JavaScript (ES5)

From Zero to Hero

NAVEEN SAGGAM 4/18/18 UiBrains.com

JavaScript with ES (5)

Contents

JavaScript with ES (5)

JavaScript with ES (5)

What is JavaScript?

JavaScript is a lightweight, interpreted, object oriented language with first class functions and it is more popular as a scripting language for webpages and also used in many non-browser environments.

Lightweight

JavaScript does not require any large kind of setup like any other languages like java.

In order to execute a simple java program, we need to do the following,

Download and install JDK, setting class path, writing a java program, compile it and run it.

When comes to JavaScript we can directly write a JavaScript in a file and execute it directly without any compilation process, and even JavaScript occupies small amount of memory an easy to implement it. Because of this we call JavaScript is a light weight programming language.

Interpreted Language

In JavaScript the program will be executed directly which does not require any compilation process and no generation of intermediate files to execute the java program.

As Java and C++ are compiled and Interpreted programming languages, means we needs to write a java program and compiles to a class file and that has to be executed.

When it comes to JavaScript, the instructions will be executed directly.

Note:

Technically there is a compilation process happens which Is totally different than any other object oriented language. We will discuss about this compilation concept JavaScript in the later sections.

Object Oriented

JavaScript is an object oriented programming language.

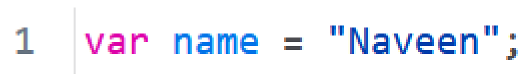
An Object orientation is a concept where in order to solve any problem, instead of writing some set of instructions to be executed, we actually provide a solution by breaking down the problem into the real world objects.

JavaScript is also an object oriented programming language which uses real world objects to solve any problems such as any other object oriented programming language does.

First Class Functions

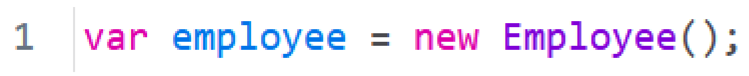
This is one of the concept of functional programming wherein we can use functions as values and also we can use functions as an arguments to any other methods just like values of a variable.

For example we can assign a string values to a variable as



```
var name = "Naveen";
```

We can also assign a object to a variable as



```
var employee = new Employee();
```

The same w can use a function itself as a value and assign it to a variable.

Note:

It clearly state that we are not going to execute a function and assigning the return value to a variable, instead we will assign the whole function itself to a variable.

We can also use functions itself as a parameters to any other function.

This is one of the major feature of functional programming language.

As JavaScript is also actually originated from the family of 'C' Language which is having this feature already in it.

Scripting Language

A scripting language is basically a language where set of instructions are written for a runtime environment.

The best example for scripting language is "shell" scripting.

In the Linux or UNIX platform we can execute various commands targeted to any runtime environments like 'listing' a directory and copying files from one place to other.

Instead of executing each command, we can actually write those commands one by one and bundled into a script file called .sh file. And that file will be executed with UNIX or Linux runtime.

These instructions / commands will be executed by Linux server directly.

The same way in JavaScript we will be writing a set of instructions and make it to a '.js' file and this will be executed by the target runtime typically a web browser.

Due to this JavaScript is called as scripting language.

There are different runtime environments for JavaScript along with the web browser is a server.

We can even execute a JavaScript in server like environments.

Ex: Node JS, Express JS

Note: Node JS is a native JavaScript which is used to execute the JavaScript in the server environments.

Express Js is a framework built on top of node js and also used for server side scripting.

Both Node JS and express JS are server side scripting languages, whereas native JavaScript, JQuery are browser side scripting languages.

Weakly Typed Language

A Weakly typed language is a language where the strict type checking is not performed before executing a program.

Basically the 'Java' language is a strongly typed language were the type checking of each and every variable and proper syntaxes before the compilation of a program.

JavaScript is actually the most forgiving language and suits to the basic level of programmers in mind.

Whereas java is actually meant for professional programmers where they need to know each of the rule of the language.

Strictly Type Language

Basically java, C++ and other programming languages are strictly typed or strongly typed languages where each and every variable method, class are strictly checked by the compiler.

There are actually a lot of strict rules to be followed in order to write and compile a program in java.

Where in JavaScript this strictly typing is not required, and it is a weakly typed language.

Note: due to this weakly typed feature of JavaScript there are some wired design designs are made in order to provide backward compatibility of the language.

Understanding Runtime Environment for JavaScript

A Typical runtime environment for JavaScript a browser.

Browser send on http requires to a server and server send back an http response to the browser.

This http response from the server is actually bunch of text which contains a set of HTML tags with data.

Browser gets the HTML code and builds a tree like structure called a DOM Tree.

Diagram:

After building a DOM Tree, it renders the page and display the rendered page on the browser.

Whenever we load a html on the browser, we get some sort of view on the browser.

If we load the same html again and again we still get the same vie and which will not be changed.

As the HTML is a static language, whenever we loads and any number of times the view will be same.

When the browser loads html code along with JavaScript, the view may get change.

As JavaScript is a dynamic programming language is having an ability to change the DOM tree of a browser.

By using JavaScript we can

1. Add a new node to a tree
2. Remove a node from a tree.
3. Edit a node from a tree

Because of all these modifications, browser builds a new Dom tree or refactor the DOM tree and render the page view.

As the browser refactor the DOM tree, obviously we will get another view and static view as with the HTML.

Why Learn JavaScript

By using JavaScript we can develop a rich client side applications.

In any web application the client side web development can be done using JavaScript.

There are some other areas of usage of JavaScript as follows,

1. Client Side Applications.
2. Server side Applications.
3. Browser extensions.
4. Desktop applications
5. Mobile Applications
6. IOT Applications.

Client Side Applications

As JavaScript is most popular in client side web development.

JavaScript is the world most popular Browser's we scripting language.as of today.

By learning JavaScript we can develop a rich client side applications.

There are some other frameworks or libraries we can use to develop client applications along with native JavaScript.

Once we learn JavaScript, It will be easy for us to learn and understand the following technologies such as JQuery, Angular Js React JS Ember and Backbone JS.

As Angular is nowadays most polar JavaScript frame work to build rich client side applications and SPA (Single Page Application).

With JavaScript knowledge we can be able to learn and understand AngularJS better way.

Server side Development

JavaScript also gained popularity in server side environments as well.

Ex: node js, Express JS.

By using Node JS and Express JS we can develop server side logic development. Express JS is a Library of Node JS.

Browser Extensions:

There are lot of browser extensions available in the market are actually built using JavaScript.

Example:

Add Blocker for Chrome

Eye Dropper.

Desktop Applications:

By using JavaScript we can even develop some of the desktop applications using JavaScript.

Example: Web Torrent Desktop

WordPress Desktop Application

Pexels Desktop Application.

We can use some of the tools like 'electron' which is a software development platform created by GitHub that lets developers to use JavaScript along with HTML, CSS to create Desktop applications..

Mobile Applications:

By using JS we can even develop some of the mobile applications using JavaScript and other tools.

Popular tools for developing mobile apps using JS are

Phone GAP / Cordova, Titanium / JQuery Mobile / Meteor

IOT Apps

JavaScript is the Top programming language for building IOT Apps.

IOT stands for Internet of Things.

IOT Is the network of physical devices, vehicles and other items embedded with electronics, software sensors with network connectivity to collect and exchange the data.

Example:

By using IOT apps we can control our home things like washing Machine, TV or A/C from anywhere in the world.

Interesting Thoughts of JavaScript:

Duglas Crockford is one of the famous person in JavaScript landscape stated that

JavaScript is the world's most misunderstood programming language

There is actually a reason behind this statement as there are some unique features of JavaScript misunderstanding concepts which leads the developer gets frustrated because we would not expect the things works in JavaScript and some of the unexpected behavior of JavaScript.

Example: This keyword, __proto__

Actually this is one survey happened in the early days of JavaScript on the concept of

The most loved language and the most hated language

After the survey the results announced and surprisingly "JavaScript" is top most among the most loved language and the most hated language list.

Why the JavaScript became the most hated language list to most of the developers is that due to lack of understanding of actual concepts and behavior of JavaScript.

Actually People / Developers from other server side language like Java, may think like JavaScript is same as Java and behaves like Java.

Actually JavaScript is syntax wise same as Java, and but there are some concepts which are not behaves as we expected. And there also some of the features which are unique to JavaScript.

Ex: IIFE, This keyword

In this initial days of development we may feel we are aware of JavaScript but once we started writing some serious JavaScript code then the developer comes to know the importance of knowing concepts of JavaScript.

History of JavaScript:

JavaScript is created by Brendan Eich in Netscape during mid 90's while the browser war between internet explorer and Netscape navigator.

While Brendon was with Netscape, there Java which already popular in those days with the concept of Applets which provide rich client Interface those days.

Initially JS is named live script but people wanted to use the popularity of Java , Later renamed it to JavaScript.

JavaScript is just to complement Java, it is named as JavaScript and not having any relation with Java.

Brendon created JavaScript around span of 10 days and rushed to Production.

JS is developed new guys in mind and it is a weakly typed and most forgiving language.

Again due to the competition which Microsoft internet Explorer, JS people wanted to standardize it.

Once it is standardize it will become a specification and all the browsers has to follow the specification.

Js is standardized by a committee called "ECMA" and it called as ECMA Script.

There are various version of EXMA Script and the latest version of ECMA is ECMA 8.

But it is not fully supported by all the modern browsers. The only supported version as of today is ECMA 5.

We will be learning ES5 version of JavaScript.

Setting Up Development Environment:

For any language development environment is required. For JavaScript the runtime environment is a web browser.

Once we install any browser that is enough with the development environment.

Example:

Mozilla, Chrome, IE, Opera, Safari.

Here in this course we will be using Mozilla.

Steps:

Download and install Mozilla Firefox Browser.

Open the browser window

Click on Developer Tools and Scratchpad.

1) Toggle Tools -> Console Tab
2) Web Tools -> Console Tab
3) Scratch Pad

Almost all the browsers comes with an inbuilt JavaScript engine and some tools to write code using JavaScript.

Example:

```
console.log("Hello JavaScript");
```

Console -> Global Object

Log -> function of Global Object

Hello JavaScript -> the String to be printed on the console.

We can also use Scratch pad to write code.

Steps:

- ➔ Open a Scratchpad.
- ➔ Write JavaScript code.
- ➔ Right Click and click on Reload and Run
- ➔ Code will be executed by the browser.
- ➔ Logs will be printed on the console.

Variables and Declaration:

In all the other languages there all certain variable and which holds some values.

Any variables acts like a container and which holds some value of it.

The variable declaration JavaScript as follows,

Example:

```
var num1 = 10;
```

var -> keyword to declare a variable

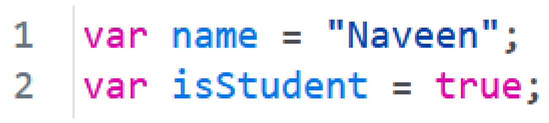
num1 -> name of the variable

10 -> value of the variable num1.

Note:

In JavaScript we will not specify any type of variable like Java. Any variable can be declared using "var" keyword.

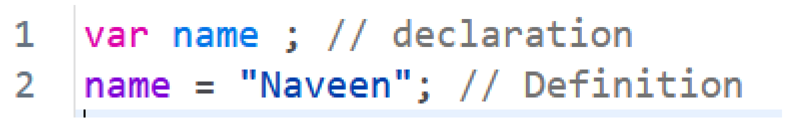
Example:



```
var name = "Naveen";
var isStudent = true;
```

JavaScript does not have a concept of variable types.

We can break declaration into two likes also.



```
var name ; // declaration
name = "Naveen"; // Definition
```

In JavaScript there is no scoping related keywords like public or private like JAVA.

Every variable which is declared is always a public in JavaScript.

JavaScript is having the following primitive types

1) Number
2) String
3) Boolean
4) Undefined
5) Null

Number:

Number is a JS Primitive type for storing numbers.

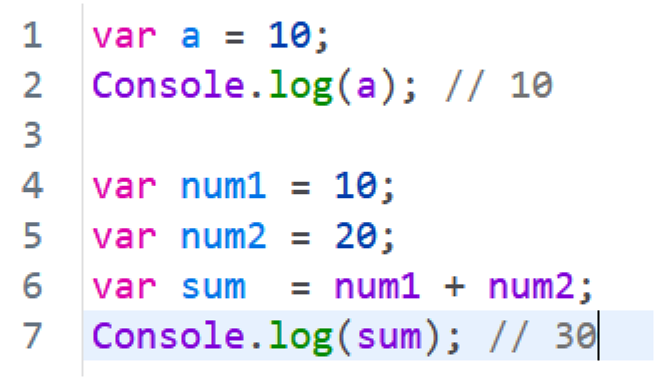
Number in JS are "Double Precision 64 bit format IEE 754 values".

It means in JS we don't have integers as such. All the Numbers which are declared are floating points Numbers only.

As in Java We might have byte, short, int, long, float double types for declaring numbers.

But in JS we have the only type called Number to declare any type of numbers.

Example:



```
var a = 10;
Console.log(a); // 10

var num1 = 10;
var num2 = 20;
var sum  = num1 + num2;
Console.log(sum); // 30
```

Note:

We can define Integer Values we can be able to see the integer values as result. But JavaScript Actually maintain them as a 64bit floating point numbers.

String:

String in JavaScript is a collection of characters.

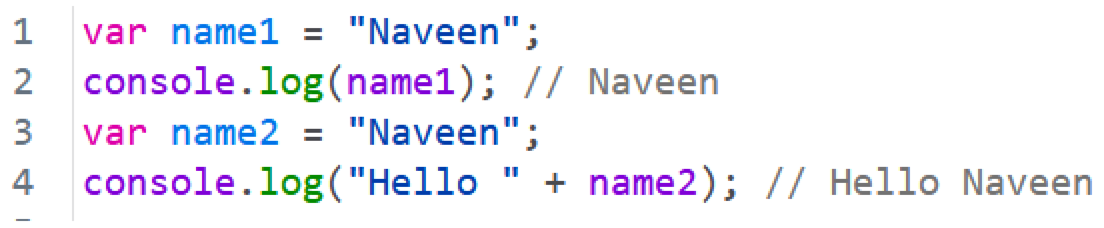
String is a collection of 16 bit Unicode characters.

In JavaScript there no character datatype. And each character is just a string with the length of ONE.

Note:

In JavaScript any text values which is assigned to a variable is of type "String" only.

Example:



```
var name1 = "Naveen";
console.log(name1); // Naveen
var name2 = "Naveen";
console.log("Hello " + name2); // Hello Naveen
```

Boolean

Boolean is a JS primitive datatype to represent true / false values.

This Boolean datatype are can use in any conditional statements.

```
var isStudent  = true;
console.log(isStudent);
```

JS Typing Interchange

In JavaScript the variable type are interchange means, once we declare a variable pf pne type we can assign the value of another type.

Example:

```
var a = 10; // a is a Number
a = "Hello"; // a is String
a = true; // a is Boolean
```

In any other languages like Java, once a variable is declared of one type it cannot be interchangeable or transferable.

If we are trying to assign some other type of value to a variable we may get compile time error.

But in JavaScript we won't get any compile time error and type interchange is also possible.

Because JavaScript is a weakly types language and there is no strict type checking is performed and which is more forgiving language.

Note: In JavaScript a variable can contain any values in any point of the entire JavaScript Program.

Undefined:

This is a unique datatype which JS available only in JS and not available in any other language.

In order to understand the "undefined" datatype we should know the difference between "declaration" and "Definition" of variables.

Declaration:

Just declare a variable without any assignment is called declaration of a variable.

```
var name; // Declaration
```

Definition:

If assign some value to a declared variable we call it as variable definition.

```
name = "Naveen"; // Definition
```

"undefined" is a value which comes between declaration and definition.

```
var name; // undefined
name = "Naveen"; // Naveen
```

Before we assign a value to a variable, till then the value of that variable is "undefined".

Once we assign a value to the variable then the value "undefined" will be replaced with the assigned value.

As in Java, once we declare a variable then some default variable will be assigned based on the type of the variable.

But in JavaScript there is no concept of type declaration at the time of declaring a variable. So for any kind of variable declaration a default value "undefined" will be assigned to it.

```
var firstName = 'Naveen';
console.log(firstName); // Naveen
```

Understanding Null:

As in JavaScript at the time of a variable declaration we have default value as "undefined".

In order to assign a value to a variable after the declaration we can assign any value to that variable.

In JavaScript there is a provision to add some dummy value to the variable is called "null".

This "null" is a value which is assigned to any type of variable in JavaScript.

Example:

```
var a;
a = null;
console.log(a); // null
```

Difference between Undefined & Null:

Undefined is value which is assigned at the time of declaration step and null is a value which is assigned at the time of definition step.

The default value before assignment of null or any value is "Undefined" and at the time of assignment there is an assignable dummy value which is null.

Example:

```
var name;
console.log(name); // undefined

name = null;
console.log(name); // null

name = 'naveen';
console.log(name); // naveen
```

Note:

For example let's take an application form where we need to fill in.

For any of the field we did not douched or filled the default value assigned to it is "undefined".

For any of the field which we add some dummy value as "Not Applicable" then the filed gets the default as null.

Summary of Variables & Types

At the time of declaration and definition of a variable we no need to declare the variable the type in JavaScript.

The same variable which can be assigned with any other types of values.

No Scoping information is available for variable declaration and definition.

JavaScript Basics

Here we will learn some of the basic concepts of JavaScript.

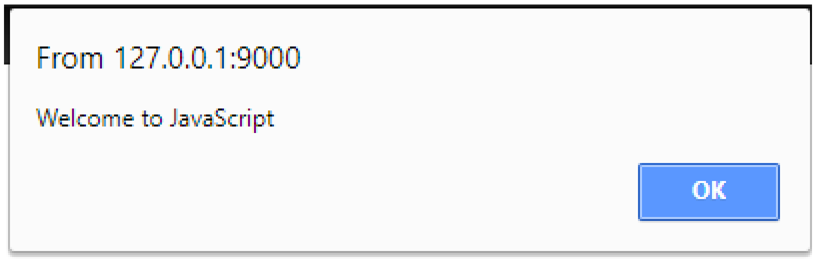
If we needs to execute any JavaScript code we don't required any HTML code base. We can just open the browser console and execute the following commands.

For logging information in JavaScript we have the following methods.

1) Alert Box
2) Confirm Box
3) Console log

By Using alert we can show any textual information on the alert box.

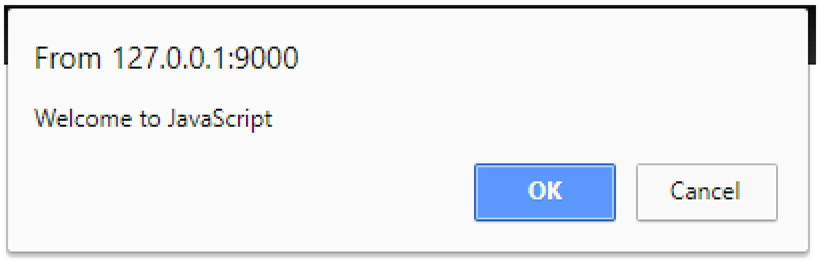
```
alert("Welcome to JavaScript");
```


From 127.0.0.1:9000

Welcome to JavaScript

By Using Confirm box also we can show some information on the webpage

```
confirm("Welcome to JavaScript");
```


From 127.0.0.1:9000

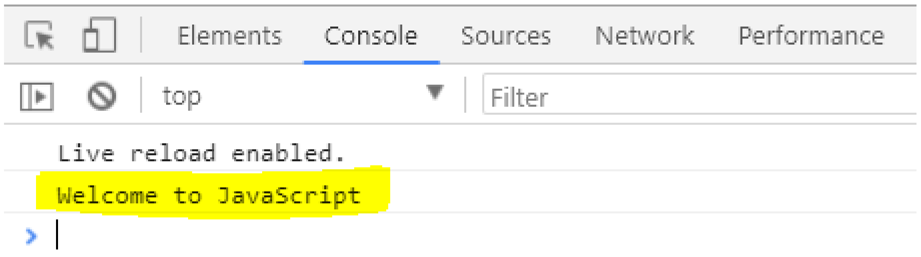
Welcome to JavaScript

The Alert and Confirm boxes are displaying on the webpage. Each time we may have to click on ok button for each and every message on the box.

If we wants to add some logging related information, then alert and confirm boxes are not recommended.

For logging related information we may use console.log method of JavaScript. This prints the messages on the browser console instead of on the webpage as follows,

```
console.log("Welcome to JavaScript");
```



```
Live reload enabled.
Welcome to JavaScript
```

Console.log is the best way of printing any log related information, this prints the information on the browser console.

Along with the textual information we can display any kind of information on the console using Console.log method/function.

Let's discuss how to display the current date information on the webpage.

```
var date = new Date();
console.log("Today is " + date);
```

JavaScript Usage

We can use the JavaScript along with the HTML in the following ways,

1) Inline Way
2) Internal Way
3) External Way

Let's discuss about each of the way of using the JavaScript.

Inline Way

Here in this way we may mix the JavaScript code along with the HTML Code. However this approach is not recommended but we may use this approach for very few lines of code.

Example:

```
<div id="text-div">
    <p id="green-p">This is a Green Color Text</p>
    <button onclick="document.getElementById('green-p').style.color = 'blue'">Change to Blue</button>
    <button onclick="document.getElementById('green-p').style.color = 'red'">Change to Red</button>
</div>
```

Internal Way

In this way we may use separate <script> Tag and add the completed JavaScript code to it. This is one of the best way instead of mixing the JavaScript code with HTML codebase.

Example:

```
<div id="text-div">
    <p id="green-p">This is a Green Color Text</p>
    <button onclick="blue();">Change to Blue</button>
    <button onclick="red();">Change to Red</button>
</div>

<script>
    function blue() {
        document.getElementById('green-p').style.color = 'blue';
    }
    function red() {
        document.getElementById('green-p').style.color = 'red';
    }
</script>
```

External Way

This is the best way of adding JavaScript to HTML codebase. Here we will be using an external ".js" file, this separates the HTML codebase with JS code base.

```
// HTML File

<div id="text-div">
    <p id="green-p">This is a Green Color Text</p>
    <button onclick="blue();">Change to Blue</button>
    <button onclick="red();">Change to Red</button>
</div>

// Separate JS file

function blue() {
    document.getElementById('green-p').style.color = 'blue';
}
function red() {
    document.getElementById('green-p').style.color = 'red';
}
```

Variable Declaration in JavaScript

For any programming language the variables are heart of Programming. A variable which holds some data on the computer system and we may process that data using variables in the complete logic of any application.

Let's discuss about the variable declaration in JavaScript,

We can declare any variable in JavaScript using a keyword called "var"

In JavaScript we may not mention the datatype information of the variable like any oops programming languages.

Here we just use "var" keyword and assign any type of value to the variable. Based on the type of value we assigned, the JavaScript will automatically get the type information.

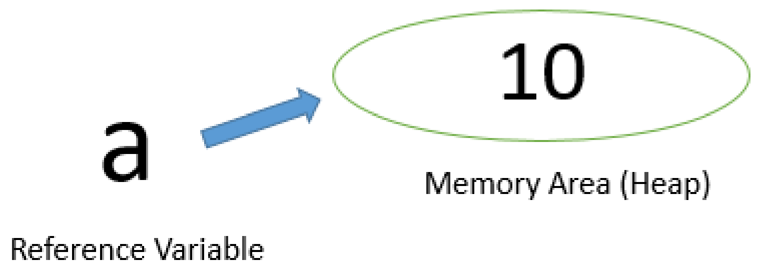
```
var a = 10;
var b = true;
var name = 'Naveen';

console.log(a);
console.log(b);
console.log(name);
```

Please note here we have not specified any datatype related information like string or number and Scoping related information like private, public.

Now deeply understand the variable declaration in JavaScript as follows,

When we declare a variable and assigned a value to it,

```
var a= 10;
```



When we declare a variable "a", "a" will be a reference variable which is pointing to a memory location (Heap Memory) which holds the value.

We may refer this value using the reference variable and use it for our mathematical calculations.

Data Types in JavaScript

In the variable declaration we can assign various types of values to the variables.

The type of value or the type of data which is assigned to a variable is called as datatype.

Let's discuss about the various datatypes available in JavaScript.

1) Number
2) String
3) Boolean
4) Undefined
5) Null

In JavaScript ECMA Script 5 version the above mentioned as the various datatypes are available.

Number Data Type

If we assign any numerical value or data to a variable is called as Number Datatype.

Normally in any other OOPS languages like Java, we may have various types of Numerical data like byte, short, int, long, float, and double based on the value of Numerical data.

But in JavaScript we do not have such kind of different types of types for Numerical data representation.

In JavaScript we can assign any Numerical data as a value to a variable, which can denoted with the datatype as Number.

Example

```
var a = 10;
var b = 20.5;
console.log(a); // 10
console.log(b); // 20.5
console.log(typeof a); // Number
console.log(typeof b); // Number
```

String Data Type

If we assign any textual value or data to a variable is called as String Datatype.

Normally in any language like Java, for any single character or letter they represent with a datatype called "Character" and any textual data which more than one letter or character they denote with the datatype called "String".

But in JavaScript there is no datatype called "Character", we can represent a single letter is also with the data type called "String".

Example:

```
var letter = 'a';
var name = 'Naveen';
console.log(letter); // a
console.log(name); // Naveen
console.log(typeof letter); // string
console.log(typeof name); // string
```

Boolean Data Type

If we assign any "true" or "false" value to a variable is called as Boolean Data Type.

Boolean data type contains only true or false value for any other language as well as in JavaScript.

Example:

```
var isEasy = true;
console.log(isEasy); // true
console.log(typeof isEasy); // boolean
```

Undefined Data type

This is one of the unique data type available only in JavaScript. If we just declare a variable and not assigned any value to it, then the default called "undefined" value will be assigned to the variable.

Example:

```
var a;
console.log(a); // undefined
console.log(typeof a); // undefined
```

Null Data Type

If we just declare a variable and not assigned any value to it, then the default value will be assigned to that variable is “undefined”. If we declare a variable and assign a dummy value instead of actual value is “null” datatype.

Example:

```
1  var a = null;
2  console.log(a); // null
3  console.log(typeof a); // Object
```

In the Above Example, at Line Number 1, we have assigned a dummy value “null” instead of some meaningful value.

At the Line Number 3, we got the type of ‘a’ as Object instead of null. In JavaScript whenever we assign a value null to a variable then we get the type of “null” will be Object instead of null. This is one of the wired behavior of JavaScript.

So please remember type of ‘Null’ will be always Object but not “null” in JavaScript.

Operators in JavaScript

In any programming language for kind of mathematical operations we do require various types of operators.

Using operators we can perform some most of the mathematical calculations for writing any processing logic of any system.

Let’s discuss about the Operators in JavaScript.

Assignment Operator (=)

In JavaScript we represent the assignment operator using equal symbol. (=)

The Assignment is used to assign values to any variables.

Example

```
var name = 'Naveen';
console.log(name);
```

Arithmetic Operator (+ , - , / , * , %)

The Arithmetic operators are used to perform some mathematical operations like addition, subtraction, division, multiplication and modulus/remainder.

Example

```
var a = 10;
var b = 20;

var sum = a + b;
console.log(sum); // 30

var sub = a - b;
console.log(sub); // -10

var mul = a * b;
console.log(mul); // 200

var div = a / b;
console.log(div); // 0.5

var mod = a % b;
console.log(mod); // 10
```

Short hand Math (+= , -= , *= , /= , %=)
The short hand math operators are used to apply shortcut mathematical operations.

If we wanted to add two values and assign to a variable we may use this short hand math operators.

```
1  var a = 10;
2  var b = 20;
3  var sum = 0;
4  sum = sum + (a + b);
5  console.log(sum); // 30
```

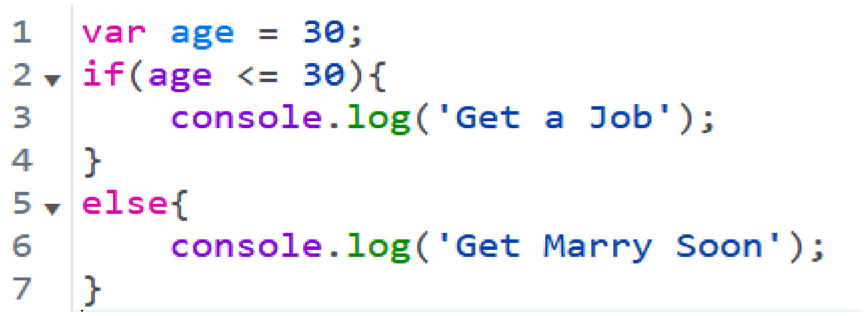
If our math operation is like the above mentioned line number 4, then for such cases we may this short hand math as follows,

```
1  var a = 10;
2  var b = 20;
3  var sum = 0;
4  sum += (a + b);
5  console.log(sum); // 30
```

Conditional Operators (< , > , <= , >= , !=)

The conditional operators are used to check logical conditions on various values of variables.

We normally use these conditional operators to evaluate some logical expressions.



```
var age = 30;
if(age <= 30){
    console.log('Get a Job');
}
else{
    console.log('Get Marry Soon');
}
```

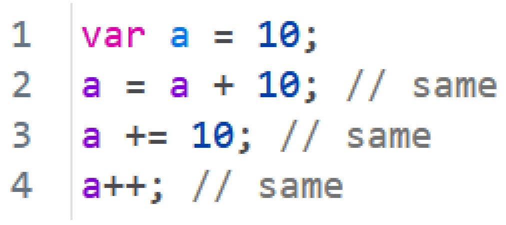
Here we are evaluating the logical expression , if the age is less than or equal to 30 , then "Get a Job" will be printed on the console, if not then "Get Marry Soon" will be printed on the console.

Unary Operator (++ , --)

All the above operators required more than one operand or variable to apply the operator, but for this it require only one variable is called unary operator.

The ++ , -- operator is also called as increment and decrement operators.

If our requirement is for incrementing or decrementing any variable value by '1' we use this operator.



```
1  var a = 10;
2  a = a + 10; // same
3  a += 10; // same
4  a++; // same
```

In the above example, at line number 2 , 3 , 4 all are same, means by using all these ways we can increment the 'a' value by 1.

For this kind of requirement the unary operator is the shortest way.

Logical Operator (&&, ||)

The logical operators are used to check some logical conditions.

&& specifies if both the values are true then only the result will be true;

|| specifies if aby one value is true then the result will be true;

Example

```
var courseCompleted = true;
var practiceCompleted = true;
if(courseCompleted && practiceCompleted){
    console.log('You will definitely get a Job');
}
else {
    console.log('Keep trying');
}
```

String Concatenation Operator (+)

The String Concatenation Operator is used to append two strings or a string and a number.

We normally use '+' symbol to add two numbers, the same operator is also acts like a String concatenation operator while we are trying to add a String to a number or both the strings.

Example:

```
var a = 10;
var b = '20';
var append = a + b;
console.log(append); // 1020

var str1 = 'Good';
var str2 = 'Morning';
var greeting = str1 + str2;
console.log(greeting); // Good Morning
```

Ternary Operator (? :)

This operator is used as an alternative to if-else condition.

The syntax of ternary operator is as follows,

```
(condition) ? trueSection : falseSection;
```

First we evaluate the condition, if it is true then the true Section will be executed and if the condition is false then the false section will be executed.

```
var age = 18;
var category ;
(age <= 18) ? category = 'Minor' : category = 'Major';
console.log(category); // Minor
```

Typeof Operator

This operator is used to check the type of a value is assigned to any variable.

Normally we use this operator to check the datatype of a value assigned to a variable.

Example

```
var a ;
console.log(typeof a); // undefined

a = 10;
console.log(typeof  a); // number

a = 'Naveen';
console.log(typeof a); // string

a = true;
console.log(typeof a); // boolean

a = null;
console.log(typeof a); // object
```

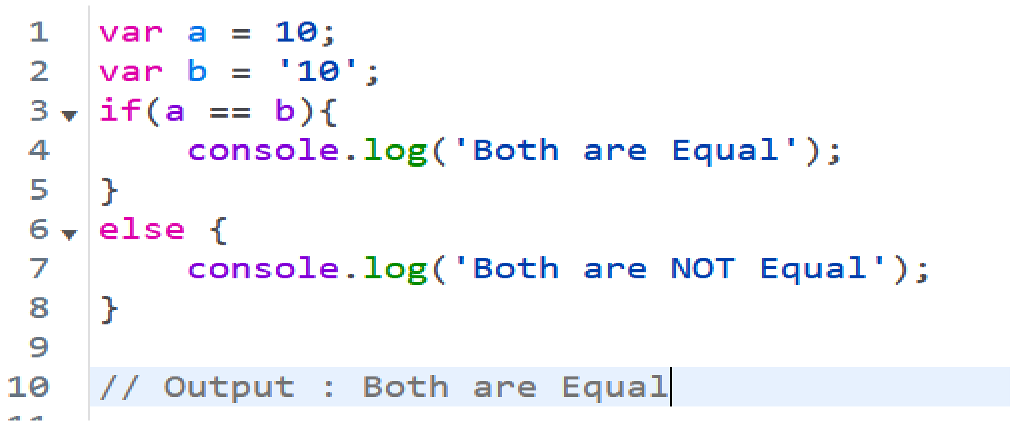
Note: If a variable is just declared and not assigned any value then the default value assigned to that variable is 'undefined'.

== Operator

In JavaScript the == operator is used to check the equality of two operands.

If we use this operator to check the equality of two operands first it will do the automatic type conversion and then apply the type conversion.

Example:



```
var a = 10;
var b = '10';
if(a == b){
    console.log('Both are Equal');
}
else {
    console.log('Both are NOT Equal');
}

// Output : Both are Equal
```

In the above example we have a number and String. The == operator first check for the type conversion possible or not and do the checks for the equality. Here it prints on the console as "Both are equal".

Even though the content is same but both are from different datatype. So normally if both are of different datatype forget about the content, they are always not equal only.

Note: we should normally check the equality of two same data typed values only.

Due to this wired behavior of automatic type conversion and checks for the equality , this approach of using == operator for use of equality is not elegant way.

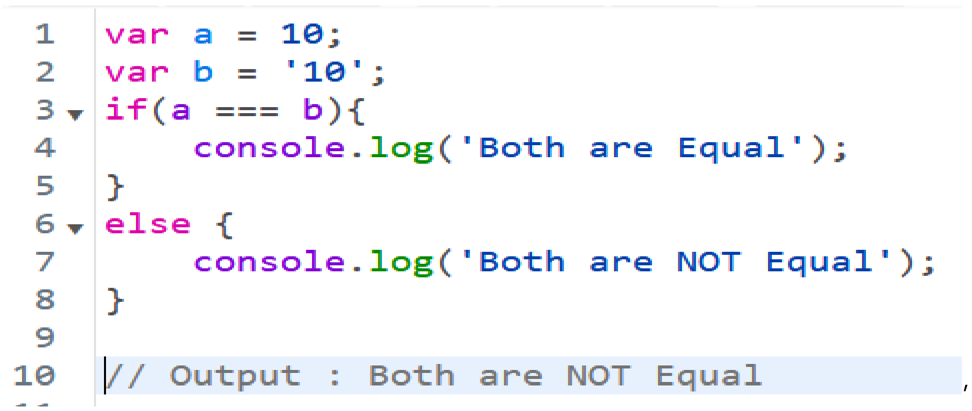
We can resolve this by using === operator.

=== operator

This is the best way of checking the equality two operands. Here there is NO automatic type conversion will be happened like == operator.

In JavaScript always use only this operator to check the equality of two operands.

Example:



```
var a = 10;
var b = '10';
if(a === b){
    console.log('Both are Equal');
}
else {
    console.log('Both are NOT Equal');
}

// Output : Both are NOT Equal
```

In the above example, it will print on the console as "Both are NOT equal".

Conditional Statements

In JavaScript we use various conditional statements to check some mathematical conditions to providing processing logic for any system.

The conditional statements / Loop statements are as follows,

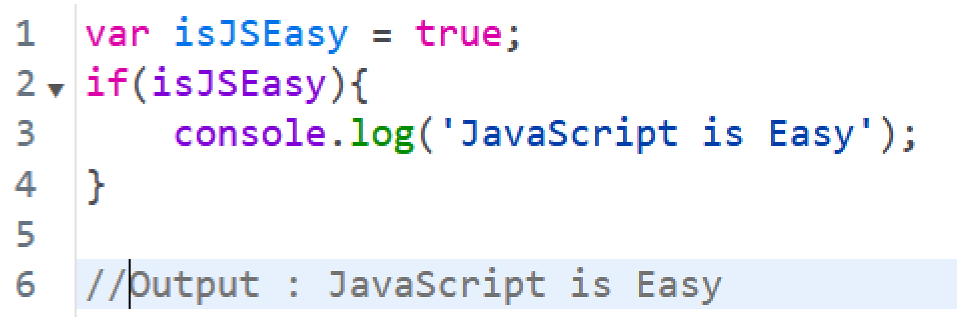
1) If Condition
2) If else Condition
3) For loop
4) While loop
5) Do while loop
6) Switch statement

Let's discuss about these conditional statements,

If Condition

The 'if' condition is used to evaluate the logical condition if any condition is matched or results to true.

Example:



```
var isJSEasy = true;
if(isJSEasy){
    console.log('JavaScript is Easy');
}

//Output : JavaScript is Easy
```

For have to specify the logical condition, if the condition is true then the block of code for 'if' statement will be executed.

This is just one way of checking the condition. If the condition is not met or if the condition is false then nothing will be done. So for this kind of cases we may use 'if-else' statements.

If-Else Condition

The 'if-else' is one of the best way checking for true part and false part also for checking the conditions for any processing logic.

If the condition is true then if condition will be executed and If the condition is false then the else statement will be executed.

Example

```
var a = 10;
var b = 20;
if(a > b){
    console.log('a is big');
}
else {
    console.log('b is big');
}

// Output : b is big
```

For Loop

The for loop is used for looping through some conditions, until the condition is true the loop will be continued, once the condition is false then the control comes out from the for loop.

The Syntax of for loop is,

```
for(initialization ; condition ; increment / decrement){
    // statements
}
```

Each for loop contains 3 sections as specified as above like initialization, condition and increment / decrement section.

Initialization section will be executed only once, and remaining sections will be executed until the condition is true.

Let's take an example of printing 1 – 10 values on the console using for loop.

```
for(var i = 1 ; i <=10; i++){
    console.log(i);
}

// Output :  1 2 3 4 5 6 7 8 9 10
```

While Loop

While loop is also a looping statement which loop through the condition and until the condition is true the block of while loop will be executed and once the condition is false then only the control comes out from while loop.

Syntax:

```
// variable declaration
while(condition){
 // statements , increments / decrements
}
```

Let's see an example of printing 1 – 10 values using while loop.

Example:

```
var i = 1;
while(i <= 10){
    console.log(i);
    i++;
}

// Output :  1 2 3 4 5 6 7 8 9 10
```

Note: We should be very careful while using while loop, if we didn't add the increment or decrement statement the loop may fall into infinite loop.

Do While Loop

We will use do while loop is also for looping purpose. The difference between while and do while is this 'do while' loop will be executed at least once.

Syntax:

```
// variable creation
do{
    // statements or increment or decrement
}
while(condition);
```

Let's see an example of do while loop to print 1 – 10 values.

Example:

```
var i = 1;
do{
    console.log(i);
    i++;
}
while(i <= 10);

// Output :  1 2 3 4 5 6 7 8 9 10
```

Note: While using do while loop, we should always add increment or decrement of variable otherwise do while loop may fall into infinite loop.

Switch Statement

This is an alternative to if-else ladder statement. Instead of using many if-else conditions we may use this switch statement.

Syntax:

```
// variable declaration
switch(condition){
    case value:
        // statements
        break;
        .............
     default:
     // statements
     break;
}
```

In the switch statement once the condition is match then the respective case of statements will be executed. If none of the case matches then finally the 'default' case will be executed.

We should always use 'break;' statement for each of the 'case' otherwise all the cases will be executed until it reaches the break statements.

Example:

```
var day = new Date().getDay();
var actualDay = '';
switch(day){
    case 0:
        actualDay = 'Sunday';
        break;
    case 1:
        actualDay = 'Monday';
        break;
    case 2:
        actualDay = 'Tuesday';
        break;
    //............
    default :
        actualDay = 'Who Knows ?';
        break;
}
console.log(actualDay);

// Output :  Monday
```

Objects in JavaScript

JavaScript is an object oriented programming language. Here in JavaScript we will can provide solutions for any requirements in Objects oriented way instead of a functional programming.

The object orientation of JavaScript is different than any other Object Oriented programming languages. In any other languages we will a have a class based structure or blue print and creates Objects for that class and the complete data of the system is evolved in objects.

But here in JavaScript we do not have a concept called classes as of ES5 version.

We can create objects directly without classes.

Let's understand the ways of creating objects and adding properties to objects and deleting properties of objects and deleting properties of an Objects.

We will create objects in JavaScript in simple way as follows,

```
var obj = {};
console.log(obj);

// Output : Object { }
```

We can add properties to an objects even after creation objects at runtime as follows,

```
    var employee = {};
    employee.firstName = "Naveen";
    employee.lastName = "Saggam";
    employee.designation = "Team Lead";
    console.log(employee);

```

The Output will be printed on the console as follows

```
▸ Object { firstName: "Naveen", lastName: "Saggam", designation: "Team Lead" }
```

Here in the above example we have created an empty objects and added all the properties to it at runtime. We can also add the properties to an Object at the time of creation also. This is called as Object Literal.

Let's see the Objects Literal Example as below,

```
    var employee = {
        firstName : 'Naveen',
        lastName : 'Saggam',
        Designation : 'TeamLead'
    };
    console.log(employee);
```

Now let's understand the ways of retrieving properties of an object.

We can access the properties of an Objects using two ways as follows,

1) Using Dot Notation
2) Using Brackets Notation.

Using Dot Notation

We normally access the properties of an object using dot notation. The syntax of Dot notation is as follows,

```
    ObjectName.propertyName;
```

Example:

```
console.log(employee.firstName); // Naveen
console.log(employee.lastName); // Saggam
console.log(employee.designation); // Team Lead
```

Brackets Notation

We can access the Object's properties using brackets notation also.

The syntax of using brackets notation is as follows,

Syntax:

```
ObjectName["propertyName"];
```

Let's access the properties of employee object using brackets notation.

Example:

```
console.log(employee["firstName"]); // Naveen
console.log(employee["lastName"]); // Saggam
console.log(employee["designation"]); // Team Lead
```

Note:

We normally use this brackets notation to retrieve invalid property names an objects like any property name starts with number. But invalid property names are not allowed inside an objects.

We use this brackets notation to access the properties of an Array. This we will discuss in Arrays concept.

We also use this brackets notation to access dynamic values of an objects , for accessing dynamic values we can't use the dot notation.

Let's see the example of access dynamic values of an Object.

```
var student = {
    name : 'John',
    age : 20,
    course : "Computers"
};
var studentName = "name";
console.log(student.studentName); // undefined
console.log(student[studentName]); // John
```

Note: In order to access the properties of an Object using Dot notation we must use the same name as the property name which is exists in the object. We cannot access the dynamic values using Dot notation. We can access the dynamic values of an objects using Brackets notation.

We have discussed about accessing the existing properties of an object. But what happens we are trying to access the properties which are not existing in an objects.

Normally for any other OOPS languages we may get some error. But here in JavaScript we won't get any compile time error but it simply returns "undefined".

Let's see the example of accessing non existing properties of an Object as follows,

```
var student = {
    name : 'John',
    age : 20,
    course : "Computers"
};
console.log(student.address); // undefined
```

Note: In the above Student object there is no property called "address", so we trying to access the non-existing property of an object in JavaScript we will get "undefined".

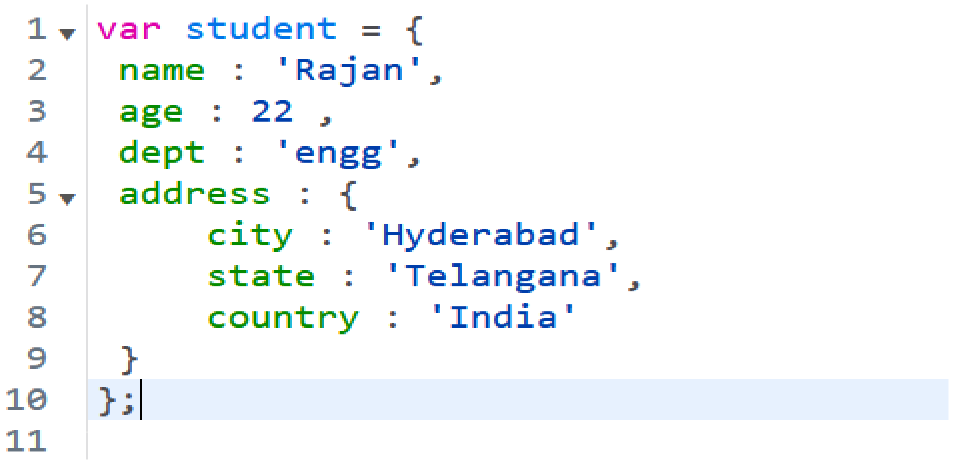
We can even create objects inside other objects is called as nested Objects.

Let's understand the Nested Objects and the way of access the properties of nested objects.

Nested Objects

A JavaScript object which is defined in side another JavaScript object is called as Nested Object. This is a very common scenario we may in almost all OOPS programming languages.

Example:

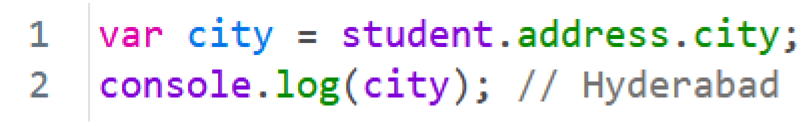


```
var student = {
 name : 'Rajan',
 age : 22 ,
 dept : 'engg',
 address : {
     city : 'Hyderabad',
     state : 'Telangana',
     country : 'India'
 }
};
```

In the above Example we have a student Object and inside we have another object called 'address'.

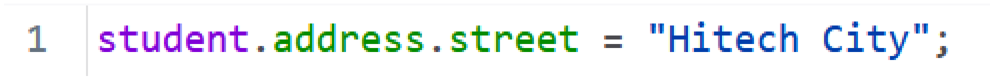
We can access the nested objects properties also just like any other property of an object.

Let's access the city property of nested object 'address' which is inside of student object.



```
var city = student.address.city;
console.log(city); // Hyderabad
```

We can even add some few extra properties to the nested object 'address' like the normal way of adding properties to an object.

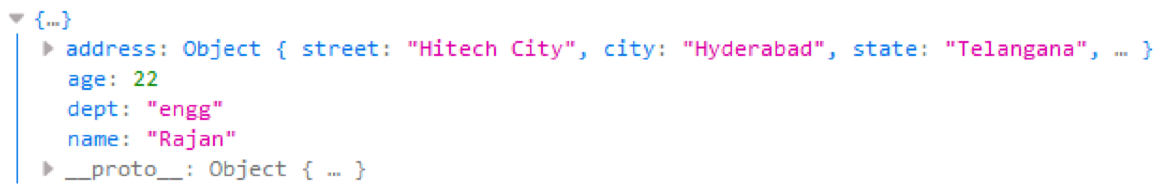


```
student.address.street = "Hitech City";
```

We can do the console log the entire student object to see whether the 'street' property is added to 'address' object or not as follows,

```
console.log(student);
```

The output is as follows,



```
▼ {…}
  ▶ address: Object { street: "Hitech City", city: "Hyderabad", state: "Telangana", … }
    age: 22
    dept: "engg"
    name: "Rajan"
  ▶ __proto__: Object { … }
```

Arrays in JavaScript

An Array is a very common concept available in almost all the programming languages.

An Array in JavaScript is an indexed collection of values of various datatypes.

Array is a 'zero' index based in all the programming languages.

Let's discuss the declaration of array and adding properties to an array and accessing the properties / elements of an array and also the default functions available to various operations with arrays.

Creation of arrays:

```
// creation of array
    var myArray = [10,20,30,40];

// Accessing an array and its properties
    console.log(myArray[0]); // 10
    console.log(myArray[3]); // 40
```

We can add properties to an array using based on the index of the array only.

```
myArray[4] = 50; // added new property
console.log(myArray);
```

We can access the length of the array using 'length' property of an array.

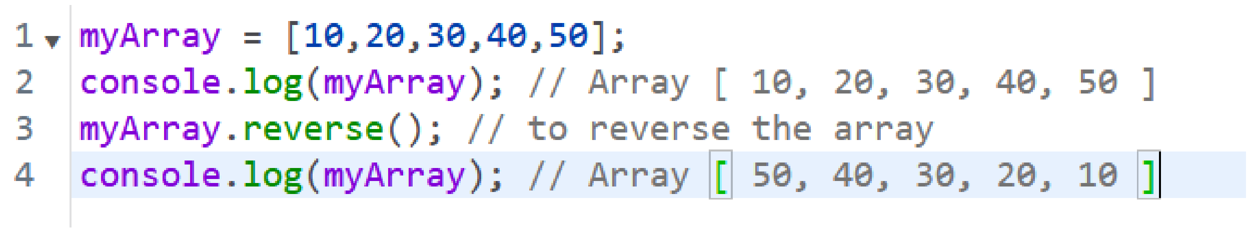
```
console.log(myArray.length); // 5
```

Let's discuss some of the default available function to perform various operations on elements of an array.

reverse ()

This method / function is used to reverse the elements of an array.

The usage of reverse function is as follows,

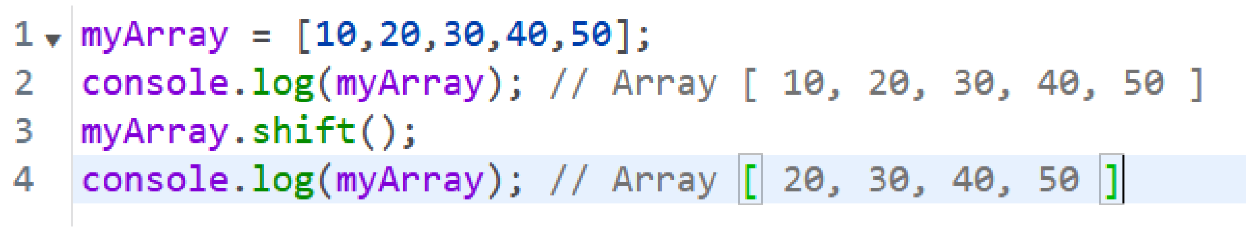


```
myArray = [10,20,30,40,50];
console.log(myArray); // Array [ 10, 20, 30, 40, 50 ]
myArray.reverse(); // to reverse the array
console.log(myArray); // Array [ 50, 40, 30, 20, 10 ]
```

Shift()

This method / function is used to remove the first element of an array.

The usage of shift function is as follows,

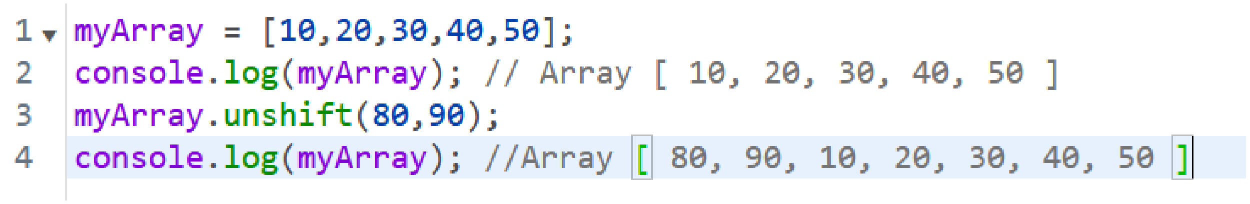


```
myArray = [10,20,30,40,50];
console.log(myArray); // Array [ 10, 20, 30, 40, 50 ]
myArray.shift();
console.log(myArray); // Array [ 20, 30, 40, 50 ]
```

Unshift()

This method / function is used to add a specific element (or elements) to the front of an array.

The usage of unshift() is as follows,



```
myArray = [10,20,30,40,50];
console.log(myArray); // Array [ 10, 20, 30, 40, 50 ]
myArray.unshift(80,90);
console.log(myArray); //Array [ 80, 90, 10, 20, 30, 40, 50 ]
```

Pop()

This method / function is used to remove the last element of an array.

The usage of pop() is as follows,

```
myArray = [10,20,30,40,50];
console.log(myArray); // Array [ 10, 20, 30, 40, 50 ]
myArray.pop();
console.log(myArray); // Array [ 10, 20, 30, 40 ]
```

Push()

This method / function is used to add an element (or elements) to the end of an array.

The usage of push() is as follows,

```
myArray = [10,20,30,40,50];
console.log(myArray); // Array [ 10, 20, 30, 40, 50 ]
myArray.push(100,200);
console.log(myArray); // Array [ 10, 20, 30, 40, 50, 100, 200 ]
```

Splice()

This method / function is used to remove an element from an array from the specified index position.

Syntax : splice(index,n);

Here 'index' means the index of the removable element and 'n' specifies the number of elements should be deleted from that index position.

The usage of splice() is as follows

```
myArray = [10,20,30,40,50];
console.log(myArray); //Array [ 10, 20, 30, 40, 50 ]
myArray.splice(1,3); // removes 20 , 30 , 40
console.log(myArray); // Array [ 10, 50 ]
```

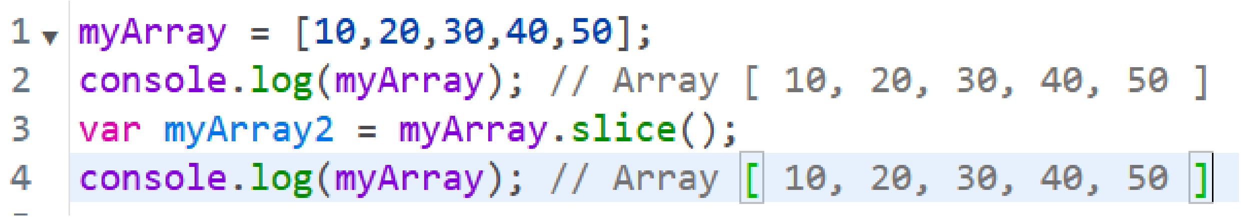
Let's see the example which removes a single element from an array.

```
myArray = [10,20,30,40,50];
console.log(myArray); // Array [ 10, 20, 30, 40, 50 ]
myArray.splice(1,1); // removes 20
console.log(myArray); // Array [ 10, 30, 40, 50 ]
```

Slice()

This method / function is used to create a new copy of an existing array and typically assigned to a new variable.

The usage of slice() is as follows,



```
myArray = [10,20,30,40,50];
console.log(myArray); // Array [ 10, 20, 30, 40, 50 ]
var myArray2 = myArray.slice();
console.log(myArray); // Array [ 10, 20, 30, 40, 50 ]
```

Indexof ()

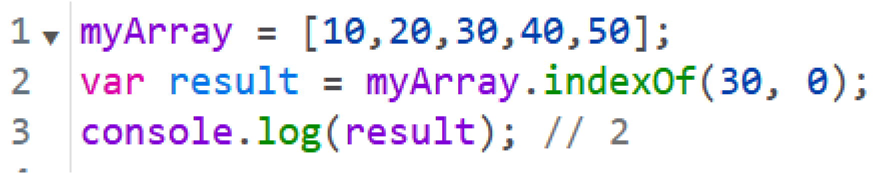
This method / function returns the index of any element. Here we can even specify the

Index position also in this method, this returns the first element that matches with that

Specified index position.

Syntax: myArray.indexof (search, index);

The usage of Indexof () is as follows,

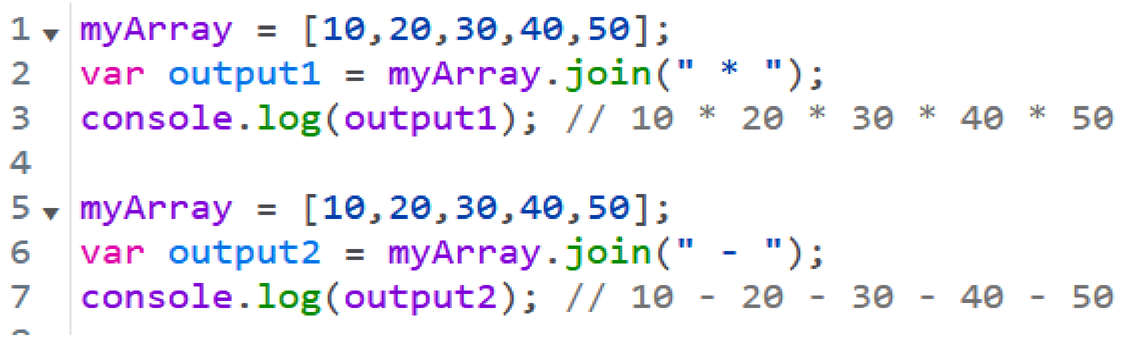


```
myArray = [10,20,30,40,50];
var result = myArray.indexOf(30, 0);
console.log(result); // 2
```

Join()

This method / function is used to return a string representation of all the array elements joined with the specified parameter.

The usage of join() is as follows,



```
myArray = [10,20,30,40,50];
var output1 = myArray.join(" * ");
console.log(output1); // 10 * 20 * 30 * 40 * 50

myArray = [10,20,30,40,50];
var output2 = myArray.join(" - ");
console.log(output2); // 10 - 20 - 30 - 40 - 50
```

Note: we do have lot of methods / functions available for array, Please have a look at the below URL for complete information about an Array of JavaScript.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Functions in JavaScript

For any programming language functions / Methods place a major role in logic development.

A Function is a block of code which takes in various arguments and internally does some processing and finally returns the result.

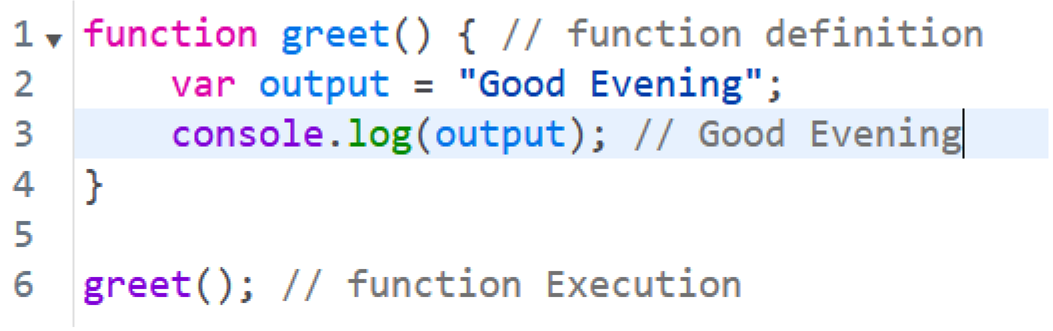
A function in any programming language is treated as an engine, which takes in some parameters and does some processing and finally returns the result.

One function calls other function and gets the results from other function does some more process based on the result got from the previous function. This is called as function chain.

In order to build any software system we do require functions for processing logic and objects for holding the data of system, functions uses these objects data and proceeds for more processing of any system.

Let's understand the functions in JavaScript.

We can create a simple function in JavaScript with no arguments as follows,



```
function greet() { // function definition
    var output = "Good Evening";
    console.log(output); // Good Evening
}

greet(); // function Execution
```

Note: For any function there are always two parts, one is function definition and second is the function execution.

Just by declaring the function definition which cannot be executed automatically, we needs to call / execute the function.

Now let's understand a creation of a simple function with arguments,

```
1  // function definition
2  function greetArgs(name) {
3      var output = "Good Evening " + name;
4      console.log(output); // Good Evening Naveen
5  }
6
7  // function Execution
8  greetArgs("Naveen");
```

Note: Here at line Number 8, we are passing "Naveen" parameter to the greetArgs() function and which takes in the name and prints it on the console.

Executing a function with less number of parameters

In any other programming languages other than JavaScript we must have to call / execute the function with the same number of parameters as mentioned in the function definition, otherwise we may get compile time error.

But in JavaScript we call a function with less number of arguments than the number of parameters specified in the function definition. In this case we won't get any error but simply the remaining parameter gets the value undefined.

Let's discuss the example as below,

```
function lessArgs(name,age) {
    var output = " Your name is : " + name + " age : " +age;
    console.log(output); //  Your name is : Naveen age : undefined
}

lessArgs("Naveen");
```

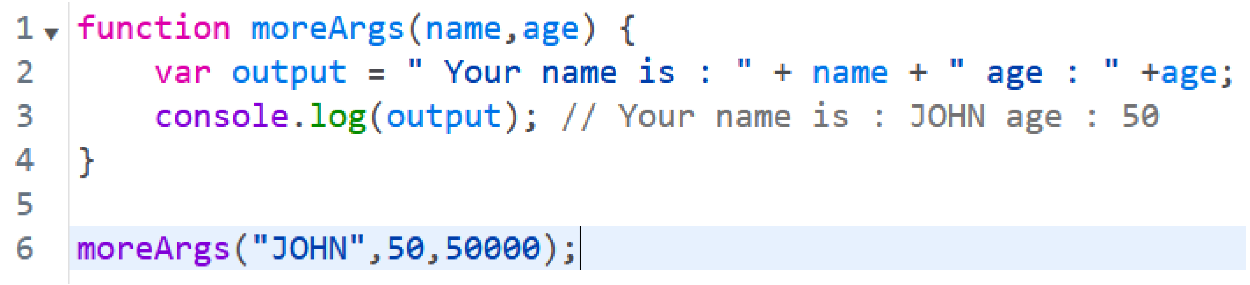
In the above example the function accepts two parameters like name, age. But while calling / executing the function we provided only one parameter 'name' then we wouldn't get any error and the age parameter get the value '*undefined*'.

Executing a function with more number of parameters

We can even execute a function by providing more number of parameters than the expected number of parameters of the function.

If we pass more number of parameters to a function than the expected number of parameters, then we won't any error in JavaScript and simply the extra parameters will be ignored by the function.

Let's discuss the example as below,



```
function moreArgs(name,age) {
    var output = " Your name is : " + name + " age : " +age;
    console.log(output); // Your name is : JOHN age : 50
}

moreArgs("JOHN",50,50000);
```

In the above example the moreArgs () function is expecting only two parameters like name, age. But while calling the function we are providing name, age, salary also.

In this case the salary parameter will be ignored by the function, and we won't get error in this case.

Note: Based on the previous scenarios a single function in JavaScript can accepts more or less number of parameters also. This implies that function overloading is not possible in JavaScript.

In any other OOPS languages a function can accepts the exact parameters as specified in the function definition. We can't pass more or less number of parameters. For this reason we may write the overloaded functions to accept the more or less number of parameters for the same function name.

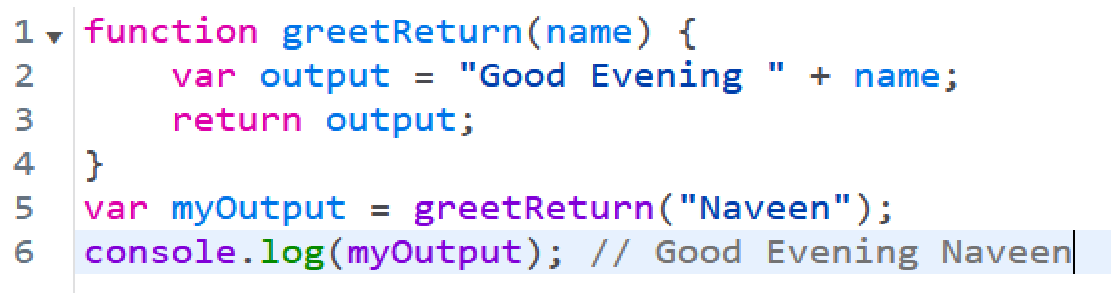
But this is not required in JavaScript. So it clearly states that function overloading is not possible in JavaScript.

In JavaScript function names are unique, we should not write more number of functions with the same name in same JavaScript file. Multiple function with same name is not allowed in JavaScript.

Functions with Return Values

As of now we have seen the function without return values. Now will understand the JavaScript functions with return values. Normally for any function in JavaScript we may not specify the return value type in the function declaration. We just return any type of value as a result.

Let's discuss the example as follows,



```
1  function greetReturn(name) {
2       var output = "Good Evening " + name;
3       return output;
4  }
5  var myOutput = greetReturn("Naveen");
6  console.log(myOutput); // Good Evening Naveen
```

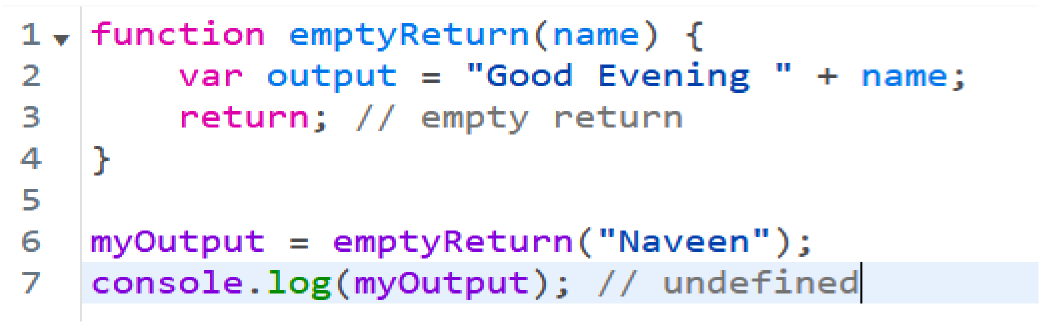
In the above example the greetRetun () function is returning ‘output’ as a result return value. We are capturing that value at line number 5 and printing it on the console.

If any function is returning with some value, then we must have to capture that return value and use it for printing to console or use it for further processing.

Function with Empty return

If any function is return with an empty value then result of it will be undefined.

Let’s discuss a function with empty return value,



```
function emptyReturn(name) {
     var output = "Good Evening " + name;
     return; // empty return
}

myOutput = emptyReturn("Naveen");
console.log(myOutput); // undefined
```

As specified in the above function if we are trying to capture the empty return, then the captured value contains the value *undefined*.

Function Expression

This is the feature of functional programming. Here we can assign the function itself as a value to a variable.

If we assign a function itself as a value to a variable is called as function Expression or First Class Function. JavaScript is contains full of first class functions.

Let’s discuss about the First Class Function / Function Expression as follows,

```
// function Expression
var greetMe = function greet() {
        var output = "Good Evening";
        console.log(output); // Good Evening
};

//function Execution
greetMe();
```

In the above example we have created a variable "*greetMe'* and assigned a function itself as a value to it. We can execute that function expression with the variable name only but not with the function name.

Anonymous Function Expression

Any function expression without a name is called as Anonymous function Expression.

In the previous example we have declared a function Expression and we have executed it with the variable name only. So here we never use the function name then function name is dummy.

So we can create a function expression without a function name as follows is called as Anonymous function expression.

```
// Anonymous Function Expression
var greetMe = function() {
    var output = "Good Morning Naveen";
    console.log(output); // Good Morning Naveen
};

// Function Execution
greetMe();
```

Anonymous Function Expression with Arguments

In the above example we discussed about anonymous function without arguments. Now let's discuss the anonymous function with arguments.

```
// Anonymous Function Expression with args
var anomymousFnArgs = function (name) {
    var output = "Good Evening " + name;
    console.log(output);
};

// Function Execution
anomymousFnArgs("Naveen");
```

Functions as Arguments without Arguments

In order to build a software system or product we should have to use various functions and share data between those functions.

Each function accepts various arguments and process the data and returns a results back to the caller function.

As of now we discussed a function accepts simple arguments like string or numbers, now let's discuss the functions with accepts functions as arguments.

```
 1  function greetMe1() {
 2      var output = "Good Evening 1";
 3      console.log(output);
 4  }
 5
 6  function greetMe2() {
 7      var output = "Good Evening 2";
 8      console.log(output);
 9  }
10
11  function greeter(fnName) { // function as args
12      fnName(); // calling the function
13  }
14
15  greeter(greetMe2); //Good Evening 2
16  greeter(greetMe1); //Good Evening 1
```

In the above example the *greeter*() function takes in the parameter as a function name itself and it executes the passed in function.

If we pass '*greetMe2*' to '*greeter*()' function at line number 15 , so the line number 12 will execute the '*greetMe2*()' function.

This is a very common scenario in JavaScript programming.

Functions as Arguments with Arguments

As of now we discussed about the functions accepts an arguments as functions itself but those are without arguments. Now let's understand the functions are arguments with arguments.

```
function greetHim1(name) {
    var output = "Good Evening " + name;
    console.log(output);
}

function greetHim2(name) {
    var output = "Good Evening " + name;
    console.log(output);
}

function greeterEngine(fnName,name) {
    fnName(name);
}

greeterEngine(greetHim1,"Naveen"); // Good Evening Naveen
greeterEngine(greetHim2,"John"); // Good Evening John
```

Functions inside an Objects

In the previous sections we have discussed about Objects creation, accessing the properties of an objects now let's discuss the Objects which contains functions inside of it.

This is also a very common scenario where we have an objects which contains various functions inside of them.

```
// Object contains a function
var employee = {
    firstName : "Naveen",
    lastName : "Saggam",
    fullName : function () {
        var output = this.firstName + " " + this.lastName;
        console.log(output);
    }
};

// Execution of a function inside an Object
employee.fullName(); // Naveen Saggam
```

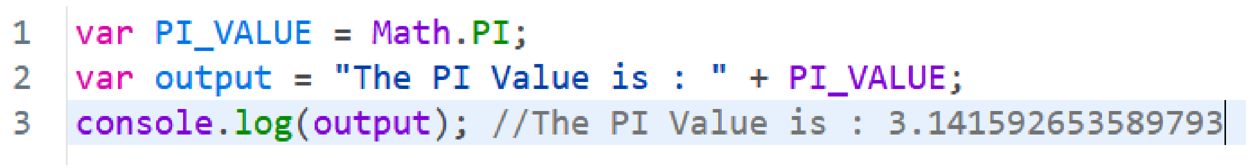
Math Object

Math Object is useful for performing various mathematical operations in developing any software system. Math Objects contains various properties to use it for any mathematical operations.

Let's understand the usage of Math Objects below,

Math.PI

Math.PI property is used to get the PI value. The usage is as follows,

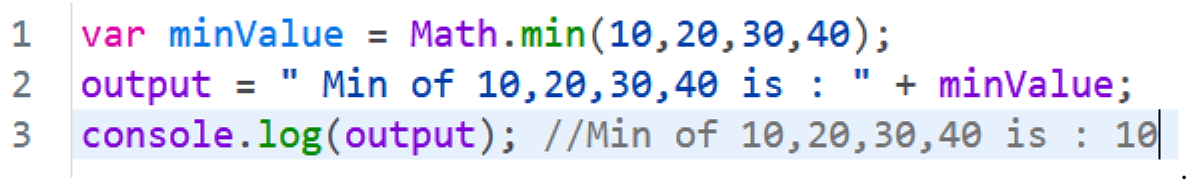


```
var PI_VALUE = Math.PI;
var output = "The PI Value is : " + PI_VALUE;
console.log(output); //The PI Value is : 3.141592653589793
```

Math.min()

This function of Math Object is used to find the min value of the specified number of parameters.

Example:

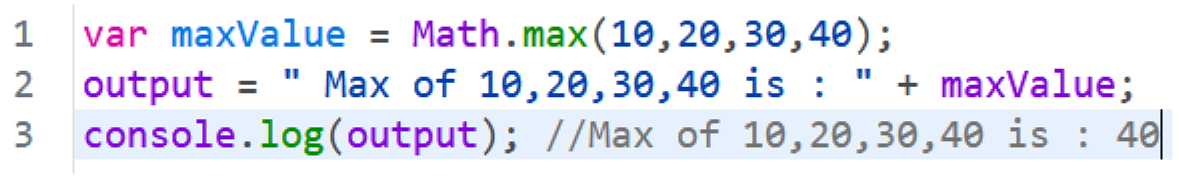


```
var minValue = Math.min(10,20,30,40);
output = " Min of 10,20,30,40 is : " + minValue;
console.log(output); //Min of 10,20,30,40 is : 10
```

Math.max()

This function of Math Object is used to find the max value of specified number of parameters.

Example:



```
var maxValue = Math.max(10,20,30,40);
output = " Max of 10,20,30,40 is : " + maxValue;
console.log(output); //Max of 10,20,30,40 is : 40
```

Math.pow()

This function of Math Object is used to find 'x' to the power of 'y' results.

Example:

```
var pow2_4 = Math.pow(2,4);
output = " 2 ^ 4 is : " + pow2_4;
console.log(output); // 2 ^ 4 is : 16
```

Math.random()

This function of Math Object is used to get a random value between 0 and 1. This method we can use to generate any random values of any range.

Example: To generate random numbers from 0 to 1000 as follows,

```
var random = Math.round(Math.random() * 1000);
output = " The Random Value : " + random;
console.log(output); // The Random Value : 834
```

Note: to know more about Math Object please have a look at the below link.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

Date Object

The Date object is used to work with Date and Time related calculations.

This Date Object's properties and functions are very much useful to get the time and dates of various time Zones.

Let's understand the properties and functions of Date Object and their usage.

In order to use Date Object, first we have to create a Date Object.

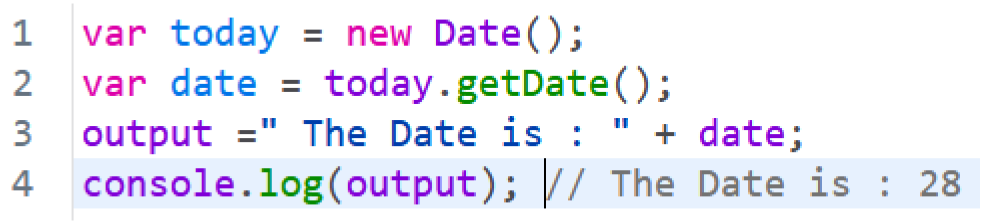
Example:

```
var date = new Date();
```

Get Today's Date

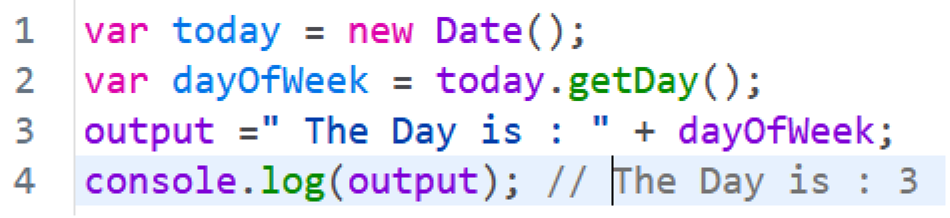
```
var today = new Date();
output = " Today is : " + today;
console.log(output);
// Today is : Wed Mar 28 2018 10:59:19 GMT+0530 (India Standard Time)
```

Get Date of the Month



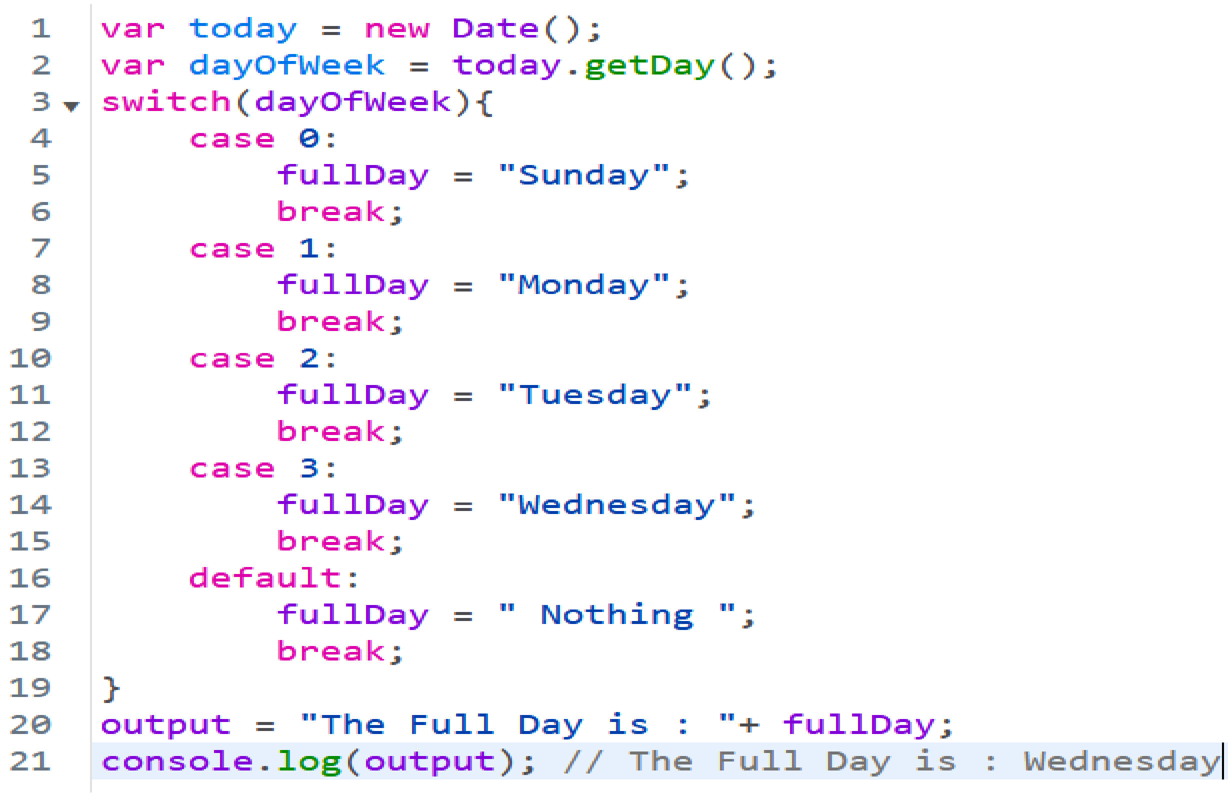
```
var today = new Date();
var date = today.getDate();
output =" The Date is : " + date;
console.log(output); // The Date is : 28
```

Get Day of a Week



```
var today = new Date();
var dayOfWeek = today.getDay();
output =" The Day is : " + dayOfWeek;
console.log(output); // The Day is : 3
```

Get Full Day of the week



```
var today = new Date();
var dayOfWeek = today.getDay();
switch(dayOfWeek){
    case 0:
        fullDay = "Sunday";
        break;
    case 1:
        fullDay = "Monday";
        break;
    case 2:
        fullDay = "Tuesday";
        break;
    case 3:
        fullDay = "Wednesday";
        break;
    default:
        fullDay = " Nothing ";
        break;
}
output = "The Full Day is : "+ fullDay;
console.log(output); // The Full Day is : Wednesday
```

Note: To Know more about Date Object please have a look at the below link

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date

Dom Manipulation

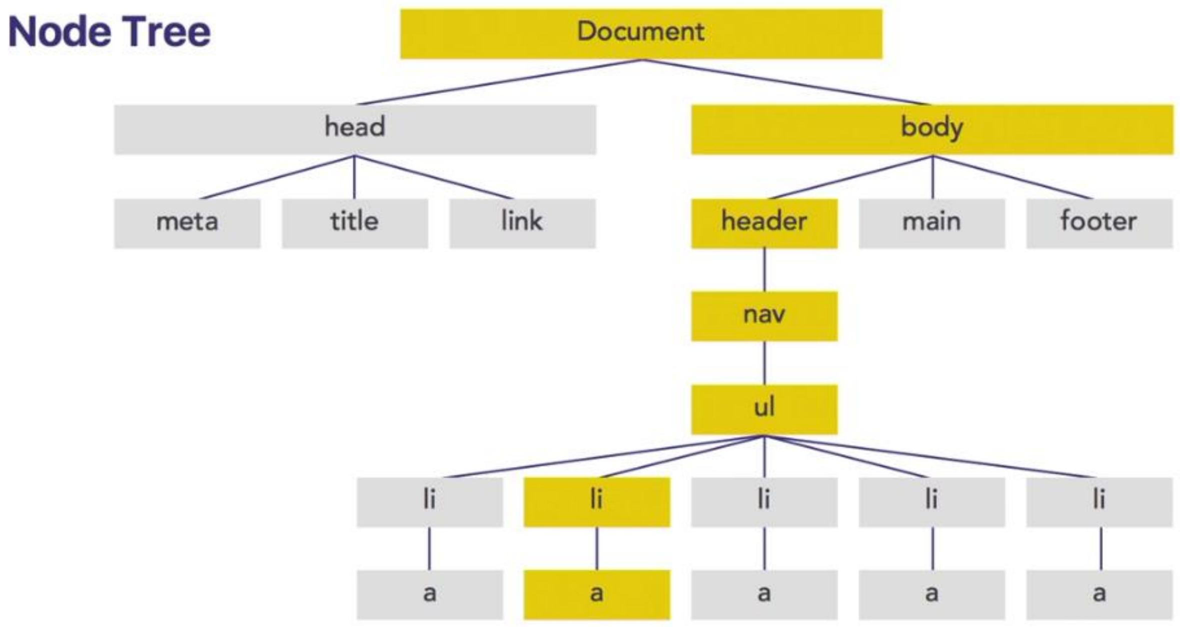
The JavaScript is used for browser side scripting and to provide dynamic nature to any website.

JavaScript provides dynamic nature to any webpage using the concept called DOM Manipulation.

First let's understand the concept of DOM (Document Object Model).

Whenever browser gets a HTML file before displaying the content on the browser, first it reads the entire HTML page and prepares a tree like structure using all the HTML tags / elements. This Tree like structure is called DOM Tree. Each html element inside the DOM Tree is called as 'DOM Node'.

Example:



The above diagram denotes a Node Tree or DOM Tree.

Once this tree structure is built, then the browser renders each html element / tag and display them on the browser / webpage.

How the JavaScript provides the dynamic nature to HTML is, by using JavaScript we will fetch each of the HTML tag / element / DOM Node from the DOM Tree and adds our own content so that browser parses the new DOM Tree nodes and displays them on the browser.

This DOM Tree of Nodes are responsible for the display of content on the webpage. So in the backend by using JavaScript we will change the DOM Nodes using the Concept called DOM Manipulation then we will get the Dynamic view on the webpage.

By Using JavaScript we can do the following to the DOM Tree.

1) Change the existing DOM Node Content
2) Add a new DOM Node to the DOM Tree
3) Remove a DOM Node from the DOM Tree

Note: Please note HTML Element, HTML Tag, DOM Node all denotes the same.

So for each HTML Document there will be one DOM Tree. So by using JavaScript we can change the HTML Content to be displayed on the webpage.

By using JavaScript we can change not only DOM Nodes but also Browser related properties.

In a Browser each tab is represented with one DOM Tree. A Browser itself is also an Object and it contains a collection of DOM Trees.

This is called as Browser Object Modal.

BOM:
Browser Object
Model

In Browser Object Model the browser itself is treated as an Object and it contains a long list of Objects as follows,

1) Document (DOM Tree)
2) Navigation buttons
3) URL

All these Objects are modeled together and is called as BOM (Browser Object Model).

NOTE: In the BOM Model the top level object is 'window' and it represents the entire Browser itself.

In the DOM Model the top level object is 'Document' and it represents the entire Document itself.

Let's discuss some of the operations that we can perform on BOM Objects by using JavaScript.

```
window.innerWidth; // to access the width of the Browser

window.innerHeight; // to access the height of the Browser

window.open(); // to open a new Tab

window.document; // to access the Document object
```

We can access the Document Object using either 'window. document' or directly by using 'document'.

Let's understand the DOM Manipulation using JavaScript as follows,

We can manipulate the DOM Nodes using the Document Object's properties and methods/functions.

DOM Properties

The following are the various DOM properties to be accessed by JavaScript.

```
// To access the complete body element
var body = document.body;

// To access the title of the current webpage
var title = document.title;

// to access the URL of the current webpage
var url = document.URL;
```

DOM Methods / Functions

The following are the various methods to be used by JavaScript to manipulate the DOM nodes.

```
// to select a HTML Element with the specific ID (main-nav)
document.getElementById('main-nav');

// to select all the HTML Elements with the class name (container)
document.getElementsByClassName('container');

// to select all the HTML Elements using the HTML tag (img)
document.getElementsByTagName('img');

// to select the first element matching the selector (.main-nav a)
document.querySelector('.main-nav a');

// to select all the HTML elements matching the selector (.container p)
document.querySelectorAll('.container p');
```

Let’s discuss some of the examples

```html
<div id="text-div">
    <p id="green-p">This is a Green Color Text</p>
    <button onclick="blue();">Change to Blue</button>
    <button onclick="red();">Change to Red</button>
</div>

<script>
    function blue() {
        document.getElementById('green-p').style.color = 'blue';
    }
    function red() {
        document.getElementById('green-p').style.color = 'red';
    }
</script>
```

In the above example we will change the text color dynamically while pressing blue , red buttons using DOM Manipulation.

Example:

```html
<div id="image-div">
    <img src="../img/google.jpg" width="300px" height="200px" id="myImage">
    <br>
    <button onclick="facebook();">Facebook</button>
    <button onclick="youtube();">Youtube</button>
</div>
```

```html
function facebook() {
    document.getElementById('myImage').setAttribute('src','../img/facebook.jpg');
}

function youtube() {
    document.getElementById('myImage').setAttribute('src','../img/youtube.jpg');
}
</script>
```

In the Above Example if click on 'Facebook' button we will get the Facebook image and if we click on YouTube button we will get the YouTube image dynamically by using JavaScript DOM manipulation.

JavaScript DOM Events

Any kind of operation or an action that can be performed on a browser is called as an event.

The moment a user is performing an action on a browser is an event.

Let's discuss some of the DOM Events,

1) Opening a browser is an event
2) Open address bar and enter URL is an event
3) User clicks on a link is an event
4) Move a mouse cursor is an event
5) Click on a button is an event
6) Focus on an input field is an event.

Mouse Events	Keyboard Events	Form Events	Document Events
Click	Keypress	Submit	Load
Dbclick	Keydown	Change	Resize
Mouseenter	Keyup	Focus	Scroll
mouseleave			unload

JS DOM Event Handling

In JavaScript we can perform the DOM Manipulation using various event handling methods as specified above.

The Actual Event handling event handling works as follows,

1) Find a node to which we trigger an event
2) Find the events of the node
3) Write a script to trigger an event for that node.

Example:

```
 HTML Code
-----------------
<div id="image-div">
    <img src="../img/google.jpg" width="300px" height="200px" id="myImage">
    <br>
    <button onclick="facebook();">Facebook</button>
    <button onclick="youtube();">Youtube</button>
</div>

JavaScript Code
------------------
function facebook() {
    document.getElementById('myImage').setAttribute('src','../img/facebook.jpg');
}

function youtube() {
    document.getElementById('myImage').setAttribute('src','../img/youtube.jpg');
}
```

JS DOM Event Listeners

In the process of DOM Manipulation first we will get a html element and hook up and event method and we can perform some operations on it.

In the process of event listeners, we normally hook up an event listener to the specific HTML element and attach an event handling method to it. If the specified event happens then the hooked up event handling method will be executed.

Syntax of Event listener is as follows,

```
htmlElement.addEventListener('event_name',event_handling_function);
```

Let's discuss an example of Event Listeners,

```
 HTML Code
 ------------

<textarea id="text-area" rows= '5' cols='10'></textarea>

JavaScript Code
-------------------
var textArea = document.querySelector("#text-area");

textArea.addEventListener('keypress',start);

function start() {
//statements of event handling
}
```

In the above example of Event Listeners way of event handling works as follows,

1) First we needs to fetch the required HTML Element using any DOM manipulation methods.(line no : 8)
2) We needs to hookup an event handling method to it along with the event name and event handling function name. (line no: 10)
3) In the last we have define a method / function for handling that event.

Note: Whenever the 'keypress' event happened for the text-area then start() will be executed.

JavaScript Scopes

JavaScript Scopes & Closures is one of the important concept to learn in JavaScript. Without these concepts also we can write the JavaScript code, but once we learn these concept code we may write more elegant code and most of the bug free code.

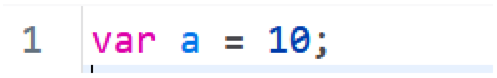
Now let's understand the concept of scopes and then closures concept.

Scopes & Block Scoping

Scopes concept is not only specific to JavaScript, these kind of terms we can even see in other OOPS languages.

A scope is considered as a block where the accessibility of variables exists. The variables declared inside the block are not accessible outside.

Example:



```
var a = 10;
```

If we declare a variable somewhere in the middle of a program, can we guess what are all the places it is accessible?. It is purely depend on the scope where we declare that variable is declared.

If the variable declared in the global scope, It will be accessed throughout the program. If the variable is declared in any block, then we can access the variable inside of that block only, outside of that block it is not accessible.

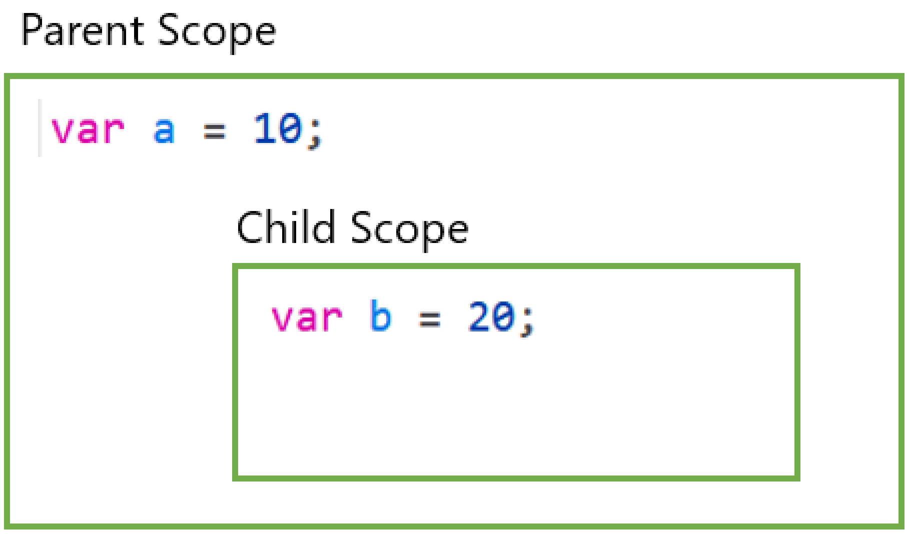
Example:

A Fish in an aquarium is having its scope inside of that aquarium only, when we bring the fish outside the aquarium then it will die. Means the scope of the fish is inside the aquarium only. So a variable declared inside of a block / scope is not accessible outside of that block or scope.

Let's discuss the concept of scope hierarchy

Scope Hierarchy

Scope Hierarchy means a chain of scopes which contains a parent scope and child scope.



```
var a = 10;
```

```
var b = 20;
```

Whatever the variable is declared inside parent scope are by default available to the child scope.

So 'x' is accessible in child scoped block.

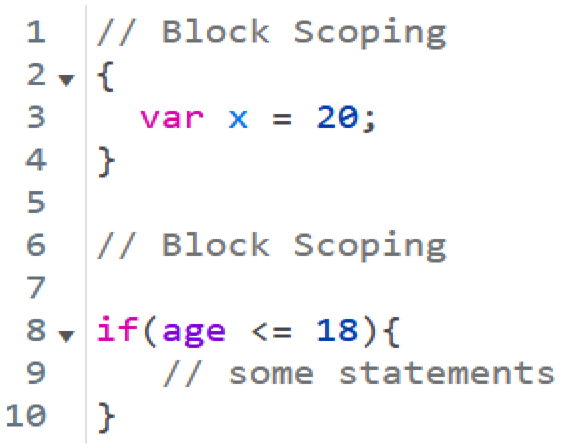
But whatever the variable is declared in child scope is not available to parent scope by default.

So 'y' is not accessible outside of child scope and also inside parent scoped block.

Scopes Creation

In any other OOPS languages like C++ or Java, we will be creating scopes as

Example:



```
// Block Scoping
{
  var x = 20;
}

// Block Scoping

if(age <= 18){
   // some statements
}
```

So in Java, C++ just placing any code inside curly braces { } creates scope of a variable of block and outside of this block the variable is not accessible, this is called as block scoping.

But JavaScript does not supports block scoping and JavaScript supports Function Scoping. In JavaScript just by creating a block it will not be a separate scope creation. We can create a new scope using functions in JavaScript.

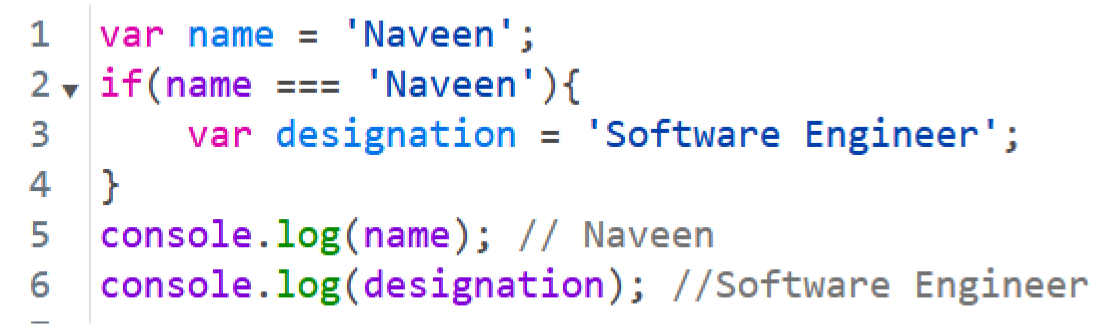
Function Scoping

In JavaScript, creation of scoping is not block level but it is a function level.

JavaScript does not encourage block scoping but in in JavaScript it is only Function Scoping.

Simply creating a block will not create a new scope in JavaScript.

Example:



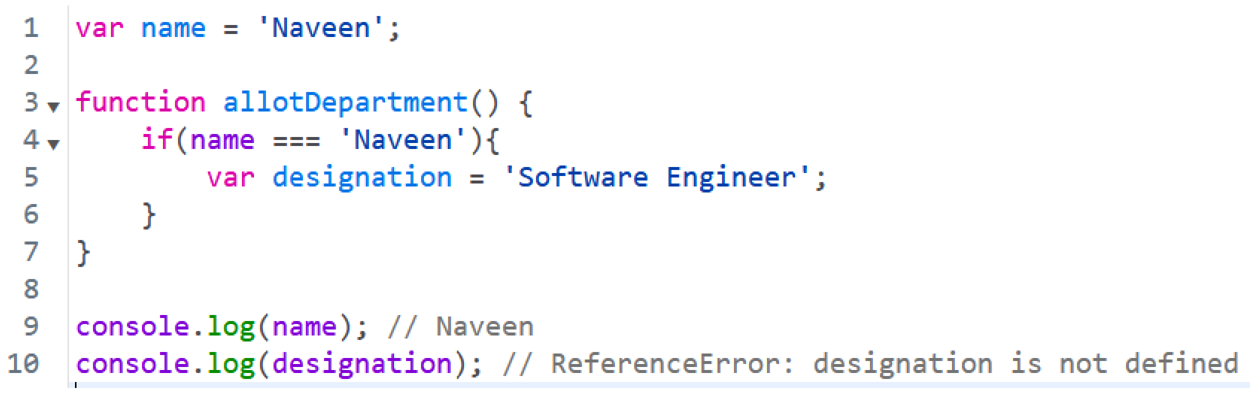
```
var name = 'Naveen';
if(name === 'Naveen'){
    var designation = 'Software Engineer';
}
console.log(name); // Naveen
console.log(designation); //Software Engineer
```

Note: In the above example we declared a variable ‘designation inside if block and still we can access that variable outside of the if block.

It clearly state that JavaScript doesn’t support Block Scoping. JavaScript supports Function Scoping.

Now let’s keep the same code inside a function.

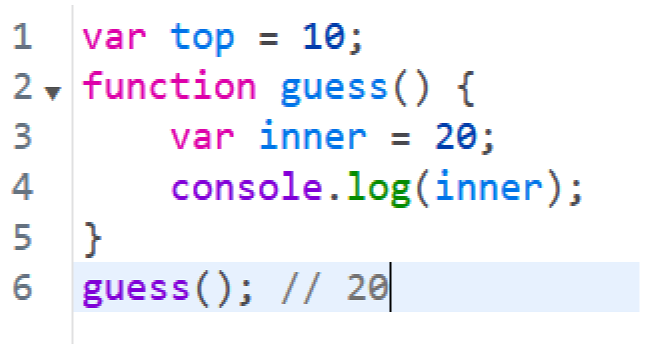
Example:



```
var name = 'Naveen';

function allotDepartment() {
    if(name === 'Naveen'){
        var designation = 'Software Engineer';
    }
}

console.log(name); // Naveen
console.log(designation); // ReferenceError: designation is not defined
```

Now discuss few examples on Scoping Concepts

Example 1:



```
var top = 10;
function guess() {
    var inner = 20;
    console.log(inner);
}
guess(); // 20
```

In the above example we declared a variable ‘inner’ inside guess() function and we accessed that variable inside of that function only.

Example 2:

```
var top = 30;
var inner = 30;
function guess() {
    var inner = 40;
}
console.log(inner); // 30
```

Here in the above example we declare two inner variables one is outside of the function and one is inside a function. But we are accessing an inner variable outside of the function then obviously we cannot access the variable which is declared inside of the function. So we can access the variable declared outside of the function, because we are executing console.log outside of the function only.

IIFE (Immediately Invoked Function Expression)

We normally create variables outside of functions and they can be accessed anywhere of the program. These variables are called as global scoped variables.

But in JavaScript creating variables in global scope is not a good idea.

For any web application there may be a lot of JavaScript files will be executing on the browser. Then the variable created in global scope are stored in the same memory area of all the JavaScript files global scoped variables and it is accessible to all the other JavaScript files also. This kind of behavior causes very serious issues in JavaScript programming.

So placing everything in global scope is not at all recommended and it is not a good programming practice.

In order to avoid the global scoping, we can create those variable inside a function.

Example:

```
function addFn() {
    var a = 10;
    var b = 20;
    console.log(a + b);
}

addFn(); // 30
```

In order to avoid creating a, b variables globally we have created inside a function. But just by creating a function and adding variables will not get printed anything on the console. So we needs to call the function also.

But here there is a new problem arises, we have created variables inside the function and we are calling the function 'addFn()' from global scope.

Let's assume, if any other JavaScript file is also have the same function and we are calling that function. Now again the same problem with global scoping.

In order to avoid this problem we can use IIFE as follows,

```
(function() {
    var a = 10;
    var b = 20;
    console.log(a + b);
})();

// output : 30
```

Advantages of IIFE:

1) Avoids creating variables in global name space.
2) Functions doesn't have a name to collide with any other JavaScript files.
3) Avoid Global scoping of function call.

Read & Write Operations

This is not related database level read & write operations. This read and write operations are specific to declaration and definition of variables and reading them.

This concept is important to understand the behavior of variables in JavaScript.

Example:

```
var a = 10; // Write Operation
console.log(a); // Read Operation
```

In the first line, we are writing a value 10 to variable 'a'. This is called as read operation.

In the second line we are reading a value from 'a' so it is a read operation.

```
var a  = 10; // write operation
var y = 20; // write operation

var z = y; // Read & write operations

// reading from y and assigning to z
```

Example

```
function greet(wish) { // write operation
    console.log(wish); // read operation
}

greet('Good Morning');
```

Implications of Read & Write Operations

In any other OOPS languages in order to use a variable, first thing is we have to declare the variable. We can use a variable only after declaring it.

Here in JavaScript the variable declaration is purely depend on what are going to do with the variable.

```
var name;
console.log(name); //undefined
```

Let's access an undeclared variable,

```
console.log(department); // ReferenceError: department is not defined
```

Let's assign some value to undeclared variable,

```
isJSEasy = true; // No error
```

```
isJSEasy = true; // No error
console.log(isJSEasy); // true
```

NOTE: Write Operation is possible for declared and undeclared variables. But for Read operation it is possible only for declared variables.

Compilation & Interpretation

As most of us knows that JavaScript is an interpreted language. But technically it is Compiled and Interpreted language.

When compared to other programming languages like Java and C++, there is actually a compilation step happens and then creates an intermediate code and that will be executed in runtime.

But in JavaScript, there is no such intermediate code like Java or C++. But still in JavaScript there is no intermediate code will be generated and the compilation step happens inside the source itself and execute the same source code only.

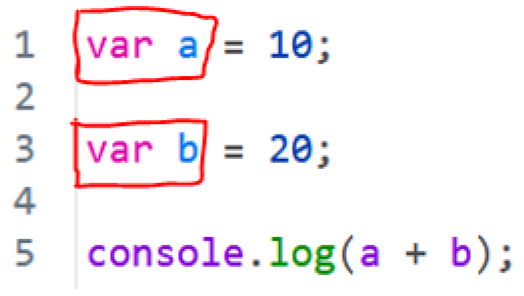
Browser actually executes the source code as is, but this does not means that there is no compilation step.

There is a compilation step in JavaScript and which is different from all other languages.

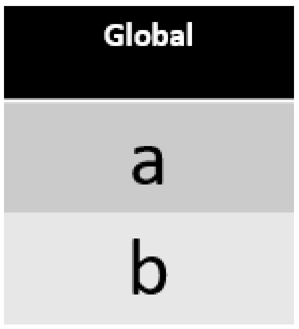
When a browser sees the JavaScript, it does not execute it directly there are some certain step happens looks at the code and note down some of the things and then starts the execution. This happens within the fraction of seconds before the actual execution starts.

Compilation Phase

Compilation step mainly looks for variable declaration and function declaration only.



```
var a = 10;

var b = 20;

console.log(a + b);
```

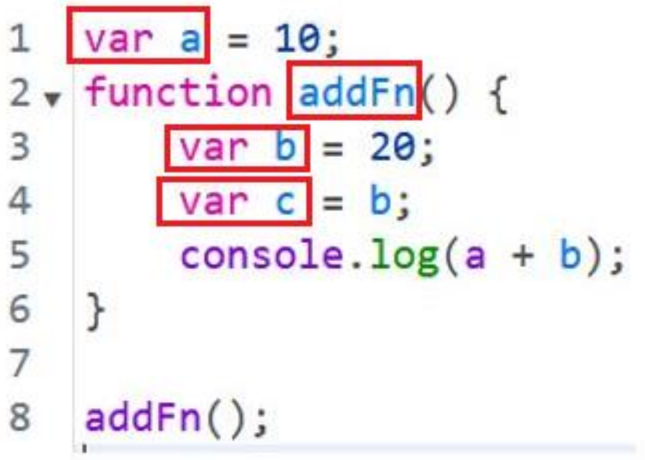


Compilation step looks for the 'var' and add them to the global scope, not the assignment phase.

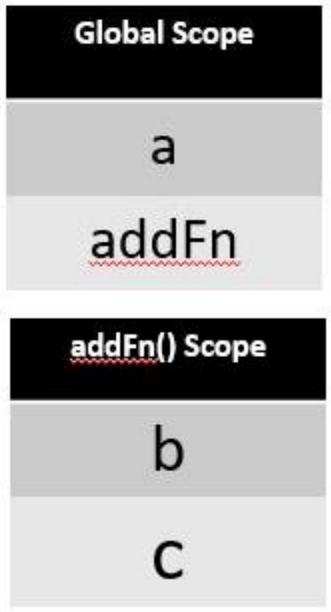
1) It looks for 'var a' and adds it to a global scope.
2) As it does not care about the assignment phase.

3) It looks for 'var b' and adds it to global scope.
4) Console.log() is not care by compilation , so finally after compilation phase we have two variable inside global scope.

Example 2:

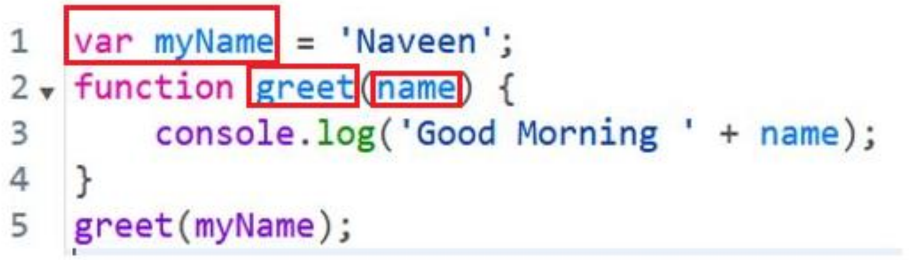


```
var a = 10;
function addFn() {
    var b = 20;
    var c = b;
    console.log(a + b);
}

addFn();
```

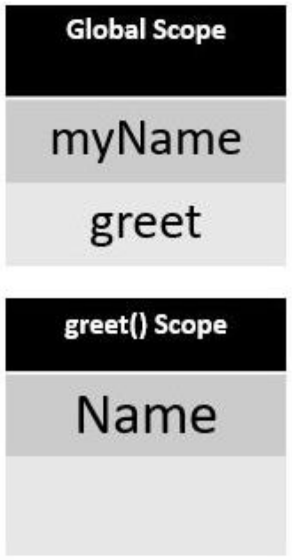


Note : In the above example 'addFn()' itself is a variable which contains a value function.

Example 3:



```
var myName = 'Naveen';
function greet(name) {
    console.log('Good Morning ' + name);
}
greet(myName);
```



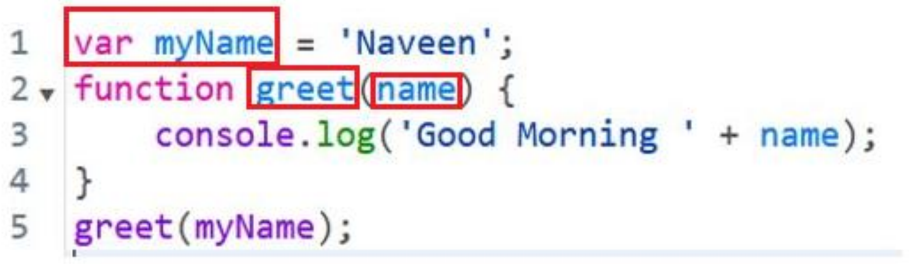
Note: For any function parameters in JavaScript are like local variables to its function.

Interpretation Phase

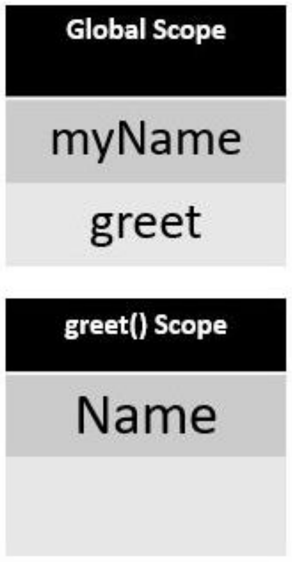
Interpretation phases follows compilation phase and it uses the scope chain created by the Compilation phase.

The Interpretation or execution phase is takes care about variable assignment and function execution.

Example:



```
var myName = 'Naveen';
function greet(name) {
    console.log('Good Morning ' + name);
}
greet(myName);
```



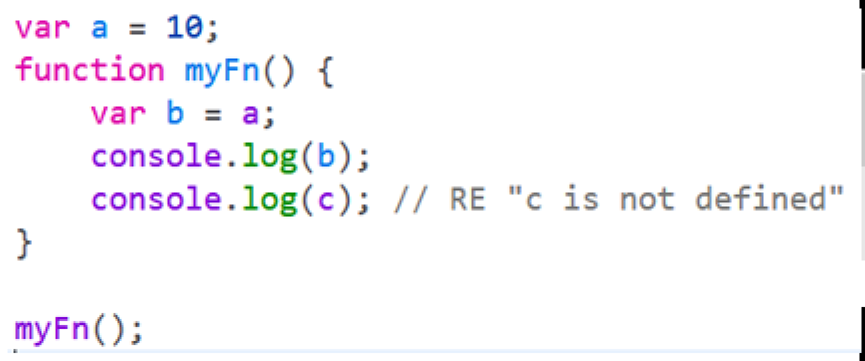
Steps:

1) Interpreter takes in the string value 'Naveen' and assign to myName variable by following the global scope.
2) It Checks if the global scope is having a 'myName'. Once found it will assign to it.
3) Interpreter does not care about function declaration.
4) It goes to function execution 'greet(name)' as this line in global scope, it checks the variable 'greet' in the global scope.
5) It gets the greet variable in global scope and also myName which is actually there in global scope.
6) Now it goes to function definition, where in greet scope. It assigns and then goes to console.log() and prints the message to the console.

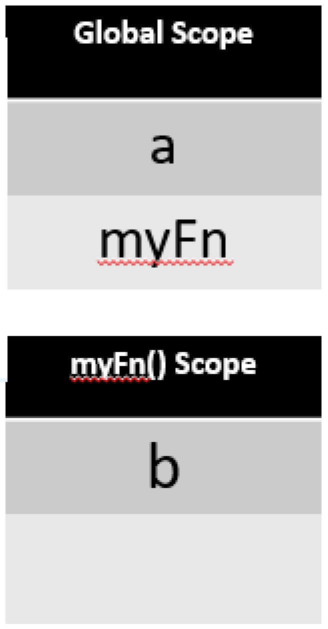
The Global Scope Problem

As of we use the variables which are already declared. What if we try to use the variable for assignment which is not even declared?

Example:



```
var a = 10;
function myFn() {
    var b = a;
    console.log(b);
    console.log(c); // RE "c is not defined"
}

myFn();
```



In the above example, the flow goes for compilation and execution, when the interpreter comes to the line

```
console.log(c);
```

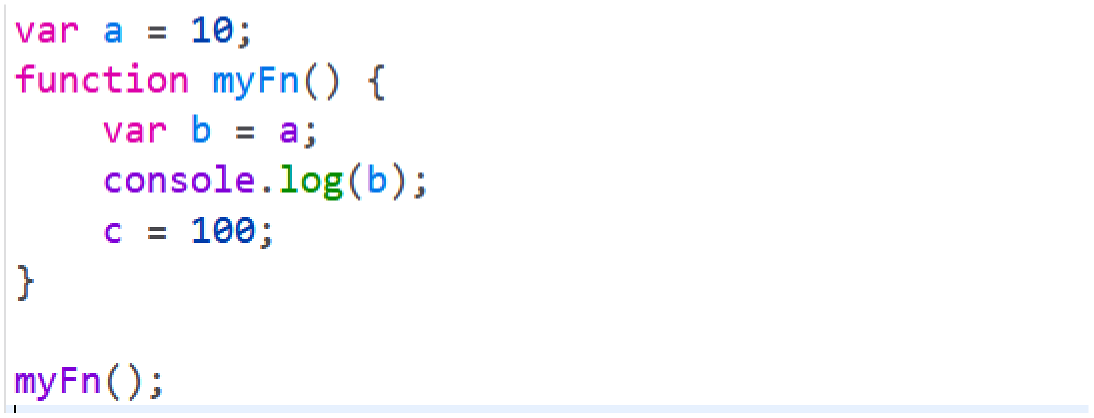
It looks for the variable ‘c’ whether it is declared in myFn Scope,as myFn scope does not have the variable declared.

It goes to one level up and checks in the global scope, if it contains a declared variable ‘c’. as the Global scope also doesn’t contain a variable ‘c’.

Now the variable is not declared anywhere else of the program. The decision will be taken by JavaScript Interpreter based on READ or WRITE operation.

As this is a READ operation and as we know the READ operation is not possible for non-declared variables, so it throws a runtime error.

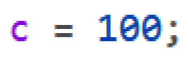
Let’s say instead of read operation we have a write operation.



```
var a = 10;
function myFn() {
    var b = a;
    console.log(b);
    c = 100;
}

myFn();
```

Again the same flow of compilation and interpretation.

And the interpreter comes to the line,



```
c = 100;
```

The interpreter looks for the variable declaration in the myFn Scope and not available and goes to one level up and looks for the variable in the global scope. The variable is still not declared in the global scope.

The decision made based on the type of operation. Since this is a 'write' operation. Write operation is possible for non-declared variables.

Now the interpreter go ahead and creates a variable in the **global** scope.

As this is one of the wired behavior of JavaScript, as we are using it in the myFn scope and JavaScript creates in the global scope.

Once a variable is declared in the global scope, this may accessible to some other JavaScript files and if they can use it and change it, the entire application may fail due to this nature.

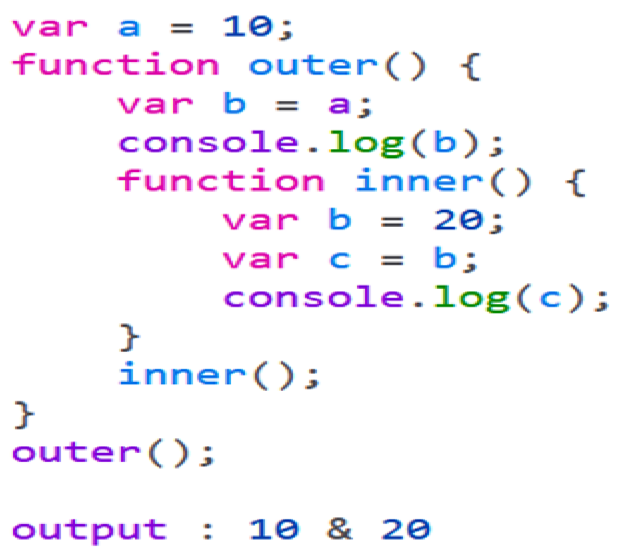
So creating Global variables in any other languages is bad and it is even worst in JavaScript.

Note:

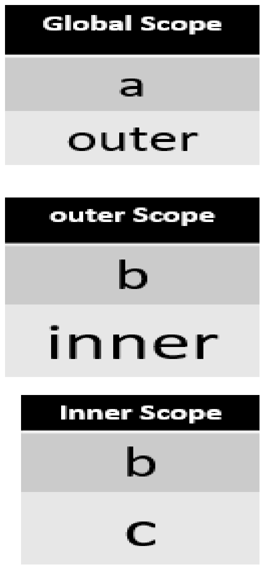
In order to avoid such situations we should always declare a variable with 'var' keyword before we starts using it.

Coding Exercises

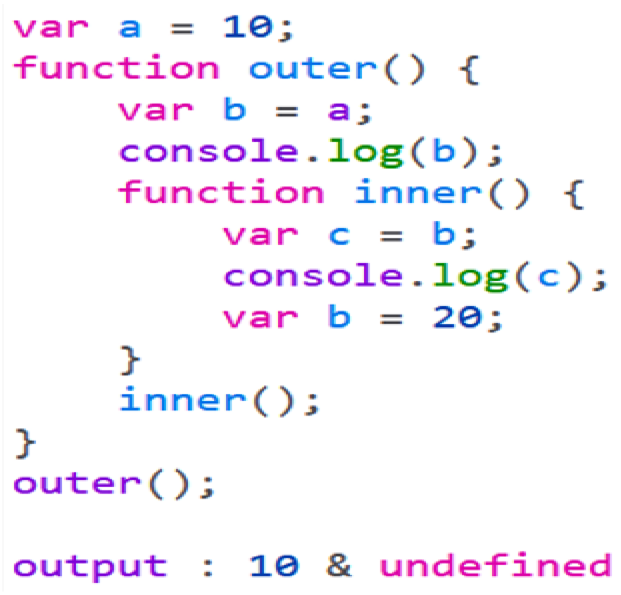
Let's understand the below code, checks for the output.



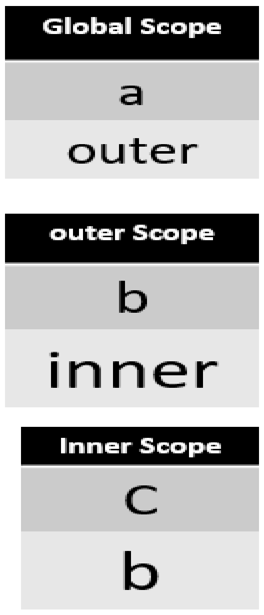
```
var a = 10;
function outer() {
    var b = a;
    console.log(b);
    function inner() {
        var b = 20;
        var c = b;
        console.log(c);
    }
    inner();
}
outer();

output : 10 & 20
```



Let's understand the below code, and guess the output



```
var a = 10;
function outer() {
    var b = a;
    console.log(b);
    function inner() {
        var c = b;
        console.log(c);
        var b = 20;
    }
    inner();
}
outer();

output : 10 & undefined
```



In the above example, the compilation step looks for inner scope and variable c, b creates in the inner scope.

Now interpreter come into picture and goes to the line

```
var c = b;
```

And checks it contains a variable 'b'. As there is a variable in the inner scope it just declared any value to it. So it contains the value 'undefined' so the same value of 'b' will be assigned to the variable 'c'.

Now it prints 'undefined' to the console and finally assigns a value '20' to the variable 'b'.

Summary:

Let's summarize the above example in a simple way as follows,

```
1  console.log(a);
2  var a = 20;
```

As at line number 1. It is a read operation and at this time the variable is not declared. We may think like we get an error.

But if observe the code carefully, before interpretation executes the console.log() , there is a compilation step and which takes care about variable declaration. So before executing the console.log(), the variable has been declared already and assigned the value 'undefined' to it.

So it prints the value 'undefined' to the console and then the interpreter assigns the value '20' to 'a'.

Hoisting in JavaScript

Hoisting is a concept available in JavaScript and once we understand about compilation and interpretation steps, this will be to understand.

The concept of hoisting is that, no matter where we declare the variables, after compilation step and before interpretation step happens all the declarations will be hoisted to the top of the declared scope.

This concept is called Hoisting in JavaScript.

Example:

```
a = 10;
console.log(b);
c++;

var a;
var b;
var c;
```

In the process of compilation it scans the entire JavaScript file and it looks for variable declarations hoist or move to the top of the declared scope.

Once the compilation step is over then interpretation follows the scope chain along with the variables declarations and starts executing them.

Once all the variables are hoisted to the top of the program, then before interpretation step the code may looks as follows,

```
var a;
var b;
var c;

a = 10;
console.log(b);
c++;
```

Note: This hoisting process is not only applies to variables, it also applies to functions.

Example:

```
// function execution
sum();

// function definition
function sum() {
    // Logic to sum
}
```

Note: even though the function declaration is placed after the functions execution, in with the process of hoisting in JavaScript (in Compilation Phase) the definition of the function is hoisted to the top.

Note: The hoisting is not possible for function expressions.

Example:

```
greet();

var greet = function() {
   console.log("Hello Good Morning");
}

//Output : TypeError: greet is not a function
```

In the above code first compilation step takes care about the variable declaration, and it just creates a variable ‘greet’ in the global scope with ‘undefined’ as the value.

Now the interpreter comes to execute the function in line number 1, as this variable already contains a value as ‘undefined’ and interpreter can’t be able to execute the ‘greet()’ as this is not a function at this point of time.

Due to this reason Hoisting is not possible for Function Expressions and the hoisting works for function declaration.

Note: While using function expressions in a JavaScript codebase, make sure you write a function expression and then execute that function expression.

Using Strict Mode:

As we discussed earlier that read operation is not possible for non-declared variables and only write operation is possible for non-declared variables.

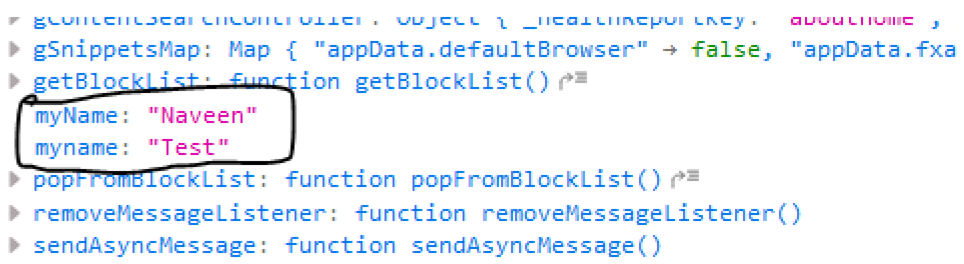
If we are trying to apply the write operation on non-declared variables, it is allowed in JavaScript but the JavaScript itself creates that variable in ‘**Global**’ scope instead of its declared scope.

In any programming language declaring any variables in the global scope is bad, it is even worst in JavaScript programming.

```
1  var myName = 'Naveen';
2  
3  // some serious logic
4  
5  myname = 'Test';
6  
7  // some serious logic
```

Let’s understand the above example, at line number 1 we have created a variable ‘**myName’** and assigned a value, after some serious code, we have assigned a value to some other variable called ‘**myname**’ instead of ‘**myName**’ and which is not declared at all. As in JavaScript the variable names are case sensitive, and both are two different variables.

As the variable ‘**myname**’ is not declared anywhere else in the program, and as it is a write operation and also write operation is possible for non-declared variables so JavaScript creates this variable in the global scope (check the window object on the browser)



```
gSnippetsMap: Map { "appData.defaultBrowser" → false, "appData.fxa
getBlockList: function getBlockList()
myName: "Naveen"
myname: "Test"
popFromBlockList: function popFromBlockList()
removeMessageListener: function removeMessageListener()
sendAsyncMessage: function sendAsyncMessage()
```

Due to the global variables are bad in JavaScript and we haven't expected the behavior of JavaScript creating a variable in the global scope.

In order to avoid this, from ES5 version we got a new feature called '**Strict Mode**'.

If we run any JavaScript in Strict Mode, It warns the non-declared variables as follows,

```
"use strict";
var myName = 'Naveen';
// some serious logic
myname = 'Test';

// ReferenceError: assignment to undeclared variable myname
```

'**Note**: We can use this Strict Mode to the entire JavaScript to run it in the strict mode just by adding this line to the top of any JavaScript File.

```
"use strict";
```

We can even execute only a function in strict mode, by adding the above line inside a function.

```
function greet(){
  "use strict";
  // Some serious logic
}
```

Note: We should always run the JavaScript in Strict Mode to avoid such errors as mentioned above.

Closures

This is one of the most important concept in JavaScript along with Scopes. This allows the programmers to write better, creative and expressive code.

A Closure is an inner function that has access to the outer (enclosing) function's variables scope chain.

The Closure is having three scope chains as follows,

1) It have access to its own scope (the variables defined in its curly braces).
2) It has access to the outer functions variables.
3) It has access to the Global Variables.

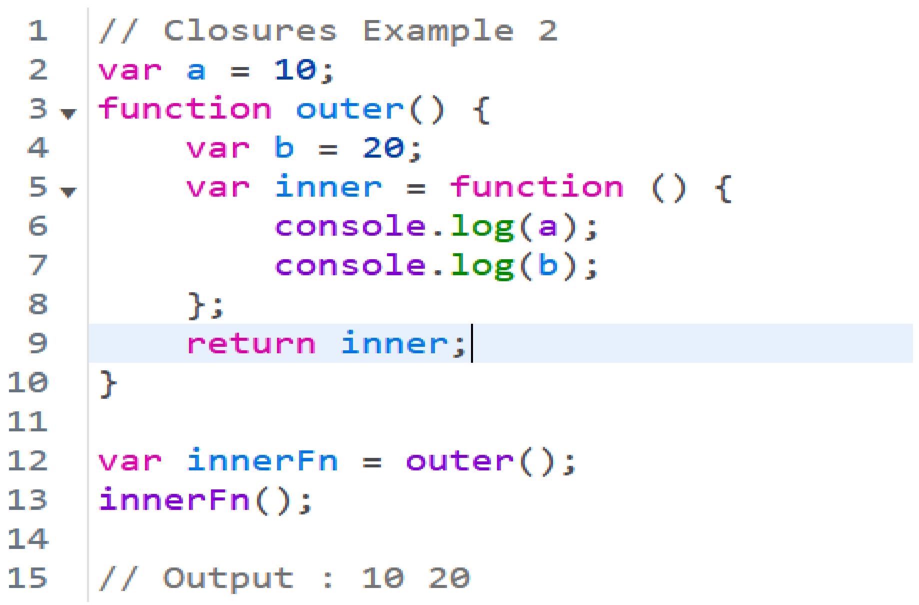
Example:

```
// Closure Example 1
var a = 10;
function outer() {
    var b = 20;
    function  inner() {
        console.log(a);
        console.log(b);
    }
    inner();
}
outer();

// Output : 10 20
```

In the above example variable 'a' is created in the global scope and variable 'b' is created in the outer scope.

We are executing the 'outer()' function in the global scope, it goes to outer() function definition and creates a variable 'b' and also executes the inner() function.

While executing the inner(), there is no variable 'a' in the inner() and outer() function, so the variable gets from the global scope and prints it on the console. And trying to access the variable 'b' in the inner() and it is not there in the inner() and then the control goes one level up and gets the variable 'b' and prints it on the console.



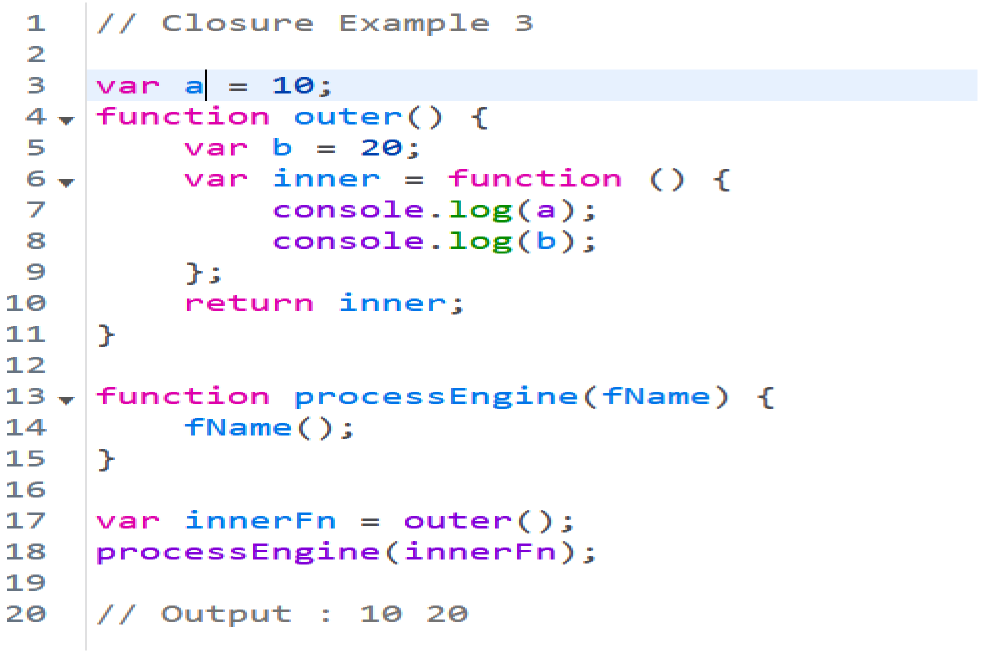
```
1  // Closures Example 2
2  var a = 10;
3  function outer() {
4      var b = 20;
5      var inner = function () {
6          console.log(a);
7          console.log(b);
8      };
9      return inner;
10 }
11
12 var innerFn = outer();
13 innerFn();
14
15 // Output : 10 20
```

In the above Example, instead of a normal function inner(), we have created a function expression, this is an isolated function inside outer().

At line number 12, we are executing the function outer(), here the variable 'b' will created in the outer() scope and the isolated function will be returned and which is assigned to 'innerFn'.

At Line number 13, we are executing an isolated function and here there is not outer involved. At this place the variable 'a' is declared in the global scope and it is available to 'innerFn' , but the scope of variable 'b' is to outer() only and how does the variable 'b' is accessible here? (this is the concept of closure).

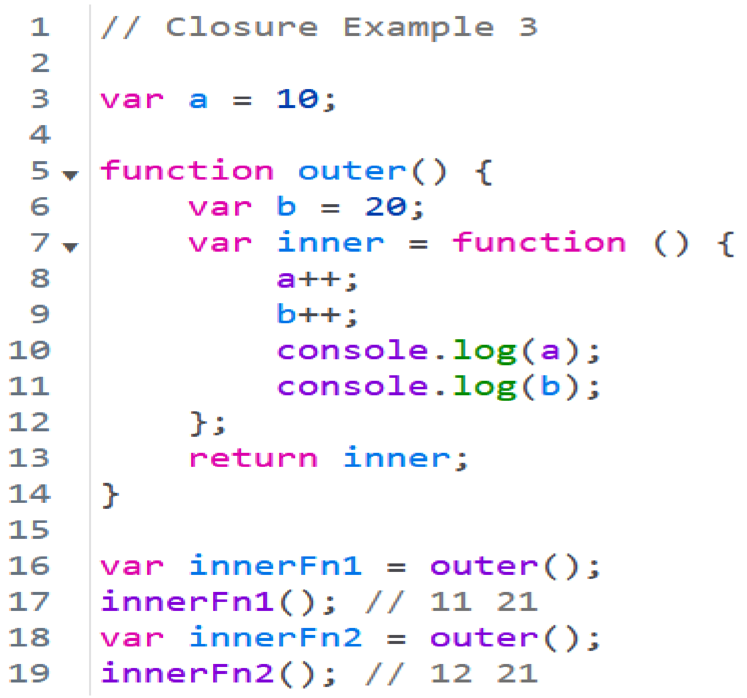
The Whole concept of closures is, at the time a function creation it remembers its scope chain and if we execute that function in a totally different context, it still remembers its scope chain. This is the concept of closure.



```
1   // Closure Example 3
2
3   var a = 10;
4   function outer() {
5       var b = 20;
6       var inner = function () {
7           console.log(a);
8           console.log(b);
9       };
10      return inner;
11  }
12
13  function processEngine(fName) {
14      fName();
15  }
16
17  var innerFn = outer();
18  processEngine(innerFn);
19
20  // Output : 10 20
```

In the above Example, at line number 17 we got an isolated function 'innerFn', this variable we are passing to another function called 'processEngine()' this is even inside a different context.

Here in line number 18, we are executing the processEngine() function, due to the concept of closures it still remembers its scope chain and prints the values 10 , 20 to the console.



```
// Closure Example 3

var a = 10;

function outer() {
    var b = 20;
    var inner = function () {
        a++;
        b++;
        console.log(a);
        console.log(b);
    };
    return inner;
}

var innerFn1 = outer();
innerFn1(); // 11 21
var innerFn2 = outer();
innerFn2(); // 12 21
```

In the above example, whenever we execute a function, then a new copy of variables will created. And there will only one copy of global variable.

Due to the concept of closures and above information it prints the values as specified in the example.

The Module Pattern

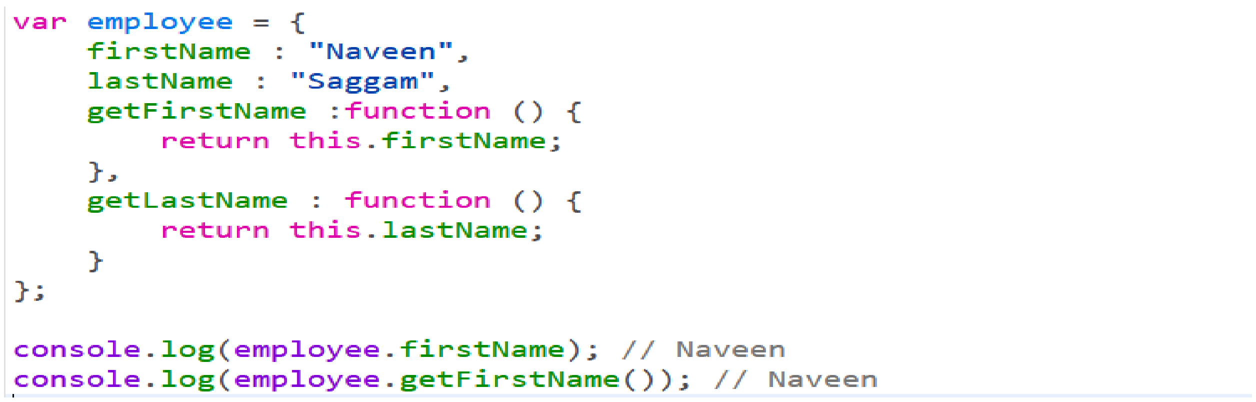
The Module pattern is one of the most commonly used design pattern in JavaScript to create Encapsulation for our code.

Normally in any other language like Java, we may create an encapsulation using 'private' keyword. The variables which are declared with 'private' keywords are not accessible to outside. Only through setter or getter method we will allow the access.

As in JavaScript there is no concept of 'private' keywords, all the properties / variables which are declared are by default 'public'.

By using the concept of Closures we can implement the Encapsulation to our JavaScript Code.

Here if we create any variables inside an objects, which accessible inside the functions inside an object as follows,

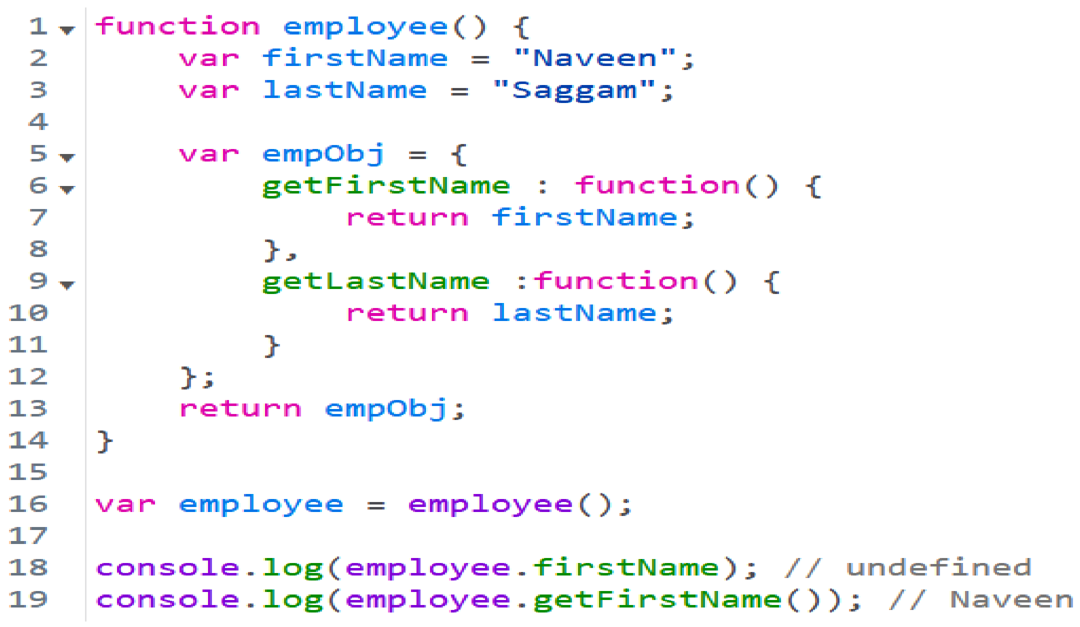


```
var employee = {
    firstName : "Naveen",
    lastName : "Saggam",
    getFirstName :function () {
        return this.firstName;
    },
    getLastName : function () {
        return this.lastName;
    }
};

console.log(employee.firstName); // Naveen
console.log(employee.getFirstName()); // Naveen
```

In the above example, we can access the first Name directly and also by using the getFirstName() function.

As per encapsulation concept we shouldn't access the firstName and lastName properties directly, we should be able to access them only by using the function names.

We can achieve this by the concept called **closures.** As follows,



```
function employee() {
    var firstName = "Naveen";
    var lastName = "Saggam";

    var empObj = {
        getFirstName : function() {
            return firstName;
        },
        getLastName :function() {
            return lastName;
        }
    };
    return empObj;
}

var employee = employee();

console.log(employee.firstName); // undefined
console.log(employee.getFirstName()); // Naveen
```

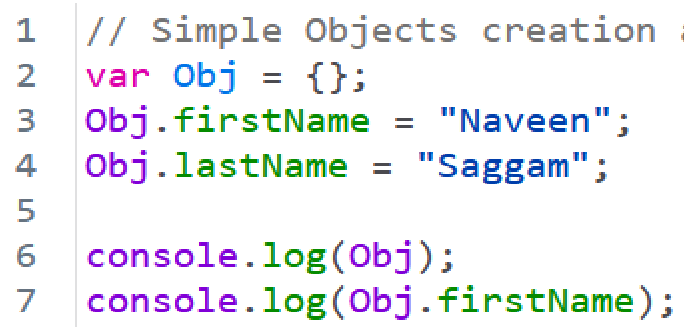
The above example denotes the Module Pattern in JavaScript where we implemented the concept of Encapsulation.

Objects in depth

Here we understand the concept of Constructor functions in JavaScript. The usage of constructors is one of the best way of creating objects.

First we understand the creation of objects and also observe how we can do this in an easy way using constructor functions.

We can create an object in JavaScript as follows,



```
// Simple Objects creation and access in JS
var Obj = {};
Obj.firstName = "Naveen";
Obj.lastName = "Saggam";

console.log(Obj);
console.log(Obj.firstName);
```

Here we have created an empty object and added all the properties to it after creation of an empty object.

Let’s create an employee object using the above approach,

```
// Creation of Employees Objects with raw data
var employee1 = {};
employee1.firstName = "John";
employee1.lastName = "Kennedy";
employee1.gender = "M";
employee1.designation = "Regional Manager";

console.log(employee1);
```

Let's create another employee object,

```
var employee2 = {};
employee1.firstName = "John";
employee1.lastName = "Cena";
employee1.gender = "M";
employee1.designation = "Delivery Manager";

console.log(employee2);
```

In order to create an employee management system and create a 100 employee objects, we may have to repeat the above steps for all the 100 employees.

Instead of this we can create a function do create an object and stuff all the properties and returns that object.

Example:

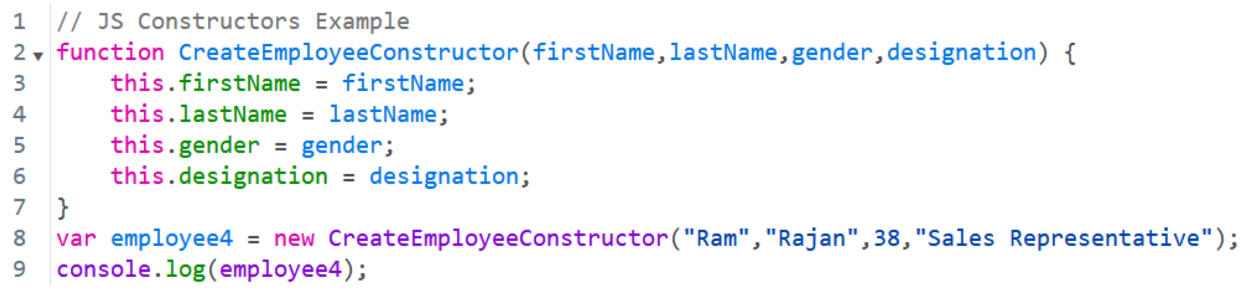
```
// Creation of a function to produce Employee Object
function createEmployee(firstName,lastName,gender,designation) {
    var employeeObj = {};
    employeeObj.firstName = firstName;
    employeeObj.lastName = lastName;
    employeeObj.gender = gender;
    employeeObj.designation = designation;
    return employeeObj;
}

var employee3 = createEmployee("Ram","Rajan",38,"Sales Representative");
console.log(employee3);
```

By using the above function we can create any number of Employee Objects using only one function.

This kind of Objects creation pretty common in JavaScript, for this approach we may use a new function called 'constructor function'.

In the above function, we created an empty object and added all the properties to that object and returns that object. So creation of an empty object and returning of that object is common for all the objects creation inside functions. This can be done automatically by JavaScript using **Constructing Functions**.

Example:



```
// JS Constructors Example
function CreateEmployeeConstructor(firstName,lastName,gender,designation) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.gender = gender;
    this.designation = designation;
}
var employee4 = new CreateEmployeeConstructor("Ram","Rajan",38,"Sales Representative");
console.log(employee4);
```

The above example denotes the JavaScript's Constructor function, we need to add the properties to **this** object (which is provided by JavaScript automatically) and also while executing the constructor function we must call the function using **new** operator.

While executing a constructor function using **new** keyword informs JavaScript to execute a function in a Constructor way.

Interview Questions on JavaScript

1) What is JavaScript
2) What is the difference between Java & JavaScript
3) What are the first class functions in JavaScript?
4) Explain the different datatypes of JavaScript.
5) What is the difference between undefined and null.
6) How many ways we can print the logs in JavaScript and explain.
7) What are the different ways of using JavaScript and which approach is the best and why?
8) Explain about String Concatenation operator in JavaScript.
9) What is the use of 'typeof' operator in JavaScript?
10) What is the difference between = , == , === operators in JavaScript
11) Print the values from 1 – 10 using for loop, while, and do while loops.
12) What is the use of 'break' statement in Switch statements what happens if we don't use break statements in switch.

13) Explain about objects in JavaScript and the ways of creation of objects in JavaScript.
14) What is the difference between DOT notation and [] notation and when to use what.
15) Explain about nested Objects creation, usage and access of properties of nested objects.
16) What is an Array and usage of array?
17) What is the difference between shift() and unshift() functions of an array.
18) What is the difference between push() and pop() in an array.
19) What is the difference between splice() and slice() in array.
20) Usage of join() function in an array of JavaScript.
21) What is the JavaScript function and execution?
22) Is JavaScript function accepts less / more number of parameters as defined in the function definition?
23) Why function overloading is not possible in JavaScript?
24) How to call a function with return values.
25) What is the difference between function expression and anonymous function expression?
26) Can we pass a function itself is an arguments to another function? Explain with an example.
27) Explain about functions inside an object and how to access the properties of an object inside the function.
28) Generate some Random numbers from 1 – 1000 using Math Object.
29) Write a code to display a full day using Date Object in JavaScript.
30) Write a code to display a digital clock in Indian Time Zone.
31) What is DOM manipulation in JavaScript.
32) What is the difference between DOM and BOM in JavaScript.
33) Explain some of the BOM properties in JavaScript.
34) Explain about few DOM properties and methods.
35) What is the difference between document.querySelector() and document.querySelectorAll() methods.
36) What is an Event in JavaScript and what is the various DOM Events.
37) Explain the process of DOM Event Handling in JavaScript.
38) Explain about the usage of DOM Event listeners in JavaScript.
39) What is Scope in JavaScript?

40) What is the difference between block scoping and Function Scoping in JavaScript?
41) Explain about the Scope hierarchy in JavaScript.
42) How to create scopes in JavaScript?
43) Why JavaScript supports only function scoping.
44) Explain about IIFE in JavaScript.
45) What are the advantages of IIFE in JavaScript?
46) Explain about Read & Write Operations in JavaScript.
47) What are the implications of Read & Write Operations in JavaScript?
48) Explain about the compilation phase of JavaScript.
49) Explain about the Interpretation Phase of JavaScript.
50) What is the Global scoping problem in JavaScript?
51) What is Hoisting in JavaScript?
52) Explain the Hosing concept for variables and functions.
53) Why Hoisting is not possible for function Expressions?
54) What is the usage of Strict Mode in JavaScript?
55) Explain the Closures concept in JavaScript.
56) Explain about the Module Pattern in JavaScript
57) What is the difference between the normal function and Constructor function in JavaScript?
58) Write a logic for this in JavaScript.
    The parameter weekday is true if it is a weekday, and the parameter vacation is true if we are on vacation. We sleep in if it is not a weekday or we're on vacation. Return true if we sleep in.

    sleepIn(false, false) → true
    sleepIn(true, false) → false
    sleepIn(false, true) → true

59) Write a logic for this in JavaScript
    We have two monkeys, a and b, and the parameters aSmile and bSmile indicate if each is smiling. We are in trouble if they are both smiling or if neither of them is smiling. Return true if we are in trouble.

    monkeyTrouble(true, true) → true
    monkeyTrouble(false, false) → true

monkeyTrouble(true, false) → false

60) Write a logic for this in JavaScript
Given two int values, return their sum. Unless the two values are the same, then return double their sum.

sumDouble(1, 2) → 3
sumDouble(3, 2) → 5
sumDouble(2, 2) → 8

61) Write a logic for this in JavaScript
Given an int n, return the absolute difference between n and 21, except return double the absolute difference if n is over 21.

diff21(19) → 2
diff21(10) → 11
diff21(21) → 0

62) Write a logic for this in JavaScript
We have a loud talking parrot. The "hour" parameter is the current hour time in the range 0..23. We are in trouble if the parrot is talking and the hour is before 7 or after 20. Return true if we are in trouble.

parrotTrouble(true, 6) → true
parrotTrouble(true, 7) → false
parrotTrouble(false, 6) → false

63) Write a logic for this in JavaScript
Given a string and a non-negative int n, return a larger string that is n copies of the original string.

stringTimes("Hi", 2) → "HiHi"
stringTimes("Hi", 3) → "HiHiHi"
stringTimes("Hi", 1) → "Hi"

64) Write a logic for this in JavaScript
Given a string and a non-negative int n, we'll say that the front of the string is the first 3 chars, or whatever is there if the string is less than length 3. Return n copies of the front;

frontTimes("Chocolate", 2) → "ChoCho"
frontTimes("Chocolate", 3) → "ChoChoCho"
frontTimes("Abc", 3) → "AbcAbcAbc"

65) Write a logic for this in JavaScript
Count the number of "xx" in the given string. We'll say that overlapping is allowed, so "xxx" contains 2 "xx".

countXX("abcxx") → 1
countXX("xxx") → 2
countXX("xxxx") → 3

66) Write a logic for this in JavaScript
Given a string, return the count of the number of times that a substring length 2 appears in the string and also as the last 2 chars of the string, so "hixxxhi" yields 1 (we won't count the end substring).

last2("hixxhi") → 1
last2("xaxxaxaxx") → 1
last2("axxxaaxx") → 2

67) Write a logic for this in JavaScript
Given two strings, append them together (known as "concatenation") and return the result. However, if the concatenation creates a double-char, then omit one of the chars, so "abc" and "cat" yields "abcat".

conCat("abc", "cat") → "abcat"
conCat("dog", "cat") → "dogcat"
conCat("abc", "") → "abc"

68) Write a logic for this in JavaScript
Given two strings, append them together (known as "concatenation") and return the result. However, if the strings are different lengths, omit chars from the longer string so it is the same length as the shorter string. So "Hello" and "Hi" yield "loHi". The strings may be any length.

minCat("Hello", "Hi") → "loHi"

minCat("Hello", "java") → "ellojava"
minCat("java", "Hello") → "javaello"

69) Write a logic for this in JavaScript
Given 2 int arrays, each length 2, return a new array length 4 containing all their elements.

plusTwo([1, 2], [3, 4]) → [1, 2, 3, 4]
plusTwo([4, 4], [2, 2]) → [4, 4, 2, 2]
plusTwo([9, 2], [3, 4]) → [9, 2, 3, 4]

70) Write a logic for this JavaScript
Given 2 ints, a and b, return their sum. However, "teen" values in the range 13..19 inclusive, are extra lucky. So if either value is a teen, just return 19.

teenSum(3, 4) → 7
teenSum(10, 13) → 19
teenSum(13, 2) → 19