About Advanced JavaScript

JavaScript is widely used light-weight, interpreted, Object-oriented client side scripting language.It helps developer to create dynamic web page through various features like creating reusable code, validation through user action, partial page rendering as well as developer can establish connection to remote server.

JavaScript helps developer to enhance web page performance through various asynchronous patterns.

In this course, you would learn how to build dynamic web page through functions as well as variable declaration through scope, access the JSON data format through user defined function along with AJAX concept, as well as understand how to create object through Object literal and Construction patterns along with some of the Object-oriented features like Encapsulation and Inheritance.

Apart from these concepts, you will also learn how to implement asynchronous patterns like Promises and Callback methods.

Why JavaScript?

In short we can consider JavaScript is exceedingly versatile and flexible which enables developer to create dynamic web pages with vast amount of extra functionality such as Browser APIs, Third-party APIs, Third-party frameworks and libraries…etc.

Advanced Features and Benefits

Advanced JavaScript can be merely consider as JavaScript itself with added advanced features and benefits. The following features and benefits makes the user to work with JavaScript in the superfluous level.

Scope and Functions: JavaScript has introduced lexical scope concept, which makes the programmer to work wider as well as limited with accessibility (visibility) of variables and enhanced function declarations through which developer can achieve more seemly approaches for various needs.

IIFE: Immediately Invoked Function Expression is a supplementary feature to JavaScript, which mainly focus on data privacy and avoid data being overridden, so that a function cannot be accessed by the outside world.

Closure: This feature can be considered as combination of Function and the Lexical environment, a function declared implementing this concept can be considered as a Closure. This feature of JavaScript helps the programmers to write better code.

Object Oriented Concepts: JavaScript supports Object oriented concepts like Classes, Objects, Encapsulation, Inheritance, Factory and Constructor which fashions the coding dominant and flexible.

Asynchronous Execution: JavaScript is basically a single threaded language which can expand its approaches beyond its basic architecture using Asynchronous Execution with use of various mechanism like Promises, Generators, Async Queue and Callback makes developer to code so productive where speed and timing is the main concern.

What is JavaScript?

JavaScript is a full-fledged client side language used for developing web applications. JavaScript is easy to learn, debug and test. It is event-based, platform independent and interpreted language with all the procedural programming capabilities.

Developers choose JavaScript to create dynamic, interactive and scalable web applications. JavaScript helps developers in extending the functionalities of web pages effectively. It helps primarily in reducing the number of request-response cycles and decreasing the network bandwidth. JavaScript helps in increasing the response time.

In this course we will discuss on the basic concepts of JavaScript, validating the forms using JavaScript, creating interactive web pages by handling events, updating the view by manipulating the DOM.

JavaScript was not originally named as JavaScript. It was created as a scripting language in 1995 over the span of 10 days with the name 'LiveScript'.

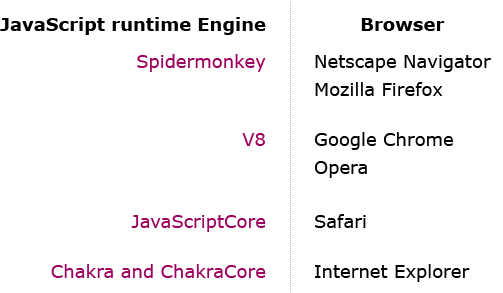
Scripting language is the one that controls the environment in which it runs.

But now JavaScript is a full-fledged programming language because of it's huge capabilities for developing web applications. It contains core language features like control structures, operators, statements, objects and functions.

JavaScript is an interpreted language. The browser interprets the JavaScript code embedded inside the web page, executes it and displays the output. It is not compiled to any other form to be executed.

All the modern web browsers come along with the JavaScript Engine. This engine takes responsibility of interpreting the JavaScript lines of code. There is absolutely no need to include any file or import any package inside the browser for JavaScript interpretation.

Popular JavaScript engines include:



JavaScript is a wide-ranging dynamic programming language, which can provide dynamic interactivity on websites when it is applied to a HTML document.

Let us have a glance on JavaScript...

When we talk about creating a web page, commonly they frame under two categories:

Static web pages

Dynamic web pages

Static Web Pages

These are stationary web pages, which will be loaded on the client’s browser as exactly how they are stored on the web server. As they are having static content the user can only read the information but can’t do any modification or interact with the information.

Static web pages are created using only basic HTML tags. Static web pages are only used when the developer needs to populate information which is no more required to be modified.



Dynamic Web Pages

They can be considered as a two-way streets, where developer can provide content to users, and also allow users to submit information back to the server. They can be classified into two categories:

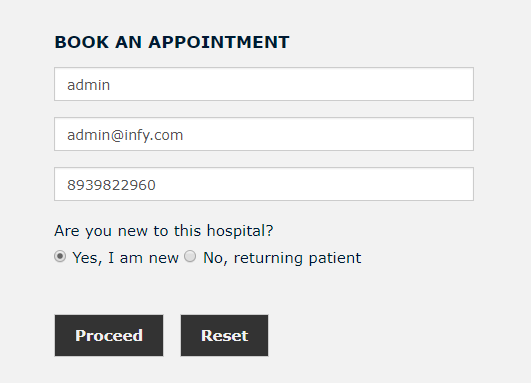
Server-Side: These are those Web pages that can be dynamically changed when a web page is loaded or visited.

Client-Side: These are those Web pages that change dynamically in response to an action within that web page, such as a mouse or a keyboard action



Let us consider the Online Appointment Booking page before and after clicking Proceed button.

Before clicking Proceed button:



Functions and Scope

What are Functions?

According to programming language implementations Functions are considered to blocks of statements which can be used by developer to achieve modularity and reusability concepts.

But in JavaScript, Functions play more vital role than this.

Functions in JavaScript :

Can be used to create methods

Can be used to create classes

Can be treated as an value

Can be passed as arguments to another function

Can inherit from other objects

Can have user defined properties and methods

Function Declaration and Expression

Developer can declare Function using function keyword followed by user-defined function name along with parentheses(). Based on requirement these parentheses can have zero or more than one parameters separated by commas.

syntax:

function functionName (parameter1, parameter2,…){

// zero or more than one executable statements

}

Functions can be declared to return some valid value and can also be declared without name which are known as anonymous function.To invoke declared function developer has either follow anyone of procedure mentioned below in his code:

Event has to be triggered.

Should be automatically executed or Self invoked.

Call the function through JavaScript code.

Function Expression

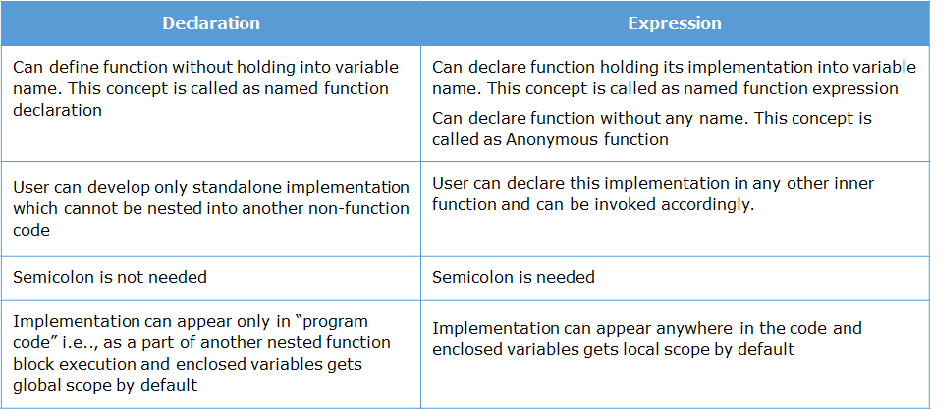
It is a concept in which function implementation can be stored into variable. In ES2015 this concept has been implemented through arrow functions.

syntax:

var x= function functionName(parameter1, parameter2,…){

// zero or more than one executable statements

}



What is Scope and JavaScript Compiler ?

While developing any application one of the traditional problem which developer encounters is to analysis where the data would be stored and its easy quick accessibility for reading and writing into and from the application.

This traditional problem in JavaScript has been resolved by below mentioned 4 different options for data storage concept.

Literal values: These are the values which represent simple value without at storage allocation. Strings, Number, Booleans, Objects, Arrays, Functions, Regular Expressions and the special value “Null” and “undefined” are some of the built-in JavaScript Literal values, which can be used to create Literal value while developing any application.

Variables: Any user defined object using var keyword represent Variable in the application.

Array items: Represents numerically indexed value declared using JavaScript Array object.

Object members: Represents string-indexed value within JavaScript object.

Each of these options internal deals with some performance cost while accessing and manipulating them. So in order to develop any application with best performance, developer need to think about the data storage location of above mentioned options, which technically called as Scope of the Object.

Definition of Scope

Accessibility or Visibility of user defined data storage options in the code is called as scope of that particular object.

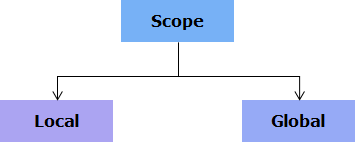
Its one of the importance feature of JavaScript which avoids naming conflicts and internally system provides automatic memory management.

JavaScript uses concept of Function scope, by which user defined data storage option value can be modified/revisited with the associated function declaration only.

JavaScript provides user with two different types of Scopes.

Local Scope

Global Scope



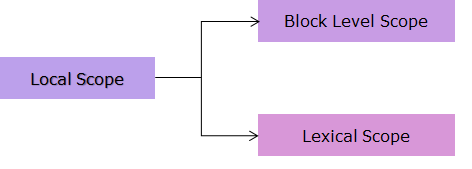
Is one of the predominant scope model which has also referred as Static scope or Compile time scope.

According to ECMA Script rules this concept has been implemented through

Block level Scope: Variables declared within Function would have this scope. These variables cannot be accessed outside the declared function block.

Lexical Scope: Scope of Variables and block of scope would be decided based on location where user declares them in code, in turn which is similar to Block level concept.

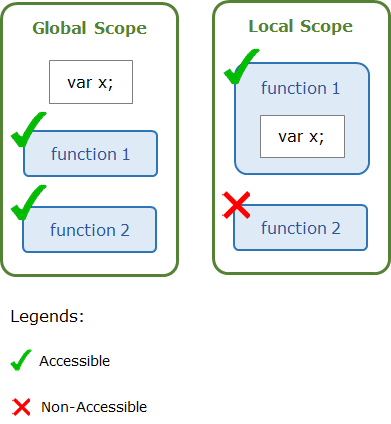
If developer declares multiple variables with same name in nested blocks, then compiler would give highest preference to innermost declared one rather than outside one.



Is a model in which any variable declared outside the Function block or declared globally would have Global scope through the application.

By assigning any property to window object, that property starts behaving as Global in nature.

Developer can declare a variable without var keyword outside Function declaration, in such situation variable gets Global scope and whereas if same declaration is used with in the Function block, in such situation compiler looks for scope chain until it finds the declaration for this variable.



JSON

What is JSON?

JavaScript Object Notation can be considered as:

Most lightweight data interchange format, which can be used for transferring data between Client to Server and vice-versa in easiest way.

Subset of JavaScript.

Language dependent but supported by many modern programming languages.

Readable in nature as it follows key-value pairs concept, due to this nature its internally easy to implement based on any coding requirement.

Value in JSON can represent any data format like String, Number, Object , Array , Boolean and special values.

Object: Is an unordered element which represents key-value pairs, where key can represent any string, value can represent any JSON value like Arrays or another JSON object or combination of array of objects or array of JSON objects.

Array: Is a collection of an ordered data, this data can represent any JSON value like Arrays or another JSON object or combination of array of objects or array of JSON objects.

String: Is a value which should wrapped within double quotes and can hold different characters as shown below:

Number: Is similar to basic Number data type as of C or Java programming language and can hold different combinations of digits as shown below. Octal and Hexadecimal formats are not acceptable under this category.

For example if you want to create JSON Array Object which can be used to represent Doctor name and Hospital location, below mentioned code snippet can be used.

//creating variable to hold JSON Array Object

var doctorJSON= [

{“doctorName":"Dr Kenneth",“hospitalLocation":"Delhi"},

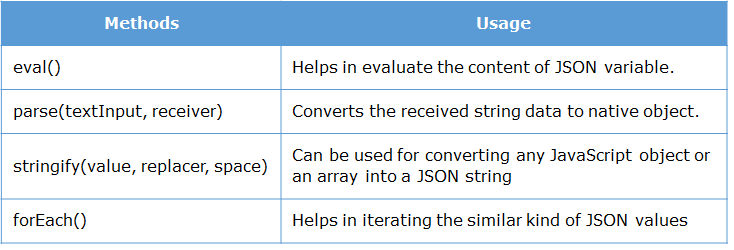
{“doctorName":"Dr Volta",“hospitalLocation":“Chennai"},

{“doctorName":"Dr Shri",“hospitalLocation":“Pune"},

{“doctorName":"Dr Jen",“hospitalLocation":"Delhi"}

];

Basic JSON Methods



Some of the points to be noted while using above mentioned methods are:

Don’t use eval(), this may lead horrendous security problems in order to avoid this issue, you can use parse().

parse() returns the same value until associated parameter remains same, it may return undefined or null if associated parameter is deleted.

parse() throws a SyntaxError exception if the string to parse is not valid JSON and it doesn’t allow trailing comments.

Now let us enhance our previous declared JSON code snippet as shown below to populate the individual values onto console of the browser.

AJAX

What is AJAX?

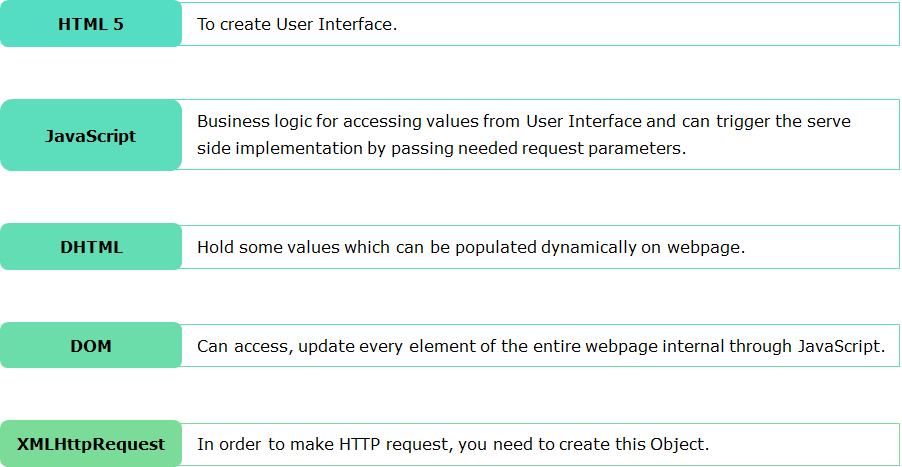
Abbreviation of AJAX is Asynchronous JavaScript And XML.

It can be considered as a client-side scripting concept, which can be used to communicate with Servers through XMLHttpRequest Object.

For any interaction User can initiate the communication using different data formats like plain text, HTML, XML and JSON.

Due to its basic Asynchronous nature, any transaction involved through this concept wouldn’t reload the entire page data but indeed it updates only need content based on user’s request.

Key Elements of AJAX



It is a special JavaScript object, which helps in achieving the Asynchronous communication in your webpage.

Instead of reloading entire page, partial content can be loaded with respective element through this Object.

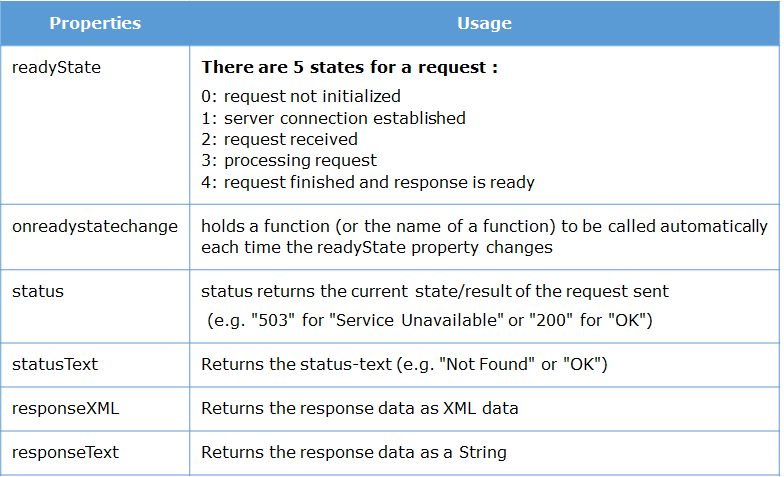
This Object is a predecessor which was introduced as an ActiveX Object called XMLHttp by Microsoft as a part of Internet Explorer 5.0 browser.

At the same time other browsers like Safari, Mozilla and all, started implementing XMLHttpRequest object which internally supports all the methods and properties of originally Microsoft ActiveX object, due to this enhancement automatically Microsoft also implemented XMLHttpRequest object.

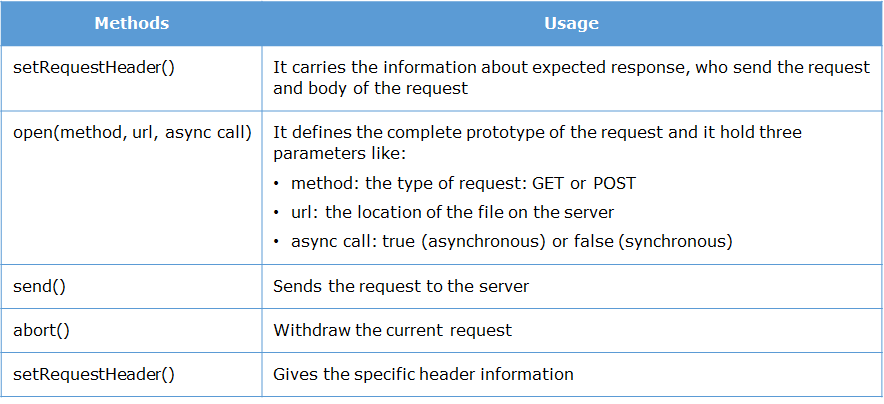
It has capacity of transferring various data formats like plain text, xml, html , JSON over HTTP request.

Let us have a glance over XMLHttpRequest supporting properties and methods and its usage in short.

Properties:



Methods:



In order to make HTTP request, you need to create XMLHttpRequest Object as mentioned below:

var XHR;

if (window.ActiveXObject) {

//For Microsoft Browsers

XHR=new ActiveXObject("Microsoft.XMLHTTP");

}

else if (window.XMLHttpRequest){

//For Mozilla and Non Microsoft Browsers

XHR=new XMLHttpRequest();

}

After making successful request, you will receive a respective response object.

You need to invoke needed business logic which has to be handled by this Response Object.

//binding anonymous function

XHR.onreadystatechange = function()

{

// business logic has to be implemented

}

As part of business logic implementation, you need to check the state and status of XHR Object.

//binding anonymous function

XHR.onreadystatechange = function()

{

// checking the state and status of XHR object

if( XHR.readyState == 4 && XHR.status == 200 )

{

// on success of XHR, code the successful implementation in this block

}

}

After checking the status of XHR Object, you can access the data by using either of below mentioned two options:

XHR.responseText

XHR.responseXML

Let us modify our code snippet accordingly:

// binding anonymous function

XHR.onreadystatechange = function()

{

// checking the state and status of XHR object, on success adding

content to respective DOM element

if( XHR.readyState == 4 && XHR.status == 200 )

{

document.getElementById(“<<element-name>").innerHTML= XHR.responseText;

}

}

To initiate the connection before implementing STEP 2 in your logic, you need to use open() and send() methods accordingly, in order to send back the exact output from Server to the Webpage from where this action has been triggered.

Enhance the code snippet as shown below:

// to establish the connection

XHR.open(method, url,async call);

// binding anonymous function

XHR.onreadystatechange = function()

{

// checking the state and status of XHR object, on success adding

content to respective DOM element

if( XHR.readyState == 4 && XHR.status == 200 )

{

document.getElementById(“<<element-name>").innerHTML= XHR.responseText;

}

}

// to send back the request to the server

XHR.send()

Point to be noted for open() are:

First parameter should represent appropriate method either GET or POST

Second parameter should represent valid url, consisting of any valid data format like text, JSON and so on. If user use JSON as data format then you need parse the values which you would receive through this call.

Third parameter you need pass Boolean value either true or false, by default it is true as it the code should execute Asynchronously.

To understand this concept in details, let us try to implement AJAX implementation for previous Doctor JSON Array Object code.

IIFE Nested Functions and Closures

Immediate Involving Function Expression

IIFE (Immediate Involving Function Expression) represents JavaScript function without any name, which would be executed as soon as interpreter runs that block.This concept can also be referred as self-executing anonymous function.

Developer can avoided variable hoisting concept, whereas interpreter provides public access to the declared method and internal maintains privacy also.

syntax:

(function() {

// zero or more than one executable statements along with return statement.

})();

Nested Functions

Developer can embed one function within another function. This concept can be termed as Nested Functions.Nested (Inner) Function can access arguments and variables of the Containing (Outer) Function.

Internally Nested Function would be private to their Containing Function. Due which developers cannot access directly the argument and variables of Inner Function. To access either of them, you need to invoke your logic with the help of Outer Function only and vice-versa Outer Function cannot access directly Inner Functions declared variable and arguments, to access them developer has to execute statements in the Containing Function.

Example:

function Outer(){

var num1=100

function Inner(){

var num2=200

return num1+num2

}

var x=Inner()

return x

}

console.log(Outer())

This concept plays an important role in creating user defined methods and Closures.

Closure

Is a concept through which developer can implement lexical scoping through functions.

Closure internal are built on below mentioned principles :

Inner/Nested functions

Function who can return another function.

In short you can consider Closure as a inner function which can outlive the lifetime of the outer function and hence can continue to access the variables of the outer function.

In the below example, the outer function is returning the inner function. Thus we see that we are able to invoke the inner function many times without having to invoke outer function each time.

Example :

function outer(){

console.log("Outer")

function inner(){

return "Inner”

}

return inner

}

private\_function=outer()

private\_function()

private\_function()

private\_function()

private\_function()

Object Oriented Concepts Object Literal and Constructor Pattern

Object Oriented Concepts

In JavaScript everything can be consider as Object. Many of the core concepts features like Arrays and Strings are some of the example of built-in Objects developer can use them to create to design a dynamic web application in easier fashion.

Developer can also create their custom objects based on their requirement, through which below mentioned concepts can be implemented programmatically, similar to other Object Oriented Programming languages like C++,PHP, Java etc…

Objects

Class

Constructor

Inheritance

Encapsulation

Abstraction

Polymorphism

Based on ECMA Script version implementation of OOC’s varies. For example Class concept can be implemented through Function in ES5 and with class keyword in ES6 and data hiding concept can be implemented partially through Function Scope in ES5 to fully extend but in ES6 user can restrict methods and variable access through Private, Public and Protected specifiers.

Objects plays very vital role in JavaScript programming.

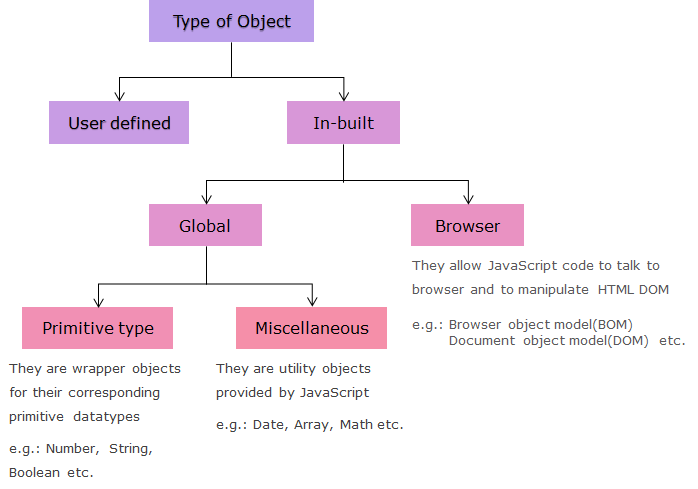
They can represent any real-world thing and it is constructs with properties and methods.

Developer has to instantiated the object before invoking them through code.

Once instantiated, the properties and methods of the object can be used accordingly.

HTML elements become objects in JavaScript and Attributes become properties in JavaScript.

JavaScript objects can be classified as shown below:



JavaScript Objects represents state and behavior in the form of collections of properties.

Where each property can be represented as key-value pair, where key holds a string or indicates any symbol and value can be any predefined JavaScript primitive data type or can also hold another user defined object.

Developer can create Objects using two approaches namely:

Object Literals or Singleton pattern

Constructor pattern

Object Literal or Singleton Pattern

This is very simplest way which can be adopted by developer to create user defined Objects.Object Literals are key value pairs always enclosed between {}.

For example, if you want to create an object representing Doctor entity having two properties like Name and Specialization can be achieved in ths way:

var doctor\_obj={

name:"Tom",

specialization:"ENT"

}

To access the properties or method of above declared doctor\_obj can be done by Dot operator (.) or Bracket [] operator has be as mentioned below:

doctor\_obj.name;

// or

doctor\_obj[“name”];

Point to be noted here is when you use doctor\_obj.name; or doctor\_obj[“name”]; this statement works as getter method but whereas if you doctor\_obj.name=“Jim”; or doctor\_obj[“name”]=“Jim”; this statement works as setter method.

Functions can be used to create methods in a object literal concept.

Behavior of an object can be represented through methods.

For example, the below code adds an availability behavior to the doctor object.

var doctor\_obj = {

name: "Tom",

specialization: "ENT",

availability: function() {

return“ From Morning 9 A.M - 11 A.M "}

}

console.log(doctor\_obj.name);

console.log(doctor\_obj.availability());

}

Expected Output:



this Keyword

It is one of the special keywords available in JavaScript. Which can holds a reference variable of the current object on which the method is being executed.

Using this keyword, developer can access the properties inside a method.

For example If you have requirement of accessing name property of previously created Doctor object in availability() then you can updated the code snippet as mentioned below, as you have added this.name your requirement would be achieved easily.

var doctor\_obj = {

name: "Tom",

specialization: "ENT",

availability: function() {

return this.name + " " + "From Morning 9 A.M to 11 A.M"

}

}

console.log(doctor\_obj.name);

console.log(doctor\_obj.availability());

Expected Output:

Image

As you have added this.name in availability() method you can access the name property easily

Please note that if you use this keyword in Outer function then properties of Outer functions cannot be accessed in Inner function also.

Constructor Pattern

You can also create Objects using Constructor Pattern.To use this approach developer need to use new keyword. It allows user to create custom Object through JavaScript built-in Object class as well as customized implementation.

For example let us create Doctor object using new keyword with the help of JavaScript built-in Object class.

// Doctor object creation using Object class.

doctor1 = new Object();

//To add dynamically properties

doctor1.name="Tom";

doctor1.specialization="E.N.T";

//code to access the declared property

console.log(doctor1.name);

console.log(doctor1.specialization);

Expected Output:



Now let us create a customize Doctor object to implement same concept.

//customized Doctor object declaration through function

function Doctor(name,specialization)

{

this.name=name;

this.specialization=specialization;

this.availability=function() { return this.name +" " + "From Morning 9 A.M - 11 A.M"}

}

// Doctor object creation.

doctor1 = new Doctor("Tom","ENT");

//code to access the declared property

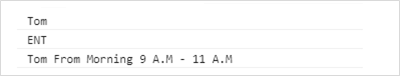
console.log(doctor1.name);

console.log(doctor1.specialization);

//code to invoke method through user created object

console.log(doctor1.availability());

Expected Output:



Using ECMA Script 5 user can create customized object only through function, as JavaScript doesn’t have class concept but from ECMA Script 6 and above this implementation can be directly achieved through class keyword.

Prototype

In JavaScript all the Object are always linked to another JavaScript Object. This linked Object is called as prototype Object. Thus every JavaScript Objects inherits the properties from its associated prototype Object.

Functions are also Objects and they also have prototypes. We can find the prototype of a function by using the prototype property.

All Objects created through a Constructor Function also inherits from the prototype object of respective associated constructor (super/parent) function.

For example, in the below code we are accessing the prototype property of the Doctor constructor function as well as for adding dynamically hospital attribute to same constructor.

function Doctor(name, specialization) {

this.name = name,

this.specialization = specialization,

this.availability = function() {

return this.name + "From Morning 9 A.M - 11 A.M"

}

}

// Doctor object creation.

doctor1 = new Doctor("Tom", "ENT");

//code to access the prototype property

console.log(Doctor.prototype);

//code to add hospital attribute dynamically through prototype object

Doctor.prototype.hospitalLocation = "Chennai";

console.log(Doctor.prototype.hospitalLocation);

Expected Output:



Please note that dynamically added property like hospitalLocation would hold static value, which can be accessed by all the instances of the Doctor class.

As seen above by using Prototype Object developer can add new attributes , delete attributes and even define new method without modifying the declared structure.

For example modify previous code and understand this concept.

function Doctor()

{}

// code to add new attributes and method

Doctor.prototype.name="Tom";

Doctor.prototype.specialization="ENT";

Doctor.prototype.availability = function() { return this.name +" " + "From Morning 9 A.M - 11 A.M"}

// Doctor object creation.

doctor1 = new Doctor();

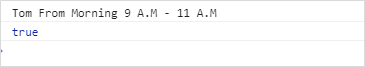
//code to invoke availability method

console.log(doctor1.availability());

// code to delete the attribute

console.log(delete doctor1.name);

Expected Output:



Prototype Pattern

Developer can inherit from one constructor to another constructor through Prototype Pattern.To implement inheritance among two constructors, the prototype object of one constructor function should be assigned to the prototype object of another constructor function.

For example let us establish inheritance between SpecializedDoctor(Child) and Doctor(Parent) through this pattern.

function Doctor(name, specialization) {

this.name = name,

this.specialization = specialization,

this.availability = function() {

return this.name + " " + "From Morning 9 A.M - 11 A.M"

}

}

function SpecializedDoctor(hospitalLocation) {

this.hospitalLocation = hospitalLocation;

}

//code for establishing inheritance between SpecializedDoctor and Doctor

SpecializedDoctor.prototype = Doctor.prototype;

//code to create child object and invoking its parent function

sDoctor1 = new SpecializedDoctor("ABC Hospital");

console.log(sDoctor1.hospitalLocation);

console.log(sDoctor1);

Expected Output:



To invoke Doctor(Parent) constructor in SpecializedDoctor(Child) constructor to set default value, call() method should be invoked in SpecializedDoctor(Child) constructor as shown below.

function Doctor(name, specialization) {

this.name = name;

this.specialization = specialization;

this.availability = function() {

return this.name + " " + "From Morning 9 A.M - 11 A.M"

}

}

function SpecializedDoctor(name, specialization, hospitalLocation) {

//code to invoke parent class

Doctor.call(this, name, specialization);

this.hospitalLocation = hospitalLocation;

}

//code for establishing inheritance between SpecializedDoctor and Doctor

SpecializedDoctor.prototype = Doctor.prototype;

//code to create child object and invoking its parent function

sDoctor1 = new SpecializedDoctor("Jim", "Dental", "ABC Hospital");

console.log(sDoctor1.name);

console.log(sDoctor1.availability());

Expected Output:

Image

Encapsulation

Encapsulation in JavaScript can be achieved through var keyword.

By using var with any attribute declared in function, scope of that variable/attribute would be private to that declared block. To access these private scope attributes developer needs to write accessor (getters) and mutator (setters) methods.

Now let us modify previous defined Doctor function, and make name attribute as private and create getter and setter for the same attribute.

Now you would have understood the Object Oriented concepts, Object Literal Notation, Constructor patterns and how to declare them, let us implement this learning to complete coding phase of Requirement 5: Blood bank module of Infy HMS application.

function Doctor(name) {

//private attribute declaration

var doctorName = name;

//getter method declaration

this.getName = function() {

return doctorName;

}

//setter method declaration

this.setName = function(name) {

this.doctorName = name;

}

this.availability = function() {

return this.doctorName + " " + "From Morning 9 A.M - 11 A.M"

}

}

// Doctor object creation and invoking getter and setter methods.

doctor1 = new Doctor("Tom");

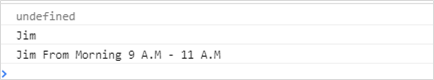
console.log(doctor1.getName());

doctor1.setName("Jim");

console.log(doctor1.getName());

console.log(doctor1.availability());

Expected Output:



Now you would have understood the Object Oriented concepts, let us implement this learning to complete coding phase of Requirement 5: Blood bank Module of Infy HMS application.

Asynchronous Execution Async Patterns Promises and Callback

What is Asynchronous Execution?

JavaScript is consider as Event-based scripting language, internally it means code written by developer executes now and some code would execute later.

Let us understand this point by analyzing two different scenarios.

Scenario 1: For example when you use console.log() statement as mentioned below:

<script type="text/javascript">

function testExecution()

{

console.log("Asynchronous Execution");

}

</script>

To populate the message which has been mentioned in console.log(), you need bind or invoke above mentioned testExecution() in your webpage.

Let us load this function on page load as shown below:

</head>

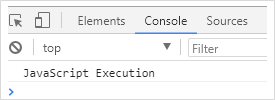
<body onLoad="testExecution()">

<h1> JavaScript Execution</h1>

</body>

</html>

Expected Output:



In order to execute simple console.log() as shown in below code snippet, you could observe that function holding this statement has been configured to JavaScript event handling method like onLoad() of body tag.

From this scenario, you should have understood that in JavaScript coding :

No code runs:

Unless until an event gets triggered.

That event should be mapped to at-least to one of JavaScript registered event handler methods.

That event should be first one in JavaScript Engine, then it would be executed or else the one in queue would execute accordingly.

In the above mentioned demo let us modify as mentioned below and have a glance over output:

<script type="text/javascript">

function testExecution()

{

console.log("JavaScript Execution");

console.log("it works");

console.log("in this fashion");

}

</script>

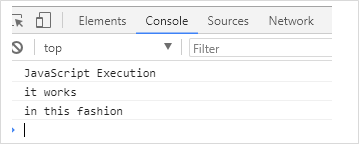
</head>

<body onLoad="testExecution()">

<h1> JavaScript Execution</h1>

</body>

Expected Output:



Now you would have observed that console.log() statements got executed sequentially.

Scenario 2: To understand the JavaScript Runtime environment, Let us have a glance over below mentioned code snippet and observe the output .

<script type="text/javascript">

function func1(arg1) {

console.log(new Error().stack);

}

function func2(arg2) {

func1(arg2 + 1);

}

function func3(arg3) {

func2(arg3 + 1);

}

func3(3);

//return;

</script>

Expected Output:



Now let us analysis above output:

Initially when JavaScript engine starts executing above mentioned code snippet, default internal call stack would be empty.

After its invoke the line func3(3) in above code, stack would have single entry i.e..,

Location in global scope

When the func2(arg3+1) is encountered by engine, stack gets appended with one new entries i.e..,

Location in func3

Location in global scope

When the func1(arg2+1) is encountered by engine, stack gets appended with one more new entries i.e..,

Location in func2

Location in func3

Location in global scope.

And due to “console.log(new Error().stack);” line of func1(arg1) method, Stack trace would be populated on screen as shown above expected output screenshot.

After the result is populated on console, automatically JavaScript engine terminates individually invoked function in order of execution and would remove the top available entries of the newly created Stack and at last Stack would be become empty and automatically the program would terminated.

If you analysis JavaScript Runtime architecture in detail, the above mentioned execution happened due to its basic nature of this technology which named as Single threaded event loop architecture.

What is Single Thread Event Loop Architecture ?

JavaScript environment :

Executes only one thread at a time, internally it executes a single piece of code associated with that current executing method or function or chunk.

By default concurrency is supported, due to this feature race condition is automatically handled by the system.

If there is any shared resource among two event handlers, automatically system executes only one event handler at run time.

Internally JavaScript engine will

Maintains a queue – task queue, in which all the event which are associated with event handler business logic are placed.

Pulls the events from queue, based on user action.

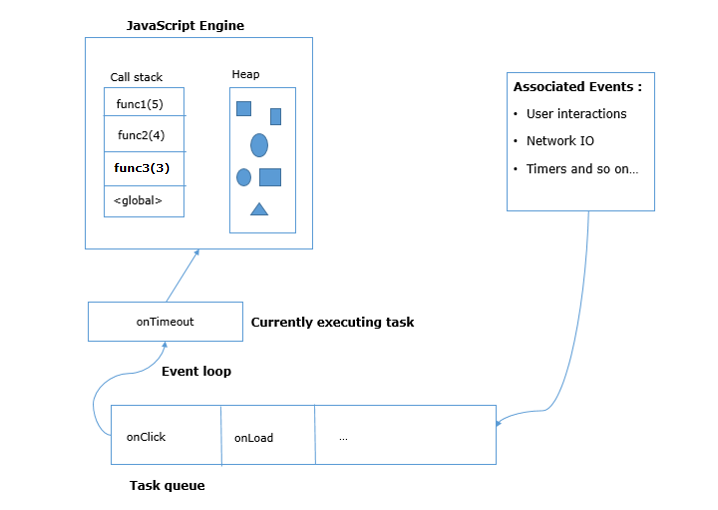
Retrieves the event handler logic

Invoke them sequentially based on their queue position

Execute the respective associated logic and process the immediate event of the queue.

Please note that JavaScript container can pipeline events to the existing queue like mouse click, Network IO process, Timers, Interrupts and so on..

Now let us have a glance over Scenario 2 in pictorial representation:



Now you should have understand the Single Threaded Event loop architecture. Let us go ahead and learn how to add events into JavaScript queue and different pattern to implement asynchronous execution.

Appending Events to Queue

User can append the Task Queue with new events by using setTimeout().

setTimeout()

Is a method of the WindowOrWorkerGlobalScope mixin (which is successor to window.setTimeout)

Helps in associating a timer with a single piece of code, which would be automatically executed once timer interval/delay expires.

Internally this method adds a timeout event to the Task Queue with the associated event handler.

Different ways to use this method

var setTimeoutID = scope.setTimeout(function[, delay, param1, param2, ...]);

var setTimeoutID = scope.setTimeout(function[, delay]);

var setTimeoutID = scope.setTimeout(code[, delay]);

Parameters which can be used are :

function: Business logic which has to be executed after timer interval expires.

code: Is an alternate syntax which allows developer to pass the string parameter instead of mentioning the associated event handler logic, this piece of string executes once timer interval expires.

delay: It is optional parameter, where time in milliseconds has to be specified and timer would wait for specified time before business logic starts executing through engine.

param1, param2 .. paramN : These are additional parameters which can be passed to timer, developer may or may not use this parameter.

Please note that code parameter is not recommended to be used as internally it make uses of eval() and above mentioned additional parameter syntax may not work with IE 9 version, in order to make the code work you need to use respective support polyfill in our business logic.

Return value: Is a positive integer value, this value can be cleared by passing to clearInterval() method which indeed would cancel the timeout event. This value would be always unique one even though its called on same object multiple time.

Let us understand setTimeout() implementation through an implementation.

Consider below given code snippet:

console.log("Before For loop execution");

for (var i = 0; i < 2; i++) {

console.log("setTimeout message");

func1();

func2();

}

console.log("After For loop execution");

function func1() {

console.log("Am in func1");

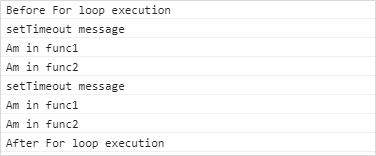
}

function func2() {

console.log("Am in func2");

}

According to JavaScript sequentially execution nature output populates as shown below:



Let us modify previous code by adding setTimeout() method in for loop as shown below and observe the output once again.

Modified code snippet:

for (var i = 0; i < 2; i++) {

setTimeout(function() {

console.log("setTimeout message");

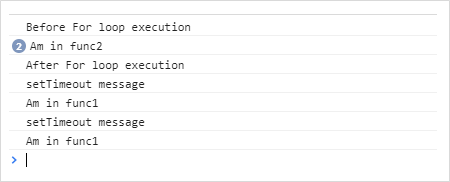
func1();

}, );

func2();

}

New Output:



When you have glance over above shown output, due to usage of setTimeout() method the entire execution of code behavior has been changed, internally parallel/multi-threading concept has been invoked.

Let us achieve asynchronous execution through setInterval() as shown below:

console.log("Before For loop execution");

var counter = 0;

var t;

t = setInterval(function() {

console.log("setTimeout message");

func1();

counter += 1;

if (counter >= 3) {

clearInterval(t);

}

}, 1000);

func2();

console.log("After For loop execution");

function func1() {

console.log("Am in func1");

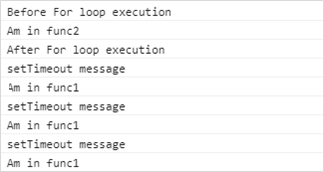
}

function func2() {

console.log("Am in func2");

}

Expected output:



By observing above output, you could find that setInterval() works similar to setTimeout() method, but the difference is setInterval() is business logic gets executed multiple times and to kill the timer you need to invoke clearTimer().

Please note that both of these methods are the only native functions to implement asynchronous execution in JavaScript environment.

Some of the real-time situation where you may use these methods are:

To make any HTTP request call.

To perform any Input/Output operations.

When you deal with Client and Server communication.

Let us learn about Asynchronous pattern and its implementation in detail.

Please note that setInterval() works similar to setTimeout() method, but the difference is setInterval() is business logic gets executed multiple times and to kill the timer you need to invoke clearTimer().

Asynchronous Patterns

Asynchronous execution in JavaScript can also be achieved through below mentioned patterns/techniques:

Callbacks

Promises

Callbacks

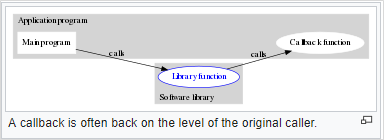
Is the traditional approach for handling asynchronous event in JavaScript.

It is a simple business logic/method, which can be registered with the built in JavaScript event handler for any event which you want to trigger some logic based on your code invocation.

Wikipedia defines Callback as:

Is any executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at a given time. This execution may be immediate as in a synchronous callback, or it might happen at a later time as in an asynchronous callback.

Pictorial representation:



Let us understand Callback through an implementation through AJAX demo, where you need to fetch the details from a contact.xml file.

function xmlCall(url, callback) {

var httpRequest; // create our XMLHttpRequest object

if (window.XMLHttpRequest) {

httpRequest = new XMLHttpRequest();

} else if (window.ActiveXObject) {

httpRequest = new

ActiveXObject("Microsoft.XMLHTTP");

}

httpRequest.onreadystatechange = function() {

if (httpRequest.readyState === 4 &&

httpRequest.status === 200) {

callback.call(httpRequest.responseXML);

}

};

httpRequest.open('GET', url);

httpRequest.send();

}

// call the function

xmlCall("contact.xml", function() {

console.log(this);

});

console.log("this will run before the above callback");

Expected Output:



In above mentioned code snippet, XML content has been populated by asynchronous AJAX call through httpRequest object and to finally call ends through call() method invoked by user defined callback method.

Now you would have understand Callback concept, let us go ahead and learn next Pattern i.e.., Promises.

Promises

The two major disadvantages of Callbacks are:

Concurrency management

Lack of trustability and sequentially

To overcome these disadvantages Promises concept has been introduced.Most of newly introduced Async APIs are internally built using this Pattern.This concept has been introduced from 2012 as a part of ES6 version.

In short , Promise represents eventual object result (completion or failure) of an asynchronous code.

Syntax:

new Promise( /\* executor \*/ function(resolve, reject) { ... } );

Parameters:

executor:

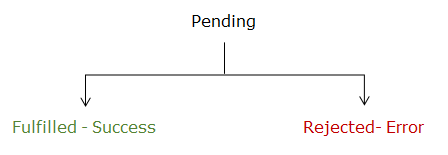
Is a function, which accepts two arguments namely resolve and reject

It gets executed before the Promise constructor invoked the code and would return object.

Internally this object would either hold success (resolve) or error (reject) value.

Resolve and Reject arguments basically represent asynchronous business logic code, which should be automatically triggered on success or failure value respectively.

Three different Stages of Promise object are:



To associate any event handler to Promise result you need to use appropriate thenable objects.

Valid thenable objects are:

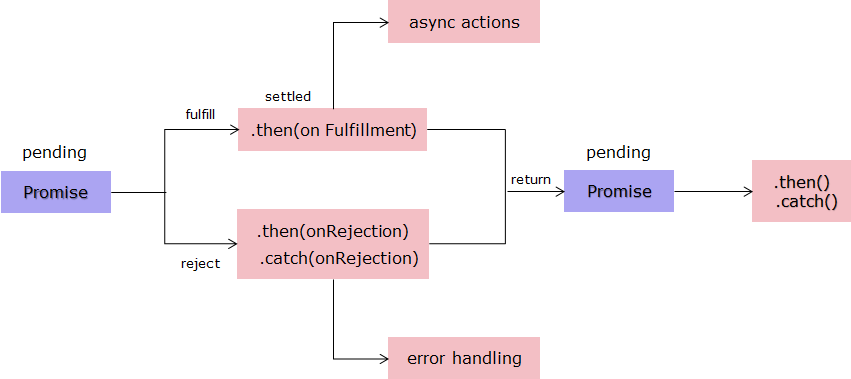
then()

catch()

Promise.all()

Promise.race()

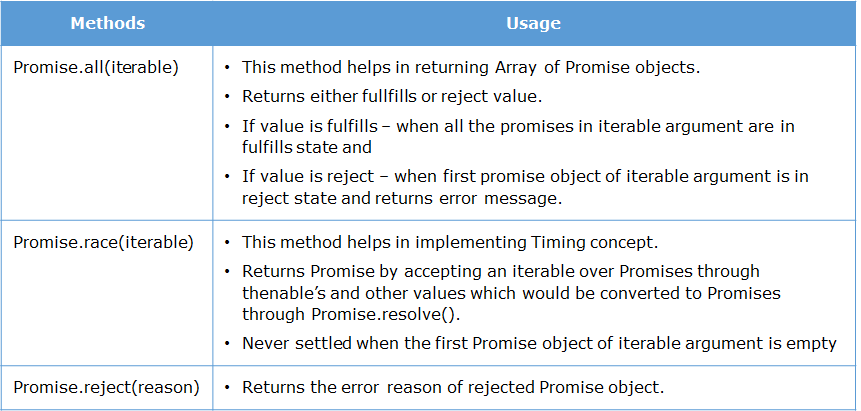
Pictorial representation of Promise execution:

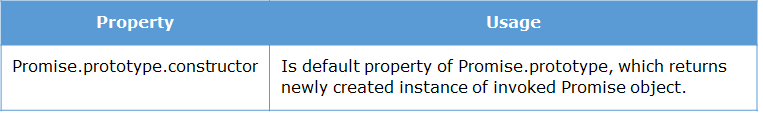


Source: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>

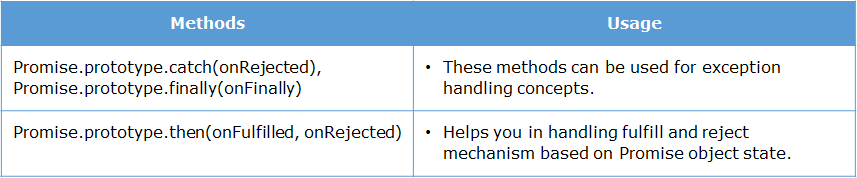
Let us have a glance over various Promise method in detail now.

Property:





Methods:



To understand this concept in details, let us try to enhance our previous callback demo with Promise pattern.

function xmlCall() {

var promise = new Promise(function(resolve, reject) {

var httpRequest;

if (window.XMLHttpRequest) {

httpRequest = new XMLHttpRequest();

} else if (window.ActiveXObject) {

httpRequest = new

ActiveXObject("Microsoft.XMLHTTP");

}

var url = "contact.xml";

httpRequest.onreadystatechange = function() {

if (httpRequest.readyState == 4) {

if (httpRequest.status == 200) {

resolve(httpRequest.responseXML);

} else

reject(httpRequest.statusXML);

}

}

httpRequest.open("GET", url);

httpRequest.send(null);

});

promise.then(function(response) {

console.log(response);

}, function(error) {

alert(error);

});

}

Expected Output:



Please note that Promise object has been created, then() has been used to display content of contact.xml file. If file is not available browser would populate alert dialog box and you will get observer error message in console.

Now you would have understood the Asynchronous execution in JavaScript, techniques and patterns, let us implement this learning to and enhance Requirement 3: Associated Doctor’s Profile module of Infy HMS application

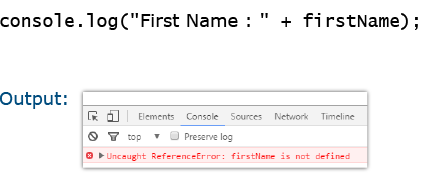
HOISTING

JavaScript interpreter follows the process called hoisting.

Hoisting means all the variable and function declarations wherever they are present throughout our program, get lifted and declared to the top of the program, in case of global scope, and on top of the function declaration in case of function/local scope.

Well, remember, only the declaration and not the initialization gets hoisted to the top.

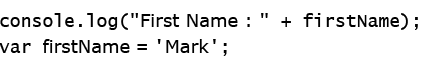
Example: Trying to access the variable without declaration:



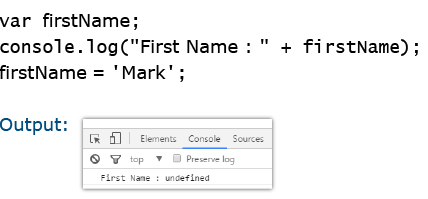
As expected, the interpreter is unable to find declaration of the variable ‘firstName’ anywhere inside the code.

Thus, reference error is thrown.

Let us now declare and initialize the variable in the code but after it is accessed.



Because of hoisting, the code is interpreted like this by the interpreter.

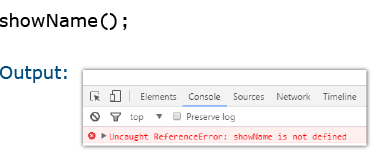


Hoisting here helps interpreter to find the declaration at the top of the program and thus reference error goes away.

But interpreter says that variable is not defined. This is because hoisting only lifted the variable declaration on the top and not initialization.

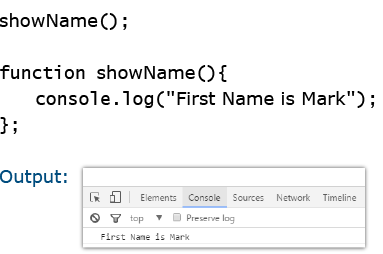
Let us now understand the function hoisting with an example:

First let us try to invoke the function showName() without declaring it in the program



As expected, the interpreter is unable to refer to any such function in the code.

Let us now declare and initialize the function but after the statement which uses it.



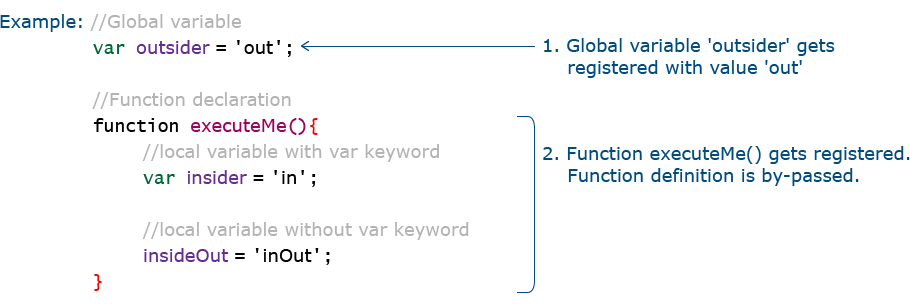
Well, the reference error goes away and interpreter now executes the function.

Here, function hoisting helped interpreter find the declaration of the function at the top.

Scope Resolution

When we write some lines of code in JavaScript, JavaScript engine interprets our code line by line and resolves the scopes of variables and functions using the lexical scoping. This means that the JavaScript code executes as per the scopes that we define in the code.

Let us understand the line by line execution considering the below example.



Step 3 is invocation of executeMe(). When invoked, the lines written inside function execute one by one as shown below:

