JAVASCRIPT

JAVASCRIPT

- HTML defines a webpage's structure and content and CSS sets the formatting and appearance,
- JavaScript adds interactivity to a webpage and creates rich web applications.
- JavaScript is used in millions of Web pages to improve the design, validate forms, create cookies, and much more.
- JavaScript is the most popular scripting language on the internet, and works in all major browsers, such as Internet Explorer, Mozilla, Firefox, Netscape, Opera.
- JavaScript is a case-sensitive language.

ES6

- JavaScript was invented by Brendan Eich in 1995, and became an ECMA standard in 1997.
- ECMAScript is the official name of the language.
- ECMAScript versions have been abbreviated to ES1, ES2, ES3, ES5, and ES6.
- **ES6 or the ECMAScript 2015 is the 6th and major edition of the ECMAScript language specification standard**. It defines the standard for the implementation of JavaScript and it has become much more

- **JavaScript ES6 brings new syntax and new features to make your code more modern and more readable**.
- It allows you to write less code and do more.
- ES6 introduced several key features like const, let, arrow functions, template literals, default parameters and lot more.
- ECMA Script is generally used for client side scripting, and it is used for writing server applications and services by using Nodejs.

WHAT IS JAVASCRIPT?

- JavaScript was designed to add interactivity to HTML pages
- JavaScript is a scripting language (a scripting language is a lightweight programming language)
- A JavaScript consists of lines of executable computer code
- A JavaScript is usually embedded directly into HTML pages
- JavaScript is an interpreted language (means that scripts execute without preliminary compilation)
- Everyone can use JavaScript without purchasing a license

Are Java and JavaScript the Same?

- NO!
- Java and JavaScript are two completely different languages in both concept and design
- Java (developed by Sun Microsystems) is a powerful and much more complex programming language - in the same category as C and C++.

<script> tags

- Place the **<script>** tags, containing JavaScript, anywhere within your web page.
- The <script> tag alerts the browser program to start interpreting all the text between these tags as a script.
- A simple syntax of your JavaScript will appear as follows.

<script ...> JavaScript code </script>

Flexibility given to include JavaScript code anywhere in an HTML document.

However the most preferred ways to include JavaScript in an HTML file are as follows −

- Script in <head>...</head> section.
- Script in <body>...</body> section.
- Script in <body>...</body> and <head>...</head> sections.
- Script in an external file and then include in <head>...</head> section.

How to Put a JavaScript Into an HTML Page?

```
<html>
<body>
<script>
document.write("Hello World!")
</script>
</body>
</html>
```

save using .html ext.

JavaScript Display Possibilities

- JavaScript can "display" data in different ways:
- Writing into an HTML element, using innerHTML.
- Writing into the HTML output using document.write().
- Writing into an alert box, using window.alert().
- Writing into the browser console, using console.log().

```
<!DOCTYPE html>
<html>
<body>
<h2>My First Web Page</h2>
<p>My First Paragraph.</p>
<p id="demo"></p>
<script >
document.getElementById("demo").innerHTML = 5 + 6;
window.alert(5 + 6);
document.write(5 + 6);
Console.log(5+6);
</script>
</body>
</html>
```

<script src="myScript.js"></script>

For debugging purposes, you can use the console.log() method to display data.

Console

- The console is a panel that displays important messages, like errors, for developers.
- If we want to see things appear on our screen, we can print, or *log, to our console directly.*
- In JavaScript, the console keyword refers to an object, a collection of data and actions, that we can use in our code.
- One action, or method, that is built into the console object is the .log() method.
- When we write console.log() what we put

Ending Statements With a Semicolon?

- With traditional programming languages, like C++ and Java, each code statement has to end with a semicolon (;).

- Many programmers continue this habit when writing JavaScript, but in general, semicolons are **optional**!

- However, semicolons are required if you want to put more than one statement on a single line.

Comments in JavaScript

- JavaScript supports both C-style and C++-style comments, Thus −
- Single line //
- Multiple line /* and */

JavaScript Variables

- Variables are used to store data.
- A variable is a "container" for information you want to store.
- A variable's value can change during the script.
- You can refer to a variable by name to see its value or to change its value.
- Rules for variable names:
  - Variable names are case sensitive

Var

- There were a lot of changes introduced in the ES6 version of JavaScript in 2015.
- One of the biggest changes was two new keywords, let and const, to create, or *declare, variables.*
- *Prior to the ES6, programmers could only use the* var keyword to declare variables.

- There are 3 ways to declare a JavaScript variable:
- Using var
- Using let
- Using const

```
var x = 5;
var y = 6;
var z = x + y;
```

JavaScript Identifiers

All JavaScript **variables** must be **identified** with **unique names.**

These unique names are called **identifiers.**

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

Var

- var pi = 3.14;
  var person = "John Doe";
  var answer = 'Yes I am!';

general rules

- The general rules for constructing names for variables (unique identifiers) are:
- Names can contain letters, digits, underscores, and dollar signs.
- Names must begin with a letter
- Names can also begin with $ and _ (but we will not use it in this tutorial)
- Names are case sensitive (y and Y are different variables)
- Reserved words (like JavaScript keywords) cannot be used as names

Let

- The let keyword was introduced in ES6 (2015).
- Variables defined with let cannot be Redeclared.
- Variables defined with let must be Declared before use.
- Variables defined with let have Block Scope.

- let x = "John Doe";

  let x = 0;

  // SyntaxError: 'x' has already been declared

- *With var you can:*

Redeclaring the variable

```
<!DOCTYPE html>
<html>
<body>
<h2>Redeclaring a Variable Using let</h2>
<p id="demo"></p>
<script>
let  x = 10;
// Here x is 10
{
  let x = 2;
  // Here x is 2
}
// Here x is 10
document.getElementById("demo").innerHTML = x;
</script></body></html>
```

Redeclaring a variable with let, in another block, IS allowed:

```
<script>
let x = 2;   // Allowed

{
  let x = 3;   // Allowed
}

{
  let x = 4;   // Allowed
}
document.getElementById("demo").innerHTML = x; 2
</script>
```

const

- The const keyword was introduced in ES6 (2015).
- Variables defined with const cannot be Redeclared.
- Variables defined with const cannot be Reassigned.
- Variables defined with const have Block Scope.

```
const PI = 3.141592653589793;
   PI = 3.14;      // This will give an error
   PI = PI + 10;   // This will also give an error
```

- JavaScript const variables must be assigned a value when they are declared:
- const PI = 3.14159265359;..... correct
- const PI;
  PI = 3.14159265359; Incorrect

When to use JavaScript const?

As a general rule, always declare a variables with const unless you know that the value will change.

Always use const when you declare:

- A new Array
- A new Object
- A new Function
- A new RegExp

JavaScript Variable Scope

- The scope of a variable is the region of your program in which it is defined.
- JavaScript variables have only two scopes.
- **Global Variables** − A global variable has global scope which means it can be defined anywhere in your JavaScript code.
- **Local Variables** − A local variable will be visible only within a function where it is defined. Function parameters are always local to that function.

Block Scope

- Before ES6 (2015), JavaScript had only **Global Scope** and **Function Scope**.
- ES6 introduced two important new JavaScript keywords: let and const.
- These two keywords provide **Block Scope** in JavaScript.
- Variables declared inside a { } block cannot be accessed from outside the block:

```
{
    let x = 2;
}
// x can NOT be used here
```

Redeclaring a JavaScript variable with `var` is allowed anywhere in a program:

Data Types

Data types are the classifications we give to the different kinds of data that we use in programming.

In JavaScript, there are seven fundamental data types:

- Number: Any number, including numbers with decimals: 4, 8, 1516, 23.42.
- String: Any grouping of characters on our keyboard (letters, numbers, spaces, symbols, etc.) surrounded by single quotes: ' ... ' or double quotes " …".

Though we prefer single quotes. Some people

- Boolean: This data type only has two possible values— either true or false (without quotes). It's helpful to think of booleans as on and off switches or as the answers to a "yes" or "no" question.
- Null: This data type represents the intentional absence of a value, and is represented by the keyword null (without quotes).
- Undefined: This data type is denoted by the keyword undefined (without quotes). It also represents the absence of a value though it has a different use than null.
- Symbol: A newer feature to the language,

Object: Collections of related data.

The first 6 of those types are considered primitive data types. They are the most basic data types in the language. Objects are more complex.

- let length = 16; // Number
  let lastName = "Johnson"; // String
  let x = {firstName:"John", lastName:"Doe"}; // Object

- When adding a number and a string, JavaScript will treat the number as a string.

let x = 16 + "Volvo"; o/p 16Volvo

- JavaScript evaluates expressions from left to right. Different sequences can produce different results:

let x = 16 + 4 + "Volvo";

o/p 20Volvo

Note

- JavaScript does not make a distinction between integer values and floating-point values.
- let x1 = 34.00; // Written with decimals
  let x2 = 34; // Written without decimals
- let y = 123e5; // 12300000
  let z = 123e-5; // 0.00123

Hoisting

- In JavaScript, Hoisting is the default behavior of moving all the declarations at the top of the scope before code execution.
- Basically, it gives an advantage that no matter where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local.
- It allows to call functions before even writing them in code.
- **Note:** JavaScript only hoists declarations, not the initializations.

JavaScript Operators

Arithmetic Operators

Operator	Description	Example	Result
+	Addition	x=2 y=2 x+y	4
-	Subtraction	x=5 y=2 x-y	3
*	Multiplication	x=5 y=4 x*y	20
/	Division	15/5 5/2	3 2,5
%	Modulus (division remainder)	5%2 10%8 10%2	1 2 0
++	Increment	x=5 x++	x=6
--	Decrement	x=5 x--	x=4

Exponential **

Assignment Operators

Operator	Example	Is The Same As
=	x=y	x=y
+=	x+=y	x=x+y
-=	x-=y	x=x-y
*=	x*=y	x=x*y
/=	x/=y	x=x/y
%=	x%=y	x=x%y

Comparison Operators

Operator	Description	Example
==	is equal to	5==8 returns false
===	is equal to (checks for both value and type)	x=5 y="5" x==y returns true x===y returns false
!=	is not equal	5!=8 returns true
>	is greater than	5>8 returns false
<	is less than	5<8 returns true
>=	is greater than or equal to	5>=8 returns false
<=	is less than or equal to	5<=8 returns true

Logical Operators

Operator	Description	Example
&&	and	x=6 y=3 (x < 10 && y > 1) returns true
||	or	x=6 y=3 (x==5 || y==5) returns false
!	not	x=6 y=3 !(x==y) returns true

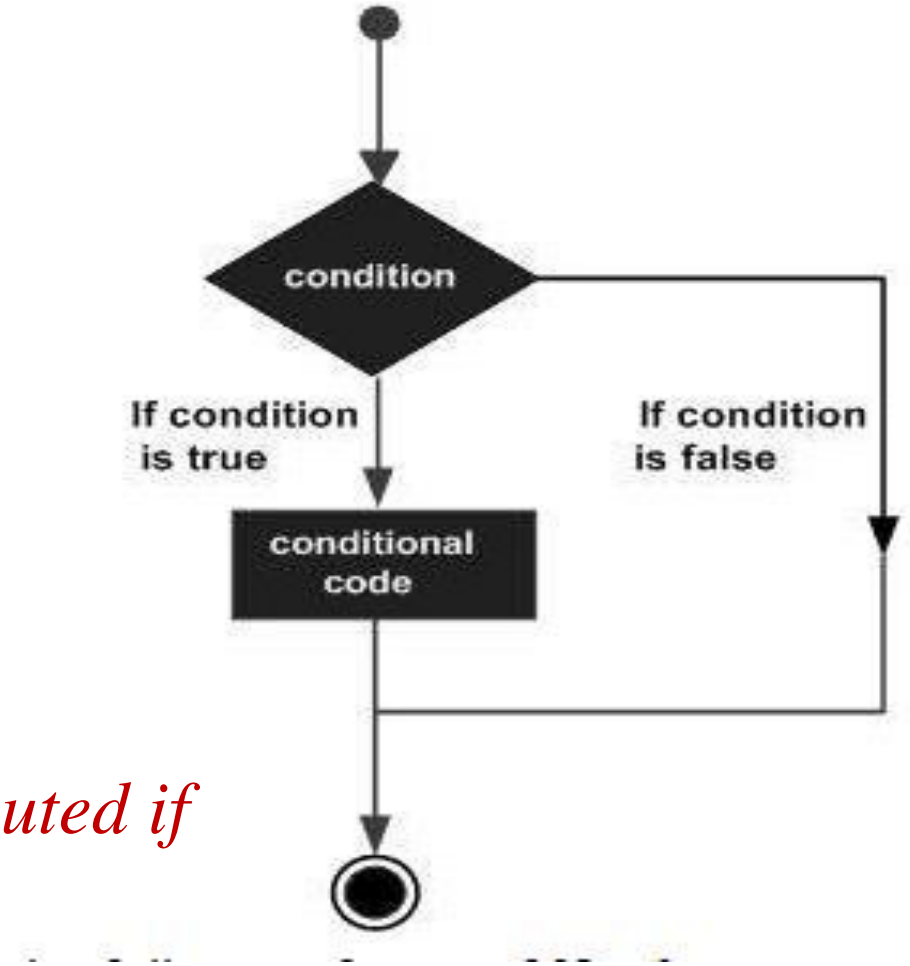
Conditional Statements

- JavaScript supports the following forms of **if..else** statement −

➢ if statement

➢ if...else statement

➢ if...else if... statemen




```
if (expression)
{
 Statement(s) to be executed if
expression is true
 }
```

if statement

```
var num = 5 ;
if (num>0)
{
console.log("number is positive")
}
```

if...else statement

```
if (expression)
{
Statement(s) to be executed if expression
    is true
 }
 else
{
Statement(s) to be executed if expression
    is false
 }
```

```
var num = 12;
if (num % 2 == 0) {
   console.log("Even");
} else {
   console.log("Odd");
}
```

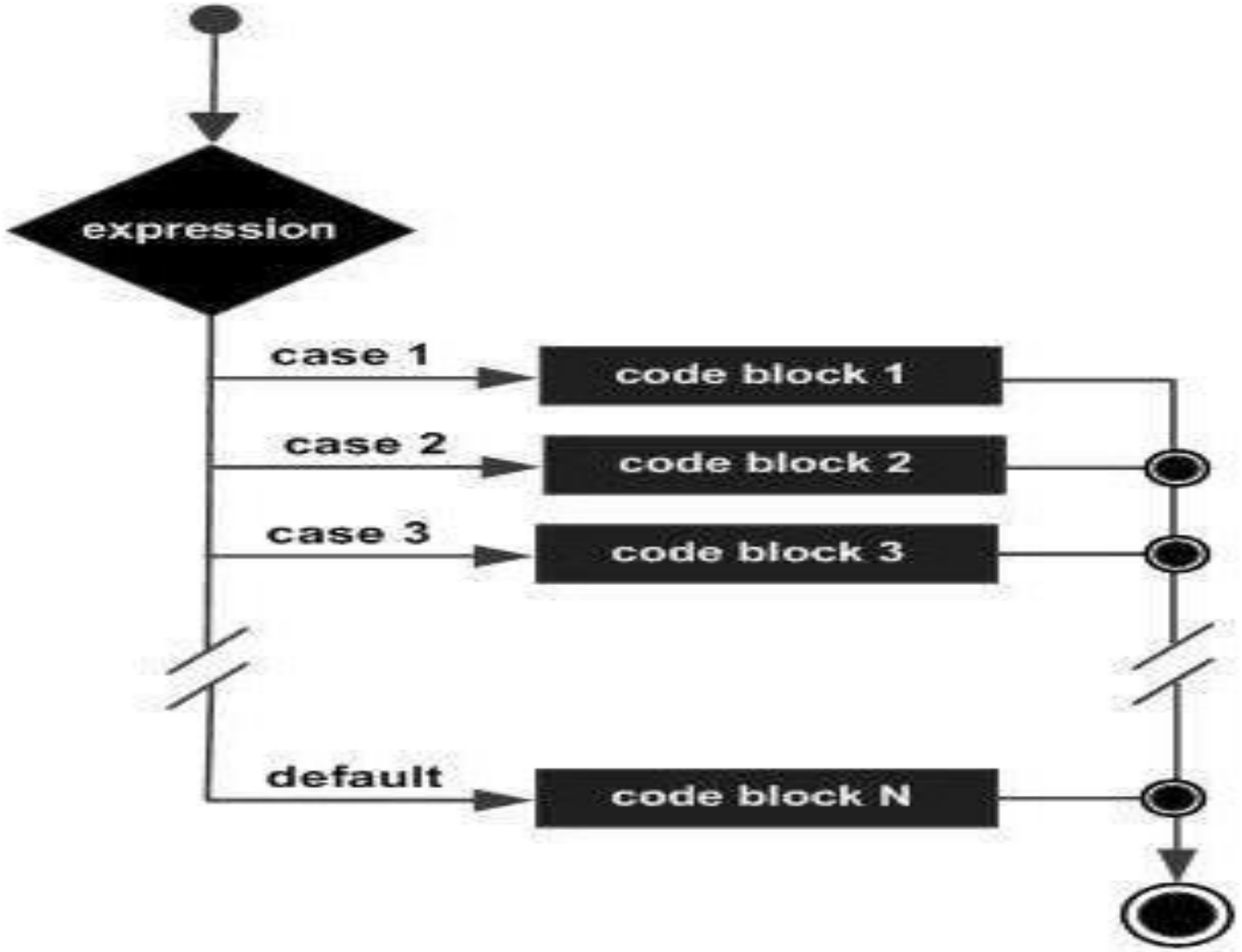
if...else if... statement

```
if (expression 1)
{
 Statement(s) to be executed if
    expression 1 is true
 }
 else if (expression 2)
{
 Statement(s) to be executed if
    expression 2 is true
 }
 else if (expression 3)
{
 Statement(s) to be executed if
    expression 3 is true
}
```

```
else
{
Statement(s) to be executed if
    no expression is true
}
```

```
var num=2
if(num > 0) {
   console.log(num+" is positive")
} else if(num < 0) {
   console.log(num+" is negative")
} else {
   console.log(num+" is neither positive nor
    negative")
}
```

Switch Case


expression
case 1
code block 1
case 2
code block 2
case 3
code block 3
default
code block N

Rules apply to a switch

- The following statement −
- There can be any number of case statements within a switch.
- The case statements can include only constants. It cannot be a variable or an expression.
- The data type of the variable_expression and the constant expression must match.
- Unless you put a break after each block of code, the execution flows into the next block.
- The case expression must be unique.

Syntax

```
switch (expression)
{
 case condition 1:
    statement(s)
    break;
 case condition 2:
    statement(s)
 break; ...
 case condition n:
    statement(s)
  break;
default: statement(s) }
```

```
var grade="A";
switch(grade) {
  case "A": {
    console.log("Excellent");
    break;
  }
  case "B": {
    console.log("Good");
    break;
  }
  case "C": {
    console.log("Fair");
    break;
  }
  case "D": {
    console.log("Poor");
    break;
  }
  default: {
    console.log("Invalid choice");
    break;
  }
}
```

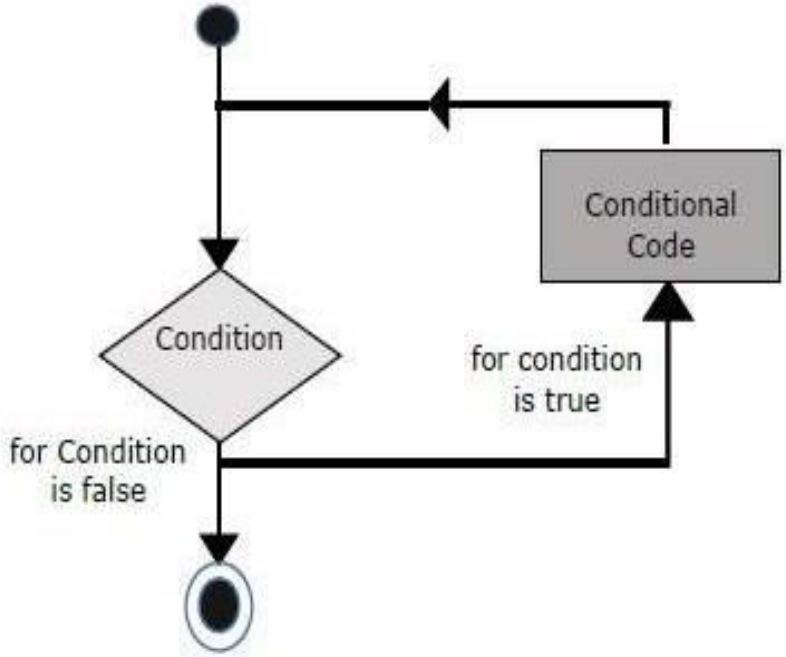
Loops

Definite Loop

A loop whose number of iterations are definite/fixed is termed as a **definite loop.**

The 'for loop' is an implementation of a **definite loop.**

For Loop



for (initialization; test condition; iteration statement)

{

Statement(s) to be executed if test condition is true

}

Factorial of num

```
var num = 5
var factorial=1;
for( let i = num ; i >= 1; i-- ) {
   factorial *= i ;
}
console.log(factorial);
```

- Multiple assignments and final expressions can be combined in a for loop, by using the comma operator (,).
- For example, the following for loop prints the first eight Fibonacci numbers –

```
for(let temp, i = 0, j = 1; j<30; temp = i, i = j, j = i + temp)
console.log(j);
```

for...in loop

- The for...in loop is used to loop through an object's properties.
- Following is the syntax of 'for...in' loop.

```
for (variablename in object)
{ statement or block to execute }
```

- In each iteration, one property from the object is assigned to the variable name and this loop continues till all the properties of the object are exhausted.

```
var obj = {a:1, b:2, c:3};

for (var prop in obj) {
   console.log(obj[prop]);
}
```

1
2
3

for…of

- The for…of loop is used to iterate iterables instead of object literals.
- Following is the syntax of 'for…of' loop.

```
for (variablename of object)
{ statement or block to execute }
```

Eg:

```
for (let val of[12 , 13 , 123])
{ console.log(val) }
```

```
12
13
123
```

Example for...of

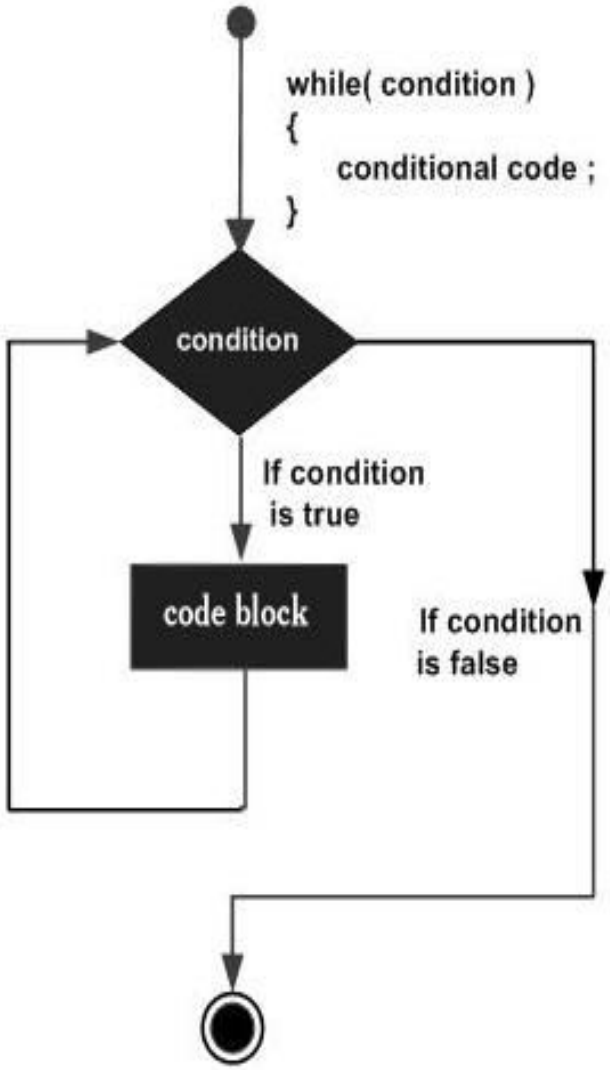
```
var fruits = ['Apple', 'Banana', 'Mango', 'Orange'];
for(let value of fruits)
{
  console.log(value);
}
```

Indefinite Loop

- An indefinite loop is used when the number of iterations in a loop is indeterminate or unknown.

- While loop
- Do… while loop

While loop

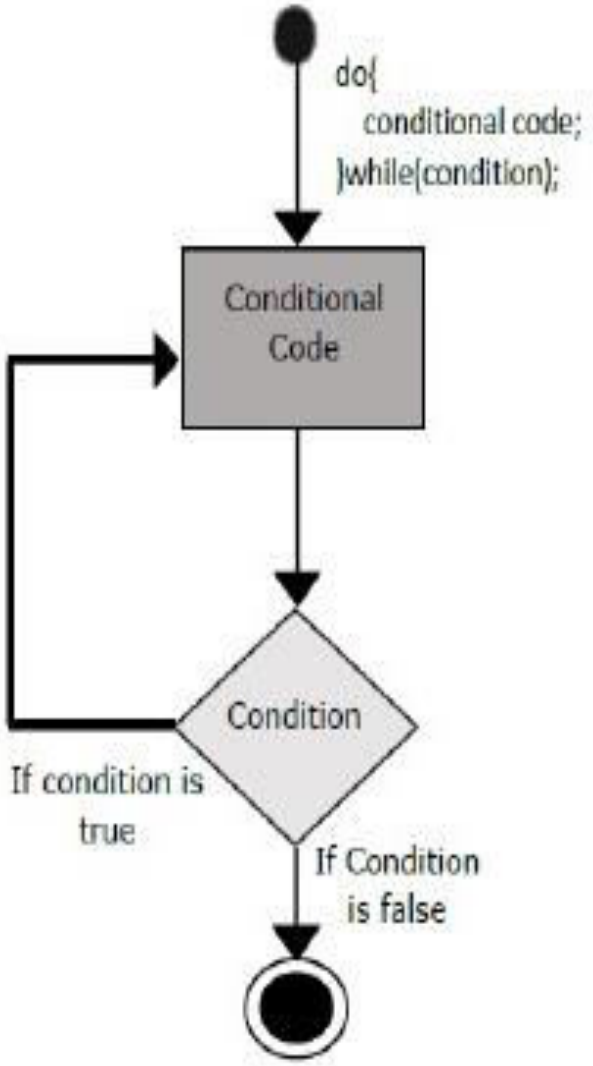



```
while (expression)
{
 Statement(s) to be executed
if expression is true
}
```

```
var num = 5;
var factorial = 1;
while(num >=1) {
   factorial = factorial * num;
   num--;
}
console.log("The factorial is "+factorial);
```

do…while loop

- The **do…while** loop is similar to the while loop except that the **do…while** loop doesn't evaluate the condition for the first time the loop executes.



```
do {
   Statement(s) to be executed;
} while (expression);
```

```
var n = 10;
do {
   console.log(n);
   n--;
} while(n> = 0);
```

Break

```
var i = 1
while(i< = 10) {
   if (i % 5 == 0) {
      console.log("The first multiple of 5 between 1 and
    10 is : "+i)
      break //exit the loop if the first multiple is found
   }
   i++
}
```

Continue

```
var num = 0
var count = 0;
for(num = 0;num< = 20;num++) {
   if (num % 2 == 0) {
      continue
   }
   count++
}
console.log(" The count of odd values between 0
    and 20 is: "+count)
```

Node.js

- Node.js is an open source server environment.
- Node.js allows you to run JavaScript on the server.
- Node.js is free
- Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
- **Node.js uses asynchronous programming!**

Asynchronous programming

- Asynchronous programming is a design pattern which ensures the non-blocking code execution.
- Non blocking code do not prevent the execution of piece of code. In general if we execute in Synchronous manner i.e one after another we unnecessarily stop the execution of those code which is not depended on the one you are executing.
- Asynchronous does exactly opposite, asynchronous code executes without having any dependency and no order. This

Node.js

- Node.js can generate dynamic page content
- Node.js can create, open, read, write, delete, and close files on the server
- Node.js can collect form data
- Node.js can add, delete, modify data in your database.
- Node.js files contain tasks that will be executed on certain events
- A typical event is someone trying to access a port on the server
- Node.js files must be initiated on the server before having any effect
- Node.js files have extension ".js"

Functions

- A JavaScript function is a block of code designed to perform a particular task.
- **Functions** are the building blocks of readable, maintainable, and reusable code.
- Functions are defined using the function keyword.
- A JavaScript function is executed when "something" invokes it (calls it).

JavaScript Function Syntax

- A JavaScript function is defined with the function keyword, followed by a **name**, followed by parentheses **()**.

```
function name(parameter1, parameter2,
  parameter3) {

    // code to be executed
  }
```

```
//define a  function
function test() {
   console.log("function called")
}
//call the function
test()
```

- Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).
- The parentheses may include parameter names separated by commas: ***(parameter1, parameter2, ...)***
- The code to be executed, by the function, is placed inside curly brackets: **{}**

- *A Function is much the same as a Procedure or a Subroutine, in other programming languages.*

- Function **arguments** are the **values** received by the function when it is invoked.
- Inside the function, the arguments (the parameters) behave as local variables.
- Function Invocation
- When an event occurs (when a user clicks a button)
- When it is invoked (called) from JavaScript code
- Automatically (self invoked)

Function Return

- When JavaScript reaches a return statement, the function will stop executing.
- If the function was invoked from a statement, JavaScript will "return" to execute the code after the invoking statement.
- Functions often compute a **return value**.
- The return value is "returned" back to the "caller":

Why Functions?

- You can reuse code: Define the code once, and use it many times.
- You can use the same code many times with different arguments, to produce different results.

Classification of Functions

- Functions may be classified as **Returning** and **Parameterized** functions.
- Returning functions
- Functions may also return the value along with control, back to the caller.
- Such functions are called as returning functions.
- Following is the syntax for the returning function.

```
function function_name() {
//statements
return value;
}
```

- A returning function must end with a return statement.
- A function can return at the most one value. In other words, there can be only one return statement per function.
- The return statement should be the last statement in the function.

```
function retStr() {
   return "hello world!!!"
}
var val = retStr()
console.log(val)
```

Parameterized functions

- Parameters are a mechanism to pass values to functions.
- Parameters form a part of the function's signature.
- The parameter values are passed to the function during its invocation.
- Unless explicitly specified, the number of values passed to a function must match the number of parameters defined.
- Following is the syntax with eg defining a parameterized function.

```
function add( n1,n2) {
   var sum = n1 + n2
   console.log("The sum of the values entered
    "+sum)
}
add(12,13)
```

Default function parameters

- In ES6, a function allows the parameters to be initialized with default values, if no values are passed to it or it is undefined.
- The same is illustrated in the following code.

```
function add(a, b = 1) {
   return a+b;
}
console.log(add(4))
```

Default is overwritten

```
function add(a, b = 1) {
   return a + b;
}
console.log(add(4,2))
```

Eg

```
function addTwoNumbers(first ,second = 10){
    console.log('first parameter is :',first)
    console.log('second parameter is :',second)
    return first+second;
  }

  console.log("case 1 sum:",addTwoNumbers(20)) // no value
  console.log("case 2 sum:",addTwoNumbers(2,3))
  console.log("case 3 sum:",addTwoNumbers())
  console.log("case 4 sum",addTwoNumbers(1,null))//null
    passed
  console.log("case 5 sum",addTwoNumbers(3,undefined))
```

- first parameter is : 20
- second parameter is : 10
- case 1 sum: 30
- first parameter is : 2
- second parameter is : 3
- case 2 sum: 5
- first parameter is : undefined
- second parameter is : 10
- case 3 sum: NaN
- first parameter is : 1
- second parameter is : null
- case 4 sum 1
- first parameter is : 3
- second parameter is : 10
- case 5 sum 13

Anonymous Function

- Functions that are not bound to an identifier (function name) are called as anonymous functions.
- These functions are dynamically declared at runtime.
- Anonymous functions can accept inputs and return outputs, just as standard functions do.
- An anonymous function is usually not accessible after its initial creation.

- Variables can be assigned an anonymous function. Such an expression is called a **function expression**.
- Following is the syntax for anonymous function.
    - var res = function([arguments]) { ... }

Eg

```
var f = function(){ return "hello"} console.log(f())
```

Example – Anonymous Parameterized Function

```
var func = function(x,y){ return x*y };
function product() {
   var result;
   result = func(10,20);
   console.log("The product : "+result)
}
product()
```

The Function Constructor

- The function statement is not the only way to define a new function;
- Define function dynamically using Function() constructor along with the new operator.
- syntax to create a function using Function() constructor along with the new operator.
  - var variablename = new Function(Arg1, Arg2..., "Function Body");

- The Function() constructor expects any number of string arguments.
- The last argument is the body of the function – it can contain arbitrary JavaScript statements, separated from each other by semicolons.
- The Function() constructor is not passed any argument that specifies a name for the function it creates.

Eg

```
var func = new Function("x", "y", "return x*y;");
function product() {
   var result;
   result = func(10,20);
   console.log("The product : "+result)
}
product()
```

Example – Recursion

```
function factorial(num) {
   if(num <= 0) {
      return 1;
   } else {
      return (num * factorial(num-1)  )
   }
}
console.log(factorial(6))
```

Arrow Function

- ES6 introduces the concept of **arrow function** to simplify the usage of **anonymous function**.
- There are 3 parts to an arrow function which are as follows −
- **Parameters** − An arrow function may optionally have parameters
- **The fat arrow notation (=>)** − It is also called as the goes to operator
- **Statements** − Represents the function's instruction set

Syntax

```
//Arrow function that points to a single line of
   code
()=>some_expression
```

OR

```
//Arrow function that points to a block of code
()=> {  //some statements   }
```

OR

```
//Arrow function with parameters
(param1,param2)=>{  //some statement   }
```

JavaScript Arrow Function

- Arrow functions allow us to write shorter function syntax:

```
hello = function() {
  return "Hello World!";
}
```

Before

```
hello = () => {
  return "Hello World!";
}
```

Using Arrow fun

Note ..

- If the function has only one statement, and the statement returns a value, you can remove the brackets *and* the return keyword:

```
hello = () => "Hello World!";
```

Eg

```
const add = (n1,n2) => n1+n2
  console.log(add(10,20))

  const isEven = (n1) => {
    if(n1%2 == 0)
      return true;
    else
      return false;
  }
  console.log(isEven(10))
```

```
// function expression
  var myfun1 = function show() {
 console.log("It is a Function Expression");
}
  // Anonymous function
  var myfun2 = function () {
    console.log("It is an Anonymous Function");

   }
   //Arrow function
  var myfun3 = () => {
```

Passing parameters

- hello = (val) => "Hello " + val;
- if have only one parameter, skip the parentheses as well:
- hello = val => "Hello " + val;

```
var hello;
hello = val => "Hello " + val;
console.log(hello("Universe!"));
```

JavaScript Events

- **Events** are a part of the Document Object Model (DOM) and every HTML element contains a set of events that can trigger JavaScript Code.
- An event is an action or occurrence recognized by the software.
- It can be triggered by a user or the system.
- Some common examples of events include a user clicking on a button, loading the web page, clicking on a hyperlink and so on.

JS Events

- Event Handlers
- On the occurrence of an event, the application executes a set of related tasks.
- The block of code that achieves this purpose is called the **eventhandler**.
- Event handlers can be used to handle and verify user input, user actions, and browser actions:
- Things that should be done every time a page loads
- Things that should be done when the page is closed

```
<button onclick="document.getElementById('demo').innerHTML =
    Date()">The time is?</button>
```

```
<button onclick="this.innerHTML = Date()">The time is?</button>
```

```
<button onclick="displayDate()">The time is?</button>
```

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

```
<html>
   <head>
      <script>
         function sayHello() {
            document.write ("Hello World")
         }
      </script>
   </head>

   <body>
      <p> Click the following button and see result</p>
      <input type = "button" onclick = "sayHello()"
    value = "Say Hello" />
   </body>
</html>
```

onmousemove

```
<html>
  <head>
    <script >
      function over() {
        document.write ("Mouse Over");
      }
      function out() {
        document.write ("Mouse Out");
      }
    </script>    </head>
```

```
<body>
    <p>Bring your mouse inside the division
  to see the result:</p>
    <div onmouseover = "over()"
  onmouseout = "out()">
        <h2> This is inside the division </h2>
    </div>    </body> </html>
```

JavaScript Classes

- **Object Orientation** is a software development paradigm that follows real-world modelling.
- Object Orientation, considers a program as a collection of objects that communicates with each other via mechanism called **methods**.
- ES6 supports these object-oriented components too.
- **Object** − An object is a real-time representation of any entity. According to Grady Brooch, every object is said to have 3 features −

OO concepts

- **State** − Described by the attributes of an object.
- **Behavior** − Describes how the object will act.
- **Identity** − A unique value that distinguishes an object from a set of similar such objects.

- **Class** − A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object.
- **Method** – Methods facilitate

JavaScript Classes

- A car is an object that has data (make, model, number of doors, Vehicle Number, etc.) and functionality (accelerate, shift, open doors, turn on headlights, etc.)
- JavaScript Class Syntax
- Use the keyword class to create a class.
- Always add a method named constructor():

```
class Class_name {
  constructor() { ... }
}
```

- Syntax: Class Expressions

```
var var_name = new Class_name { }
```

- A class definition can include the following −
- **Constructors** − Responsible for allocating memory for the objects of the class.
- **Functions** − Functions represent actions an object can take. They are also at times referred to as methods.
- These components put together are termed as the data members of the class.
- **Note** − A class body can only contain methods, but not data properties.

Example: Declaring a class

```
class Polygon {
    constructor(height, width)
    {
        this.height = height;
        this.width = width;
    }
}
```

Class Expression:

```
var Polygon = class Polygon {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}
```

Hoisting

- An important difference between **function declarations** and **class declarations** is that
- while functions can be called in code that appears before they are defined,
- classes must be defined before they can be constructed.

Creating Objects

- To create an instance of the class, use the new keyword followed by the class name.
- Following is the syntax for the same.
  - var object_name= new class_name([arguments])
- Where,
- The new keyword is responsible for instantiation.
- The right hand side of the expression invokes the constructor. The constructor should be passed values if it is parameterized.
- Example: Instantiating a class

Accessing Functions

- A class's attributes and functions can be accessed through the object.
- Use the '.' **dot notation** to access the data members of a class.

```
//accessing a function
obj.function_name()
```

```
class Polygon {
   constructor(height, width) {
      this.h = height;
      this.w = width;
   }
   test() {
      console.log("The height of the polygon: ",
    this.h)
      console.log("The width of the polygon: ",this.
    w)
   } }
//creating an instance
var polyObj = new Polygon(10,20);
```

Class Inheritance

- ES6 supports the concept of **Inheritance**.
- Inheritance is the ability of a program to create new entities from an existing entity - here a class.
- The class that is extended to create newer classes is called the **parent class/super class**.
- The newly created classes are called the **child/sub classes**.
- Inheritance is useful for code reusability: reuse properties and methods of an existing class when you create a new class.

Following is the syntax for the same.

class child_class_name extends parent_class_name

```
class Shape {
   constructor(a) {
      this.Area = a
   }
}
class Circle extends Shape {
   disp() {
      console.log("Area of the circle:  "+this.Area)
   }
}
var obj = new Circle(223);
obj.disp()
```

- Inheritance can be classified as –
- **Single** – Every class can at the most extend from one parent class.
- **Multiple** – A class can inherit from multiple classes. **ES6 doesn't** support multiple inheritance.
- **Multi-level** – Consider the following example.

```
class Root {
   test() {
      console.log("call from parent class")
   }
}
class Child extends Root {}
class Leaf extends Child

//indirectly inherits from Root by virtue of
   inheritance {}
var obj = new Leaf();
obj.test()
```

The Super Keyword

- ES6 enables a child class to invoke its parent class data member.
- This is achieved by using the **super** keyword.
- The super keyword is used to refer to the immediate parent of a class.

```
class PrinterClass {
   doPrint() {
      console.log("doPrint() from Parent called…")
   }
}
class StringPrinter extends PrinterClass {
   doPrint() {
      super.doPrint()
      console.log("doPrint() is printing a string…")
   }
}
var obj = new StringPrinter()
obj.doPrint()
```

Setters and Getters

Setters

- A setter function is invoked when there is an attempt to set the value of a property.
- The **set keyword** is used to define a setter function.
- The syntax for defining a setter function is given below −

{set prop(val) { . . . }}

{set [expression](val) { . . . }}

- **prop** is the name of the property to bind to the given function.
- **val** is an alias for the variable that holds the value attempted to be assigned to property.

```
class Student {

    constructor(rno,fname,lna
    me){
        this.rno = rno
        this.fname = fname
        this.lname = lname
        console.log('inside
    constructor')
      }
      set rollno(newRollno){
        console.log("inside
    setter")
        this.rno = newRollno
      }   }
```

```
let s1 = new Student(101,'Sachin','Te
  console.log(s1)
  //setter is called
  s1.rollno = 201
  console.log(s1)
```

The following example shows how to use an **expression** as a property name with a **setter function**.

```
let expr = 'name';
    let obj = {
    fname: 'Sachin',
    set [expr](v) { this.fname = v; }
  };
  console.log(obj.fname);
  obj.name = 'John';
  console.log(obj.fname);
```

Getters

- A **getter function** is invoked when there is an attempt to fetch the value of a property.
- The **get keyword** is used to define a getter function.
- The syntax for defining a getter function is given below −
  - **{get prop() { ... } }**
  - **{get [expression]() { ... } }**
- **prop** is the name of the property to bind to the given function.
- **expression** − Starting with ES6, can also use expressions as a property name to bind to the given function.

```
class Student {
    constructor(rno,fname,lname){
      this.rno = rno
      this.fname = fname
      this.lname = lname
      console.log('inside constructor')
    }
    get fullName(){
      console.log('inside getter')
      return this.fname + " - "+this.lname
    }
  }
  let s1 = new Student(101,'Sachin','Tendulkar')
  console.log(s1)
  //getter is called
  console.log(s1.fullName)
```

Expression

```
let expr = 'name';
  let obj = {
    get [expr]() { return 'Sachin'; }
  };
  console.log(obj.name);
```

Class Inheritance and Method Overriding

Method Overriding is a mechanism by which the child class redefines the superclass method. The following example illustrates the same

```
class PrinterClass {
   doPrint() {
      console.log("doPrint() from Parent called… ");
   }
}
class StringPrinter extends PrinterClass {
   doPrint() {
      console.log("doPrint() is printing a string…");
   }
}
```

Introduction to Iterator

- Iterator is an object which allows us to access a collection of objects one at a time.
- The following built-in types are by default iterable −
- String
- Array
- Map
- Set
- An object is considered **iterable**, if the object implements a function whose key is **[Symbol.iterator]** and returns an iterator.
- A for...of loop can be used to iterate a

Example

- The following example declares an array, marks, and iterates through it by using a **for..of** loop.

```
let marks = [10,20,30]
  //check iterable using for..of
  for(let m of marks){
    console.log(m);
  }
```

- The following example declares an array, marks and retrieves an iterator object.
- The [Symbol.iterator]() can be used to retrieve an iterator object.
- The next() method of the iterator returns an object with 'value' and 'done' properties .
- 'done' is Boolean and returns true after reading all items in the collection.

```
let marks = [10,20,30]
  let iter = marks[Symbol.iterator]();
  console.log(iter.next())
  console.log(iter.next())
  console.log(iter.next())
  console.log(iter.next())
```

Custom Iterable

- Certain types in JavaScript are iterable (E.g. Array, Map etc.) while others are not (E.g. Class).
- JavaScript types which are not iterable by default can be iterated by using the iterable protocol.
- The following example defines a class named **CustomerList** which stores multiple customer objects as an array.
- Each customer object has firstName and lastName properties.
- To make this class iterable, the class must implement **[Symbol.iterator]()** function.
- This function returns an iterator object.

```
//user defined iterable
  class CustomerList {
    constructor(customers){
      //adding customer objects to an array
      this.customers = [].concat(customers)
    }
    //implement iterator function
    [Symbol.iterator](){
      let count=0;
      let customers = this.customers
      return {
        next:function(){
        //retrieving a customer object from the array
          let customerVal = customers[count];
          count+=1;
          if(count<=customers.length){
            return {
              value:customerVal,
              done:false
            }
          }
          //return true if all customer objects are iterated
          return {done:true}
        }
      }
    }
```

```
 //create customer objects
  let c1={
    firstName:'Sachin',
    lastName:'Tendulkar'
  }
  let c2={
    firstName:'Rahul',
    lastName:'Dravid'
  }
  //define a customer array and initialize it let
customers=[c1,c2]
  //pass customers to the class' constructor
  let customersObj = new
CustomerList(customers);
  //iterating using for..of
  for(let c of customersObj){
    console.log(c)
  }
  //iterating using the next() method
  let iter = customersObj[Symbol.iterator]();
  console.log(iter.next())
  console.log(iter.next())
  console.log(iter.next())
```

Generator

- ES6 introduces functions known as Generator which can stop midway and then continue from where it stopped.
- A generator prefixes the function name with an asterisk * character and contains one or more **yield** statements.
- The **yield** keyword returns an iterator object.

Syntax

```
function * generator_name() {
yield value1
...
yield valueN
}
```

Example

- The example defines a generator function **getMarks** with three yield statements.
- Unlike normal functions, the **generator function getMarks()**,when invoked, doesn't execute the function but returns an iterator object that helps to execute code inside the generator function.
- On the first call to **markIter.next()** operations in the beginning would run and the yield statement pauses the execution of the

```
//define generator function
   function * getMarks(){
      console.log("Step 1")
      yield 10
      console.log("Step 2")
      yield 20
      console.log("Step 3")
      yield 30
      console.log("End of function")
   }
   //return an iterator object
      let markIter = getMarks()
   //invoke statements until first yield
      console.log(markIter.next())
   //resume execution after the last yield until second yield expression
      console.log(markIter.next())
   //resume execution after last yield until third yield expression
      console.log(markIter.next())
      console.log(markIter.next()) // iteration is completed;no value is
     returned
```

Example

- The following example creates an infinite sequence of even numbers through * evenNumberGenerator generator function.
- We can iterate through all even numbers by using next() or using for of loop as shown below

```
function * evenNumberGenerator(){
    let num = 0;
    while(true){
      num+=2
      yield num
    }
}
// display first two elements
  let iter =
evenNumberGenerator();
  console.log(iter.next())
  console.log(iter.next())
  //using for of to iterate till
12
  for(let n of
evenNumberGenerator()){
```

JavaScript Promises

- In JavaScript, a promise is an object that returns a value which you hope to receive in the future, but not now.
- Because the value will be returned by the promise in the future, the promise is very well-suited for handling asynchronous operations.
- Suppose that you promise to complete learning JavaScript by next month.
- You don't know if you will spend your time and effort to learn JavaScript until next

Promise

- A promise has three states:
- **Pending**: you don't know if you will complete learning JavaScript by the next month.
- **Fulfilled**: you complete learning JavaScript by the next month.
- **Rejected**: you don't learn JavaScript at all.
- A promise starts in the pending state which indicates that the promise hasn't completed.
- It ends with either fulfilled (successful) or rejected (failed) state.

Promise Syntax

- The Syntax related to promise is mentioned below where,
- **p** is the promise object,
- **resolve** is the function that should be called when the promise executes successfully and
- **reject** is the function that should be called when the promise encounters an error.

```
let p = new Promise(function(resolve,reject){
  let workDone = true; // some time consuming work
    if(workDone){

    //invoke resolve function passed

    resolve('success promise completed')
  }
  else{
    reject('ERROR , work could not be completed')
  }
})
```

- The Promise constructor accepts a function as an argument. This function is called the executor.
- The executor accepts two functions with the names, by convention, resolve() and reject().
- When you call the new Promise(executor), the executor is called automatically.
- Inside the executor, you manually call the resolve() function if the executor is

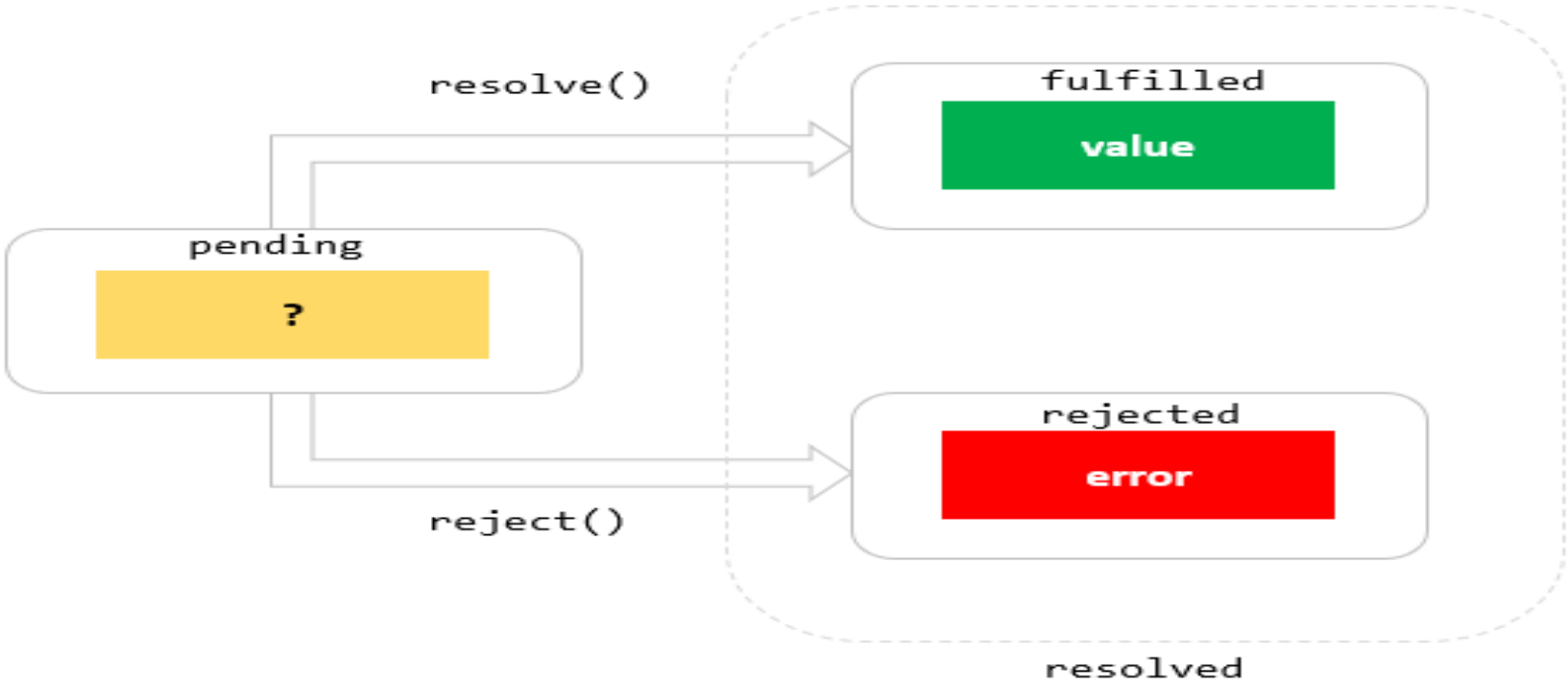
To see the pending state of the promise, we wrap the code of the executor in the setTimeout() function:

```
let completed = true;

let learnJS = new Promise(function (resolve, reject) {
    setTimeout(() => {
        if (completed) {
            resolve("I have completed learning JS.");
        } else {
            reject("I haven't completed learning JS yet.");
        }
    }, 3 * 1000);
});
```

Now, you see that the promise starts with the pending state with the value is undefined.

The promise value will be returned later once the promise is completed.



•Once the promise reaches either fulfilled state or rejected state, it stays in that state and can't switch.

•In other words, a promise cannot go from the fulfilled state to the rejected state and vice versa. It also cannot go back from the fulfilled state or rejected state to the pending state.

•If the promise reaches fulfilled state or rejected state, the promise is resolved.

Promise Methods

- The Promise methods are used to handle the rejection or resolution of the **Promise** object.
- Description of Promise methods.
- **.then()**
- This method invokes when a Promise is either fulfilled or rejected.
- This method can be chained for handling the rejection or fulfillment of the Promise.
- It takes two functional arguments for **resolved** and **rejected**.
- The first one gets invoked when the Promise is fulfilled, and the second one (which is optional) gets invoked when the Promise is rejected.

- **.catch()**
- It is a great way to handle failures and rejections.
- It takes only one functional argument for handling the errors.

- **.resolve()**
- It returns a new Promise object, which is resolved with the given value.
- If the value has a **.then()** method, then the returned Promise will follow that **.then()** method adopts its eventual state; otherwise, the returned Promise will be fulfilled with value.

- **.reject()**
- It returns a rejected **Promise** object with the given value.
- **.all()**
- It takes an array of Promises as an argument.
- This method returns a **resolved Promise** that fulfills when all of the Promises which are passed as an iterable have been fulfilled.
- **.race()**
- This method is used to return a resolved Promise based on the first referenced Promise that resolves.

Example

- The example given below shows a function **add_positivenos_async()** which **adds two numbers asynchronously.**
- The promise is resolved if positive values are passed.
- The promise is rejected if negative values are passed.

```
function add_positivenos_async(n1, n2) {
    let p = new Promise(function (resolve, reject) {
      if (n1 >= 0 && n2 >= 0) {
        //do some complex time consuming work
        resolve(n1 + n2)
      }
      else
        reject('NOT_Postive_Number_Passed')
      })
      return p;
  }
  add_positivenos_async(10, 20)
    .then(successHandler) // if promise resolved
    .catch(errorHandler);// if promise rejected
```

```
add_positivenos_async(-10, -20)
    .then(successHandler) // if promise resolved
    .catch(errorHandler);// if promise rejected

  function errorHandler(err) {
    console.log('Handling error', err)
  }
  function successHandler(result) {
    console.log('Handling success', result)
  }

  console.log('end')
```

Promises Chaining

- **Promises chaining** can be used when we have a sequence of **asynchronous tasks** to be done one after another.
- Promises are chained when a promise depends on the result of another promise.
- This is shown in the example below
- ***add_positivenos_async() function*** *adds two numbers asynchronously and rejects if negative values are passed.*
- *The result from the current asynchronous function call is passed as parameter to the subsequent function calls.*
- *Note each* ***then()*** *method has a return statement.*

```
function add_positivenos_async(n1, n2) {
    let p = new Promise(function (resolve,
  reject) {
        if (n1 >= 0 && n2 >= 0) {
            //do some complex time consuming
  work
            resolve(n1 + n2)
        }
        else

  reject('NOT_Postive_Number_Passed')
    })
```

```
add_positivenos_async(10,20)
  .then(function(result){
     console.log("first result",result)
     return add_positivenos_async(result,result)
  }).then(function(result){
  console.log("second result",result)
     return add_positivenos_async(result,result)
  }).then(function(result){
     console.log("third result",result)
  })

  console.log('end')
```

- **Promises** are a clean way to implement async programming in JavaScript (ES6 new feature).
- Prior to promises, Callbacks were used to implement async programming.

JavaScript Callbacks

- A callback is a function passed as an argument to another function
- This technique allows a function to call another function
- A callback function can run after another function has finished

```
function notifyAll(fnSms, fnEmail) {
    console.log('starting notification process');
    fnSms();
    fnEmail();
}
notifyAll(function() {
    console.log("Sms send ..");
},
function() {
    console.log("email send ..");
});
console.log("End of script");
//executes last or blocked by other methods
```

*In the **notifyAll()** method shown above, the notification happens by sending SMS and by sending an e-mail. Hence, the invoker of the notifyAll method has to pass two functions as parameters. Each function takes up a single responsibility like sending SMS and sending an e-mail.*

- Promises are "Continuation events" and they help you execute the multiple async operations together in a much cleaner code style.

What is fetch?

- Fetch is a newer Promise based API that is integrated into modern Browsers.
- Fetch makes it easier to make web requests and handle responses than with the older XMLHttpRequest, which often requires additional logic (for example, for handling redirects)
- The Fetch API is a simple interface for fetching resources.
- Fetch makes it easier to make web requests and handle responses than with the older XMLHttpRequest, which often requires

```
if (!('fetch' in window)) {
  console.log('Fetch API not found, try including
    the polyfill');
  return;
}
// We can safely use fetch from now on
```

fetch

- The **fetch()** method in JavaScript is used to request to the server and load the information in the webpages.
- The request can be of any APIs that returns the data of the format JSON or XML.
- This method returns a promise.
- **Syntax:**

fetch(url, options)

- **URL:** It is the URL to which the request is to be made.
- **Options:** It is an array of properties. It is an **optional** parameter.
- **Return Value:** It returns a promises whether it is resolved or not.
- The return data can be of the format JSON or XML. It can be the array of objects or simply a single object.

Eg: making get request using fetch

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content=
    "width=device-width, initial-scale=1.0">
      <title>JavaScript | fetch() Method</title>
</head>
  <body>
    <script>
          // API for get requests
let fetchRes = fetch("https://jsonplaceholder.typicode.com/todos/1");
```

```
// fetchRes is the promise to resolve
        // it by using.then() method
        fetchRes.then(res =>
            res.json()).then(d => {
                console.log(d)
            })
    </script>
</body>

</html>
```

Making Post Request using Fetch: Post requests can be made using fetch by giving options as given below:

```
let options = {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(data)
}
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content= "width=device-width,
     initial-scale=1.0">
      <title>JavaScript | fetch() Method</title>
</head>
  <body>
    <script>
        user = {
            "name": “TEIT",
            "age": "23"
        }
```

```
// Options to be given as parameter in fetch for making requests  other then GET
    let options = {
        method: 'POST',
        headers: {
            'Content-Type':
                'application/json;charset=utf-8'
        },
        body: JSON.stringify(user)
    }
     // Fake api for making post requests
    let fetchRes =
fetch(http://dummy.restapiexample.com/api/v1/create,options);
    fetchRes.then(res =>
        res.json()).then(d => {
            console.log(d)
        })    </script></body>  </html>
```