name;
  },
  set:function(_fullName){
    this.title = _fullName.split(' ')[0];
    this.name = _fullName.split(' ')[1];
  }
});

student1.fullName = "Mr. John";

console.log(student1.title);
console.log(student1.name);
```

Output:

Mr.

John

JavaScript - this Keyword

The this keyword is one of the most widely used and yet confusing keyword in JavaScript. Here, you will learn everything about this keyword.

this points to a particular object. Now, which is that object is depends on how a function which includes 'this' keyword is being called.

Look at the following example and guess what the result would be?

```
<script>
var myVar = 100;

function WhoIsThis() {
   var myVar = 200;

   alert(myVar); // 200
   alert(this.myVar); // 100
}

WhoIsThis(); // inferred as window.WhoIsThis()
```

```
var obj = new WhoIsThis();
alert(obj.myVar);
</script>
```

The following four rules applies to this in order to know which object is referred by this keyword.

- Global Scope
- Object's Method
- call() or apply() method
- bind() method

Global Scope

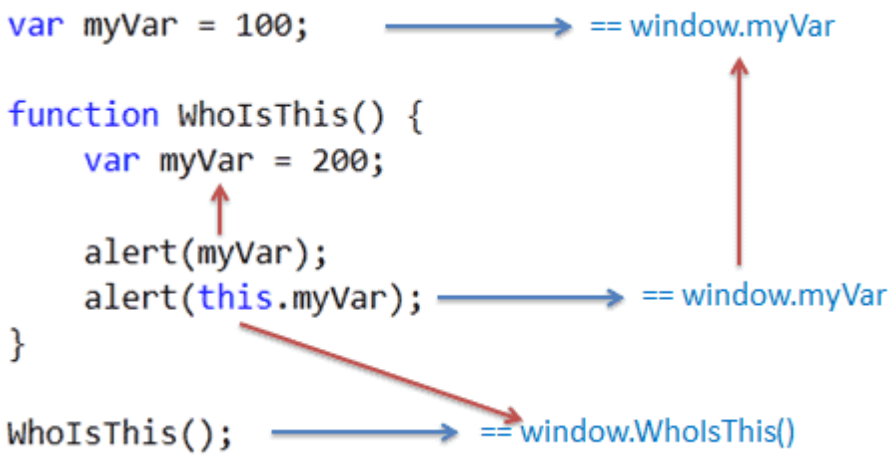
If a function which includes 'this' keyword, is called from the global scope then this will point to the window object. Learn about global and local scope here.

Example: this keyword

```
<script>
var myVar = 100;

function WhoIsThis() {
   var myVar = 200;

   alert("myVar = " + myVar); // 200
   alert("this.myVar = " + this.myVar); // 100
}

WhoIsThis(); // inferred as window.WhoIsThis()
</script>
```

In the above example, a function WhoIsThis() is being called from the global scope. The global scope means in the context of window object. We can optionally call it like window.WhoIsThis(). So in the above example, this keyword in WhoIsThis() function will refer to window object. So, this.myVar will return 100. However, if you access myVar without this then it will refer to local myVar variable defined in WhoIsThis() function.

The following figure illustrates the above example.



Note: In the strict mode, value of 'this' will be undefined in the global scope.

'this' points to global window object even if it is used in an inner function. Consider the following example.

Example: this keyword inside inner function

```
var myVar = 100;

function SomeFunction() {

   function WhoIsThis() {
      var myVar = 200;

      alert("myVar = " + myVar); // 200
      alert("this.myVar = " + this.myVar); // 100
   }

   WhoIsThis();
}

SomeFunction();
```

So, if 'this' is used inside any global function and called without dot notation or using window. then this will refer to global object which is default window object.

this Inside Object's Method

As you have learned here, you can create an object of a function using new keyword. So, when you create

an object of a function using new keyword then this will point to that particular object. Consider the following example.

Example: this keyword

```
var myVar = 100;

function WhoIsThis() {
   this.myVar = 200;
}
var obj1 = new WhoIsThis();

var obj2 = new WhoIsThis();
obj2.myVar = 300;

alert(obj1.myVar); // 200
alert(obj2.myVar); // 300
```

In the above example, this points to obj1 for obj1 instance and points to obj2 for obj2 instance. In JavaScript, properties can be attached to an object dynamically using dot notation. Thus, myVar will be a property of both the instances and each will have a separate copy of myVar.

Now look at the following example.

Example: this keyword

```
var myVar = 100;

function WhoIsThis() {
   this.myVar = 200;

   this.display = function(){
      var myVar = 300;

      alert("myVar = " + myVar); // 300
      alert("this.myVar = " + this.myVar);  // 200
   };
}
var obj = new WhoIsThis();

obj.display();
```

In the above example, obj will have two properties myVar and display, where display is a function expression. So, this inside display() method points to obj when calling obj.display().

this behaves the same way when object created using object literal, as shown below.

Example: this keyword

```
var myVar = 100;

var obj = {
        myVar : 300,
            whoIsThis: function(){
                var myVar = 200;

                alert(myVar); // 200
                alert(this.myVar); // 300
                }
};

obj.whoIsThis();
```

call() and apply()

In JavaScript, a function can be invoked using () operator as well as call() and apply() method as shown below.

Example: Function call

```
function WhoIsThis() {
   alert('Hi');
}

WhoIsThis();
WhoIsThis.call();
WhoIsThis.apply();
```

In the above example, WhoIsThis(), WhoIsThis.call() and WhoIsThis.apply() executes a function in the same way.

The main purpose of call() and apply() is to set the context of this inside a function irrespective whether that function is being called in the global scope or as object's method.

You can pass an object as a first parameter in call() and apply() to which the this inside a calling function

should point to.

The following example demonstrates the call() & apply().

Example: call() & apply()

```
var myVar = 100;

function WhoIsThis() {

   alert(this.myVar);
}

var obj1 = { myVar : 200 , whoIsThis: WhoIsThis };

var obj2 = { myVar : 300 , whoIsThis: WhoIsThis };

WhoIsThis(); // 'this' will point to window object

WhoIsThis.call(obj1); // 'this' will point to obj1

WhoIsThis.apply(obj2); // 'this' will point to obj2

obj1.whoIsThis.call(window); // 'this' will point to window object

WhoIsThis.apply(obj2); // 'this' will point to obj2
```

As you can see in the above example, when the function WhoIsThis is called using () operator (like WhoIsThis()) then this inside a function follows the rule- refers to window object. However, when the WhoIsThis is called using call() and apply() method then this refers to an object which is passed as a first parameter irrespective of how the function is being called.

Therefore, this will point to obj1 when a function got called as WhoIsThis.call(obj1). In the same way, this will point to obj2 when a function got called like WhoIsThis.apply(obj2)

bind()

The bind() method was introduced since ECMAScript 5. It can be used to set the context of 'this' to a specified object when a function is invoked.

The bind() method is usually helpful in setting up the context of this for a callback function. Consider the following example.

Example: bind()

```
var myVar = 100;

function SomeFunction(callback)
{
   var myVar = 200;

   callback();
};

var obj = {
        myVar: 300,
        WhoIsThis : function() {
           alert("'this' points to " + this + ", myVar = " + this.myVar);
        }
    };

SomeFunction(obj.WhoIsThis);
SomeFunction(obj.WhoIsThis.bind(obj));
```

In the above example, when you pass obj.WhoIsThis as a parameter to the SomeFunction() then this points to global window object insted of obj, because obj.WhoIsThis() will be executed as a global function by JavaScript engine. You can solve this problem by explicitly setting this value using bind() method. Thus, SomeFunction(obj.WhoIsThis.bind(obj)) will set this to obj by specifying obj.WhoIsThis.bind(obj).

Precedence

So these 4 rules applies to this keyword in order to determine which object this refers to. The following is precedence of order.

- bind()
- call() and apply()
- Object method
- Global scope

So, first check whether a function is being called as callback function using bind()? If not then check whether a function is being called using call() or apply() with parmeter? If not then check whether a function is being called as an object function? Otherise check whether a function is being called in the global scope without

dot notation or using window object.

Thus, use these simple rules in order to know which object the 'this' refers to inside any function.

JavaScript new Keyword

We have seen in the Object section that an object can be created with new keyword. Here, you will learn about the steps it performs while creating an object.

```
function MyFunc() {
   this.x = 100;
}

var obj1 = new MyFunc();
obj1.x;
```

The built-in primitive types in JavaScript are functions only e.g. Object, Boolean, String, Number is built-in JavaScript functions. If you write Object in browser's console window and press Enter then you will see the output "function Object()".

In the above example, we have created an object of MyFunc using new keyword. This MyFunc() is called a constructor function. The new keyword constructs and returns an object (instance) of a constructor function.

The new keyword performs following four tasks:

1. It creates new empty object e.g. obj = { };
2. It sets new empty object's invisible 'prototype' property to be the constructor function's visible and accessible 'prototype' property. (Every function has visible 'prototype' property whereas every object includes invisible 'prototype' property)
3. It binds property or function which is declared with this keyword to the new object.
4. It returns newly created object unless the constructor function returns a non-primitive value (custom JavaScript object). If constructor function does not include return statement then compiler will insert 'return this;' implicitly at the end of the function. If the constructor function returns a primitive value then it will be ignored.

Let's see how new keyword creates an object using following example.

Example: new keyword

```
function MyFunc() {
   var myVar = 1;
   this.x = 100;
}
```

```
MyFunc.prototype.y = 200;
var obj1 = new MyFunc();
obj1.x; // 100
obj1.y; // 200
```

Let's understand what happens when you create an object (instance) of MyFunc() using new keyword.

First of all, new keyword creates an empty object - { }.

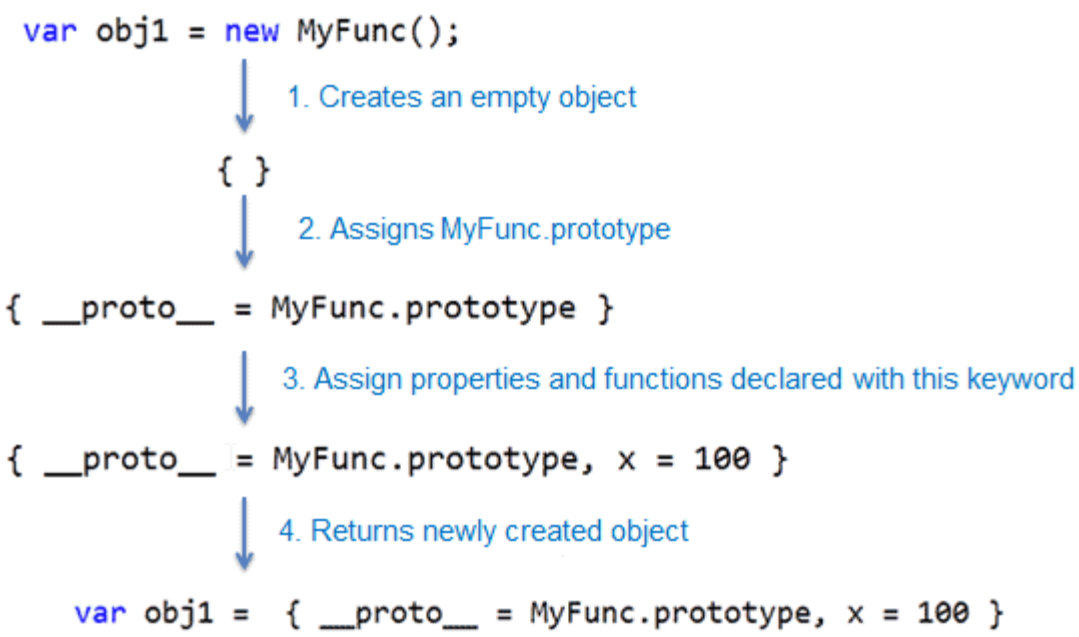
Second, it set's invisible 'prototype' property (or attribute) of this empty object to myFunc's prototype property. As you can see in the above example, we have assigned new property 'y' using MyFunc.prototype.y. So, new empty object will also have same prototype property as MyFunc which includes y property.

In third step, it binds all the properties and function declared with this keyword to new empty object. Here, MyFunc includes only one property x which is declared with this keyword. So new empty object will now include x property. MyFunc also includes myVar variable which does not declared with this keyword. So myVar will not be included in new object.

In the fourth and last step, it will return this newly created object. MyFunc does not include return statement but compiler will implicitly insert 'return this' at the end.

So thus, object of MyFunc will be returned using new keyword.

The following figure illustrates the above process.



Object creation process

The new keyword ignores return statement that returns primitive value.

Example: new keyword

```
function MyFunc() {
   this.x = 100;

   return 200;
}

var obj = new MyFunc();
alert(obj.x); // 100
```

If function returns non-primitive value (custom object) then new keyword does not perform above 4 tasks.

Example: new keyword

```
function MyFunc() {
   this.x = 100;

   return { a: 123 };
}

var obj1 = new MyFunc();
```

```
alert(obj1.x); // undefined
```

Thus, new keyword builds an object of a function in JavaScript.

Inheritance in JavaScript

Inheritance is an important concept in object oriented programming. In the classical inheritance, methods from base class get copied into derived class.

In JavaScript, inheritance is supported by using prototype object. Some people call it "Prototypal Inheriatance" and some people call it "Behaviour Delegation".

Let's see how we can achieve inheritance like functionality in JavaScript using prototype object.

Let's start with the Person class which includes FirstName & LastName property as shown below.

```
function Person(firstName, lastName) {
   this.FirstName = firstName || "unknown";
   this.LastName = lastName || "unknown";
};

Person.prototype.getFullName = function () {
   return this.FirstName + " " + this.LastName;
}
```

In the above example, we have defined Person class (function) with FirstName & LastName properties and also added getFullName method to its prototype object.

Now, we want to create Student class that inherits from Person class so that we don't have to redefine FirstName, LastName and getFullName() method in Student class. The following is a Student class that inherits Person class.

Example: Inheritance

```
function Student(firstName, lastName, schoolName, grade)
{
   Person.call(this, firstName, lastName);

   this.SchoolName = schoolName || "unknown";
   this.Grade = grade || 0;
```

```
}
//Student.prototype = Person.prototype;
Student.prototype = new Person();
Student.prototype.constructor = Student;
```

Please note that we have set Student.prototype to newly created person object. The new keyword creates an object of Person class and also assigns Person.prototype to new object's prototype object and then finally assigns newly created object to Student.prototype object. Optionally, you can also assign Person.prototype to Student.prototype object.

Now, we can create an object of Student that uses properties and methods of the Person as shown below.

Example: Inheritance

```
function Person(firstName, lastName) {
   this.FirstName = firstName || "unknown";
   this.LastName = lastName || "unknown";
}

Person.prototype.getFullName = function () {
   return this.FirstName + " " + this.LastName;
}
function Student(firstName, lastName, schoolName, grade)
{
   Person.call(this, firstName, lastName);

   this.SchoolName = schoolName || "unknown";
   this.Grade = grade || 0;
}
//Student.prototype = Person.prototype;
Student.prototype = new Person();
Student.prototype.constructor = Student;

var std = new Student("James","Bond", "XYZ", 10);

alert(std.getFullName()); // James Bond
alert(std instanceof Student); // true
alert(std instanceof Person); // true
```

Thus we can implement inheritance in JavaScript.

JavaScript Encapsulation

The JavaScript Encapsulation is a process of binding the data (i.e. variables) with the functions acting on that data. It allows us to control the data and validate it. To achieve an encapsulation in JavaScript: -

Use var keyword to make data members private.
Use setter methods to set the data and getter methods to get that data.
The encapsulation allows us to handle an object using the following properties:

Read/Write - Here, we use setter methods to write the data and getter methods read that data.
Read Only - In this case, we use getter methods only.
Write Only - In this case, we use setter methods only.

Encapsulation Example

```
<script>
class Student
 {
  constructor()
  {
    var name;
    var marks;
  }
     getName()
     {
       return this.name;
     }
   setName(name)
   {
    this.name=name;
   }

   getMarks()
   {
    return this.marks;
   }
  setMarks(marks)
  {
   this.marks=marks;
```

```
    }

    }
    var stud=new Student();
     stud.setName("John");
     stud.setMarks(80);
     document.writeln(stud.getName()+" "+stud.getMarks());
</script>
```

Activity

1. Design signup form to validate username, password, and phone numbers etc using Java script.
2. Write a JavaScript program to determine whether a given year is a leap year in the Gregorian calendar.
3. Write a JavaScript program to convert temperatures to and from celsius, Fahrenheit.
4. Write a JavaScript to design a simple calculator to perform the following operations: sum, product, difference and quotient
5. Write a JavaScript that calculates the squares and cubes of the numbers from 0 to 10 and outputs HTML text that displays the resulting values in an HTML table format.