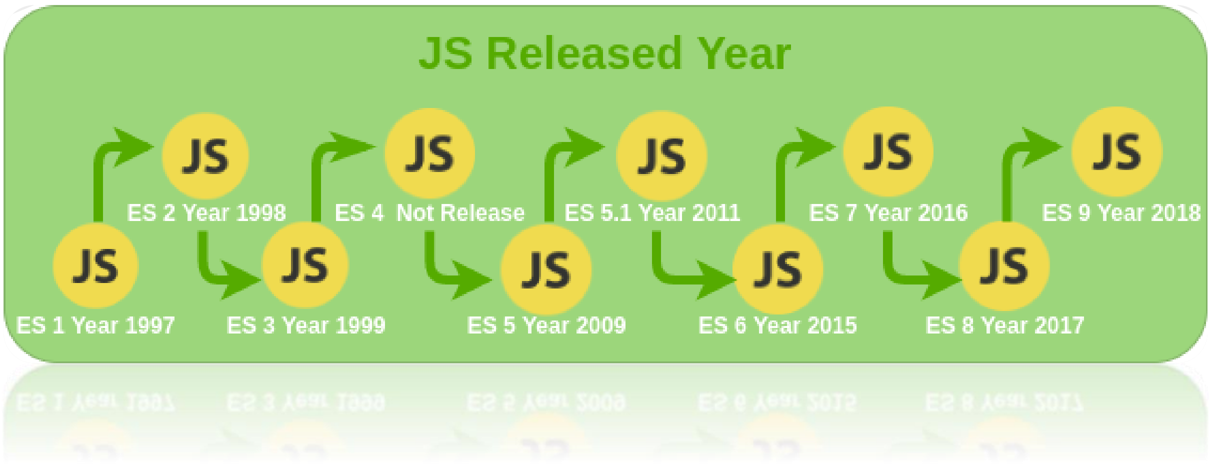
History of JavaScript: It was created in 1995 by **Brendan Eich** while he was an engineer at Netscape. It was originally going to be named LiveScript but was renamed. Unlike most programming languages, the JavaScript language has no concept of input or output. It is designed to run as a scripting language in a host environment, and it is up to the host environment to provide mechanisms for communicating with the outside world. The most common host environment is the browser.

JavaScript is an interpreted programming language that conforms to the ECMAScript specification. JavaScript is high-level, often just-in-time compiled, and multi-paradigm. It has curly-bracket syntax, dynamic typing, prototype-based object-orientation, and first-class functions. In this path you will learn the basics of JavaScript as well as more advanced topics such as promises, asynchronous programming, proxies and reflection.

Java Script Overview & Basics

JavaScript is the dominant client-side scripting language of the Web, with 98% of all websites (mid–2022) using it for this purpose. Scripts are embedded in or included from HTML documents and interact with the DOM. All major web browsers have a built-in JavaScript engine that executes the code on the user's device.

- **Client-side**: It supplies objects to control a browser and its Document Object Model (DOM). Like if client-side extensions allow an application to place elements on an HTML form and respond to user events such as mouse clicks, form input, and page navigation. Useful libraries for the client-side are AngularJS, ReactJS, VueJS and so many others.
- **Server-side**: It supplies objects relevant to running JavaScript on a server. Like if the server-side extensions allow an application to communicate with a database, and provide continuity of information from one invocation to another of the application, or perform file manipulations on a server. The useful framework which is the most famous these days is node.js.
- **Imperative language** – In this type of language we are mostly concern about how it is to be done. It simply controls the flow of computation. The procedural programming approach, object, oriented approach comes under this like async await we are thinking what it is to be done further after async call.
- **Declarative programming** – In this type of language we are concern about how it is to be done, basically here logical computation requires. Here main goal is to describe the desired result without direct dictation on how to get it like arrow function do.



JavaScript can be added to your HTML file in two ways:
Internal JS: We can add JavaScript directly to our HTML file by writing the code inside the <script> tag. The <script> tag can either be placed inside the <head> or the <body> tag according to the requirement.
Syntax:

```
<script>
// JavaScript Code
</script>
```

External JS: We can write JavaScript code in other file having an extension.js and then link this file inside the <head> tag of the HTML file in which we want to add this code.

```
<!DOCTYPE html>
<html>
<body>
        <h2>External JavaScript</h2>
        <p id="demo">Geetanjali College</p>
        <button type="button" onclick="myFunction()">Try it</button>
<script src="myScript.js"></script>
</body>
</html>
```

Advantages of External JavaScript:

- **Cached JavaScript files can speed up page loading**
- **It makes JavaScript and HTML easier to read and maintain**
- **It separates the HTML and JavaScript code**
- **It focuses on code re usability that is one JavaScript Code can run in various HTML files.**

JavaScript in body or head: Scripts can be placed inside the body or the head section of an HTML page or inside both head and body.
JavaScript in head: A JavaScript function is placed inside the head section of an HTML page and the function is invoked when a button is clicked.
JavaScript in body: A JavaScript function is placed inside the body section of an HTML page and the function is invoked when a button is clicked.
Variable, Conditional Statements, Loops in JS

A **JavaScript variable** is simply a name of storage location. There are two types of variables in JavaScript: **local** variable and **global** variable.
There are some rules while declaring a JavaScript variable (also known as identifiers).

1. Name must start with a letter (a to z or A to Z), underscore (_), or dollar($) sign.
2. After first letter we can use digits (0 to 9), for example value1.
3. JavaScript variables are case sensitive, for example x and X are different variables.

Example of JavaScript variable

```
<script>
var x = 10;
var y = 20;
var z=x+y;
document.write(z);
</script>
```

Output

```
30
```

JavaScript local variable

A JavaScript local variable is declared inside block or function. It is accessible within the function or block only.
For example:

```
<script>
function abc(){
        var x=10;//local variable
}
</script>
```

JavaScript global variable

A JavaScript global variable is accessible from any function. A variable i.e. declared outside the function or declared with window object is known as global variable. For example:

```
<script>
var data=200;//gloabal variable
function a(){
        document.writeln(data);
}
function b(){
        document.writeln(data);
}
a();//calling JavaScript function
b();
</script>
```

Output

```
200
```

JavaScript If-else

The JavaScript if-else statement is used to execute the code whether condition is true or false. There are three forms of if statement in JavaScript.

- If Statement
- If else statement
- if else if statement

JavaScript If statement

It evaluates the content only if expression is true. The signature of JavaScript if statement is given below.

```
if(expression){
//content to be evaluated
}
<script>
var a=20;
if(a>10){
        document.write("value of a is greater than 10");
}
</script>
```

Output

```
value of a is greater than 10
```

JavaScript If...else Statement

It evaluates the content whether condition is true of false. The syntax of JavaScript if-else statement is given below.

```
if(expression){
        //content to be evaluated if condition is true
}
else{
        //content to be evaluated if condition is false
}
<script>
var a=20;
if(a%2==0){
        document.write("a is even number");
}
else{
        document.write("a is odd number");
}
</script>
```

Output

```
a is even number
```

JavaScript If...else if statement

It evaluates the content only if expression is true from several expressions. The signature of JavaScript if else if statement is given below.

```
if(expression1){
//content to be evaluated if expression1 is true
}
```

```
else if(expression2){
//content to be evaluated if expression2 is true
}
else if(expression3){
//content to be evaluated if expression3 is true
}
else{
//content to be evaluated if no expression is true
}
```

```
<script>
var a=20;
if(a==10){
        document.write("a is equal to 10");
}
else if(a==15){
        document.write("a is equal to 15");
}
else if(a==20){
        document.write("a is equal to 20");
}
else{
        document.write("a is not equal to 10, 15 or 20");
}
</script>
```

Output

```
a is equal to 20
```

JavaScript Switch

The JavaScript switch statement is used to execute one code from multiple expressions. It is just like else if statement that we have learned in previous page. But it is convenient than if..else..if because it can be used with numbers, characters etc.

The signature of JavaScript switch statement is given below.

```
switch(expression){
        case value1:
                code to be executed;
        break;
        case value2:
                code to be executed;
        break;
        ......

        default:
                code to be executed if above values are not matched;
}
```

```
<script>
var grade='B';
var result;
switch(grade){
        case 'A':
                result="A Grade";
        break;
        case 'B':
                result="B Grade";
        break;
        case 'C':
                result="C Grade";
        break;
        default:
                result="No Grade";
}
document.write(result);
</script>
```

JavaScript Loops
The JavaScript loops are used to iterate the piece of code using for, while, do while or for-in loops. It makes the code compact. It is mostly used in array.

There are four types of loops in JavaScript.

- **for loop**
- **while loop**
- **do-while loop**
- **for-in loop**

JavaScript For loop
The JavaScript for loop iterates the elements for the fixed number of times. It should be used if number of iteration is known. The syntax of for loop is given below.

```
for (initialization; condition; increment){
        code to be executed
}
```

Let's see the simple example of for loop in javascript.

```
<script>
for (i=1; i<=5; i++)
{
        document.write(i + "<br/>")
}
</script>
```

JavaScript while loop

The JavaScript while loop iterates the elements for the infinite number of times. It should be used if number of iteration is not known. The syntax of while loop is given below.

```
while (condition){
        code to be executed
}
```

Let's see the simple example of while loop in javascript.

```
<script>
var i=11;
while (i<=15){
        document.write(i + "<br/>");
        i++;
}
</script>
```

JavaScript do while loop

The JavaScript do while loop iterates the elements for the infinite number of times like while loop. But, code is executed at least once whether condition is true or false. The syntax of do while loop is given below.

```
do{
        code to be executed
}while (condition);
```

Let's see the simple example of do while loop in javascript.

```
<script>
var i=21;
do{
        document.write(i + "<br/>");
        i++;
}while (i<=25);
</script>
```

Functions, Arrays & Events in JS

JavaScript Functions

JavaScript functions are used to perform operations. We can call JavaScript function many times to reuse the code.

Advantage of JavaScript function

There are mainly two advantages of JavaScript functions.

Code reusability: We can call a function several times so it save coding.

Less coding: It makes our program compact. We don't need to write many lines of code each time to perform a common task.

JavaScript Function Syntax

The syntax of declaring function is given below.

```
function functionName([arg1, arg2, ...argN]){
//code to be executed
}
```

JavaScript Functions can have 0 or more arguments.

JavaScript Function Example

Let's see the simple example of function in JavaScript that does not has arguments.

```
<script>
function msg(){
        alert("hello! this is message");
}
</script>
<input type="button" onclick="msg()" value="call function"/>
```

JavaScript Function Arguments

We can call function by passing arguments. Let's see the example of function that has one argument.

```
<script>
function getcube(number){
        alert(number*number*number);
}
</script>
<form>
        <input type="button" value="click" onclick="getcube(4)"/>
</form>
```

Function with Return Value

We can call function that returns a value and use it in our program. Let's see the example of function that returns value.

```
<script>
function getInfo(){
        return "hello hardikchavda! How r u?";
}
</script>
<script>
        document.write(getInfo());
</script>
```

JavaScript Function Object
In JavaScript, the purpose of Function constructor is to create a new Function object. It executes the code globally. However, if we call the constructor directly, a function is created dynamically but in an unsecured way.

Syntax
```
new Function ([arg1[, arg2[, ....argn]],] functionBody)
```

Parameter
- arg1, arg2, ... , argn - It represents the argument used by function.
- functionBody - It represents the function definition.
-

Let's see function methods with description.

Method	**Description**
apply()	It is used to call a function contains this value and a single array of arguments.
bind()	It is used to create a new function.
call()	It is used to call a function contains this value and an argument list.
toString()	It returns the result in a form of a string.

JavaScript Function Object Examples

Example 1
Let's see an example to display the sum of given numbers.
```
<script>
var add=new Function("num1","num2","return num1+num2");
document.writeln(add(2,5));
</script>
```

Example 2
Let's see an example to display the power of provided value.
```
<script>
var pow=new Function("num1","num2","return Math.pow(num1,num2)");
document.writeln(pow(2,3));
</script>
```

JavaScript Array
JavaScript array is an object that represents a collection of similar type of elements.

There are 3 ways to construct array in JavaScript

By array literal
By creating instance of Array directly (using new keyword)
By using an Array constructor (using new keyword)

JavaScript array literal

The syntax of creating array using array literal is given below:

```
var arrayname=[value1,value2.....valueN];
```

As you can see, values are contained inside [ ] and separated by , (comma).

Let's see the simple example of creating and using array in JavaScript.

```
<script>
var emp=["Sonoo","Vimal","Ratan"];
for (i=0;i<emp.length;i++){
        document.write(emp[i] + "<br/>");
}
</script>
//The .length property returns the length of an array.
```

Output

```
Sonoo
Vimal
Ratan
```

JavaScript Array directly (new keyword)

The syntax of creating array directly is given below:

```
var arrayname=new Array();
```

Here, new keyword is used to create instance of array.

Let's see the example of creating array directly.

```
<script>
var i;
var emp = new Array();
emp[0] = "Arun";
emp[1] = "Varun";
emp[2] = "John";

for (i=0;i<emp.length;i++){
        document.write(emp[i] + "<br>");
}
</script>
```

Output

```
Arun
Varun
John
```

JavaScript array constructor (new keyword)

Here, you need to create instance of array by passing arguments in constructor so that we don't have to provide value explicitly.

The example of creating object by array constructor is given below.

```
<script>
var emp=new Array("Jai","Vijay","Smith");
for (i=0;i<emp.length;i++){
        document.write(emp[i] + "<br>");
}
</script>
```

Output

```
Jai
Vijay
Smith
```

JavaScript Array Methods

Let's see the list of JavaScript array methods with their description.

Methods	**Description**
concat()	It returns a new array object that contains two or more merged arrays.
copywithin()	It copies the part of the given array with its own elements and returns the modified array.
entries()	It creates an iterator object and a loop that iterates over each key/value pair.
every()	It determines whether all the elements of an array are satisfying the provided function conditions.
flat()	It creates a new array carrying sub-array elements concatenated recursively till the specified depth.
flatMap()	It maps all array elements via mapping function, then flattens the result into a new array.
fill()	It fills elements into an array with static values.
from()	It creates a new array carrying the exact copy of another array element.
filter()	It returns the new array containing the elements that pass the provided function conditions.
find()	It returns the value of the first element in the given array that satisfies the specified condition.
findIndex()	It returns the index value of the first element in the given array that satisfies the specified condition.
forEach()	It invokes the provided function once for each element of an array.
includes()	It checks whether the given array contains the specified element.
indexOf()	It searches the specified element in the given array and returns the index

	of the first match.
isArray()	It tests if the passed value ia an array.
join()	It joins the elements of an array as a string.
keys()	It creates an iterator object that contains only the keys of the array, then loops through these keys.
lastIndexOf()	It searches the specified element in the given array and returns the index of the last match.
map()	It calls the specified function for every array element and returns the new array
of()	It creates a new array from a variable number of arguments, holding any type of argument.
pop()	It removes and returns the last element of an array.
push()	It adds one or more elements to the end of an array.
reverse()	It reverses the elements of given array.
reduce(function, initial)	It executes a provided function for each value from left to right and reduces the array to a single value.
reduceRight()	It executes a provided function for each value from right to left and reduces the array to a single value.
some()	It determines if any element of the array passes the test of the implemented function.
shift()	It removes and returns the first element of an array.
slice()	It returns a new array containing the copy of the part of the given array.
sort()	It returns the element of the given array in a sorted order.
splice()	It add/remove elements to/from the given array.
toLocaleString()	It returns a string containing all the elements of a specified array.
toString()	It converts the elements of a specified array into string form, without affecting the original array.
unshift()	It adds one or more elements in the beginning of the given array.
values()	It creates a new iterator object carrying values for each index in the array.

JavaScript Events

The change in the state of an object is known as an Event. In html, there are various events which represents that some activity is performed by the user or by the browser. When javascript code is included in HTML, js react over these events and allow the execution. This process of reacting over the events is called Event Handling. Thus, js handles the HTML events via Event Handlers.

For example, when a user clicks over the browser, add js code, which will execute the task to be performed on the event.

Some of the HTML events and their event handlers are:

Mouse events:

Event Performed	Event Handler	Description
click	onclick	When mouse click on an element
mouseover	onmouseover	When the cursor of the mouse comes over the element
mouseout	onmouseout	When the cursor of the mouse leaves an element
mousedown	onmousedown	When the mouse button is pressed over the element
mouseup	onmouseup	When the mouse button is released over the element
mousemove	onmousemove	When the mouse movement takes place.

Keyboard events:

Event Performed	Event Handler	Description
Keydown & Keyup	onkeydown & onkeyup	When the user press and then release the key

Form events:

Event Performed	Event Handler	Description
focus	onfocus	When the user focuses on an element
submit	onsubmit	When the user submits the form
blur	onblur	When the focus is away from a form element
change	onchange	When the user modifies or changes the value of a form element

Window/Document events

Event Performed	Event Handler	Description
load	onload	When the browser finishes the loading of the page
unload	onunload	When the visitor leaves the current webpage, the browser unloads it
resize	onresize	When the visitor resizes the window of the browser

Click Event

```
<html>
<head> Javascript Events </head>
<body>
<script language="Javascript" type="text/Javascript">
function clickevent()
{
        document.write("This is HardikChavda");
}
</script>
        <form>
                <input type="button" onclick="clickevent()" value="Who's this?"/>
        </form>
</body>
</html>
```

MouseOver Event

```
<html>
<head>
<h1> Javascript Events </h1>
</head>
<body>
<script language="Javascript" type="text/Javascript">
function mouseoverevent()
{
        alert("This is HardikChavda");
}
</script>
        <p onmouseover="mouseoverevent()"> Keep cursor over me</p>
</body>
</html>
```

Focus Event

```
<html>
<head> Javascript Events</head>
<body>
        <h2> Enter something here</h2>
        <input type="text" id="input1" onfocus="focusevent()"/>
<script>
function focusevent()
{
        document.getElementById("input1").style.background=" aqua";
}
</script>
</body>
</html>
```

Keydown Event

```
<html>
<head> Javascript Events</head>
<body>
<h2> Enter something here</h2>
<input type="text" id="input1" onkeydown="keydownevent()"/>
<script>
function keydownevent()
{
document.getElementById("input1");
alert("Pressed a key");
}
</script>
</body>
</html>
Load event
<html>
<head>Javascript Events</head>
</br>
<body onload="window.alert('Page successfully loaded');">
<script>
document.write("The page is loaded successfully");
</script>
</body>
</html>
```

ES6 Overview & Basics

ES6 or ECMAScript 6 is a scripting language specification which is standardized by ECMAScript International. This specification governs some languages such as JavaScript, ActionScript, and Jscript. ECMAScript is generally used for client-side scripting, and it is also used for writing server applications and services by using Node.js.

ES6 allows you to write the code in such a way that makes your code more modern and readable. By using ES6 features, we write less and do more, so the term 'Write less, do more' suits ES6.

What is ES6?
ES6 is an acronym of ECMAScript 6 and also known as ECMAScript 2015.

ES6 or ECMAScript6 is a scripting language specification which is standardized by ECMAScript International. It is used by the applications to enable client-side scripting. This specification is affected by programming languages like Self, Perl, Python, Java, etc. This specification governs some languages such as JavaScript, ActionScript, and Jscript. ECMAScript is generally used for client-side scripting, and it is also used for writing server applications and services by using Node.js.

ES6 allows you to make the code more modern and readable. By using ES6 features, we write less and do more, so the term 'Write less, do more' suits ES6. ES6 introduces you many great features such as scope variable, arrow functions, template strings, class destructions, modules, etc.

ES6 was created to standardize JavaScript to help several independent implementations. Since the standard was first published, JavaScript has remained the well-known implementation of ECMAScript, comparison to other most famous implementations such as Jscript and ActionScript.

History
The ECMAScript specification is the standardized specification of scripting language, which is developed by Brendan Eich (He is an American technologist and the creator of JavaScript programming language) of Netscape (It is a name of brand which is associated with Netscape web browser's development).

Initially, the ECMAScript was named Mocha, later LiveScript, and finally, JavaScript. In December 1995, Sun Microsystems (an American company that sold the computers and its components, software, and IT services. It created Java, NFS, ZFS, SPARC, etc.) and Netscape announced the JavaScript during a press release.

During November 1996, Netscape announced a meeting of the ECMA International standard organization to enhance the standardization of JavaScript.

ECMA General Assembly adopted the first edition of ECMA-262 in June 1997. Since then, there are several editions of the language standard have published. The Name 'ECMAScript' was a settlement between the organizations which included the standardizing of the language, especially Netscape and Microsoft, whose disputes dominated the primary standard sessions. Brendan Eich commented that 'ECMAScript was always an unwanted trade name which sounds like a skin disease (eczema).'

Both JavaScript and Jscript aim to be compatible with the ECMAScript, and they also provide some of the additional features that are not described in ECMA specification.

Prerequisite
Before learning ES6, you should have a basic understanding of JavaScript.

ES6 Classes, functions & Promises

ES6 Classes

Classes are an essential part of object-oriented programming (OOP). Classes are used to define the blueprint for real-world object modeling and organize the code into reusable and logical parts.

Before ES6, it was hard to create a class in JavaScript. But in ES6, we can create the class by using the class keyword. We can include classes in our code either by class expression or by using a class declaration.

A class definition can only include constructors and functions. These components are together called as the data members of a class. The classes contain constructors that allocates the memory to the objects of a class. Classes contain functions that are responsible for performing the actions to the objects.

Syntax: Class Expression

```
var var_name = new class_name {}
```

Syntax: Class Declaration

```
class Class_name{}
```

Let us see the illustration for the class expression and class declaration.

Example - Class Declaration

```
class Student{
        constructor(name, age){
        this.name = name;
        this.age = age;
        }
}
```

Example - Class Expression

```
var Student = class{
          constructor(name, age){
          this.name = name;
          this.age = age;
          }
}
```

Instantiating an Object from class

Like other object-oriented programming languages, we can instantiate an object from class by using the new keyword.

Syntax

```
var obj_name = new class_name([arguements])
```

Accessing functions

The object can access the attributes and functions of a class. We use the '.' dot notation (or period) for accessing the data members of the class.

Syntax

```
obj.function_name();
```

Example

```
'use strict'
class Student {
          constructor(name, age) {
          this.n = name;
          this.a = age;
}
stu() {
          console.log("The Name of the student is: ", this.n)
          console.log("The Age of the student is: ",this. a)
       }
}
var stuObj = new Student('Peter',20);
stuObj.stu();
//The function stu() in the class will print the values of name and age.
```

Output

```
The Name of the student is:  Peter
The Age of the student is:  20
```

In the above example, we have declared a class Student. The constructor of the class contains two arguments name and age, respectively. The keyword 'this' refers to the current instance of the class. We can also say that the above constructor initializes two variables 'n' and 'a' along with the parameter values passed to the constructor.

Note: Including a constructor definition is mandatory in class because, by default, every class has a constructor.

The Static keyword

The static keyword is used for making the static functions in the class. Static functions are referenced only by using the class name.

Example

```
'use strict'
class Example {
        static show() {
        console.log("Static Function")
        }
}
Example.show() //invoke the static method
```

Class inheritance

Before the ES6, the implementation of inheritance required several steps. But ES6 simplified the implementation of inheritance by using the extends and super keyword.

Inheritance is the ability to create new entities from an existing one. The class that is extended for creating newer classes is referred to as superclass/parent class, while the newly created classes are called subclass/child class.

A class can be inherited from another class by using the 'extends' keyword. Except for the constructors from the parent class, child class inherits all properties and methods.

Syntax

```
class child_class_name extends parent_class_name{}
```

A class inherits from the other class by using the extends keyword.

Example

```
'use strict'
class Student {
        constructor(a) {
        this.name = a;
        }
}
class User extends Student {
        show() {
        console.log("The name of the student is:  "+this.name)
        }
}
var obj = new User('Sahil');
obj.show()
```

Output

```
The name of the student is:  Sahil
```

In the above example, we have declared a class student. By using the extends keyword, we can create a new class User that shares the same characteristics as its parent class Student. So, we can see that there is an inheritance relationship between these classes.

Types of inheritance

Inheritance can be categorized as Single-level inheritance, Multiple inheritance, and Multi-level inheritance. Multiple inheritance is not supported in ES6.

Single-level Inheritance

It is defined as the inheritance in which a derived class can only be inherited from only one base class. It allows a derived class to inherit the behavior and properties of a base class, which enables the reusability of code as well as adding the new features to the existing code. It makes the code less repetitive.

Multiple Inheritance

In multiple inheritance, a class can be inherited from several classes. It is not supported in ES6.

Multi-level Inheritance

In Multi-level inheritance, a derived class is created from another derived class. Thus, a multi-level inheritance has more than one parent class.

Let us understand it with the following example.

```
class Animal{
        eat(){
                console.log("eating...");
        }
}
class Dog extends Animal{
        bark(){
                console.log("barking...");
        }
}
class BabyDog extends Dog{
        weep(){
                console.log("weeping...");
        }
}
var d=new BabyDog();
d.eat(); d.bark(); d.weep();
```

Output

```
eating...
barking...
weeping...
```

Method Overriding and Inheritance
It is a feature that allows a child class to provide a specific implementation of a method which has been already provided by one of its parent class.

There are some rules defined for method overriding -

- **The method name must be the same as in the parent class.**
- **Method signatures must be the same as in the parent class.**

Let us try to understand the illustration for the same:

Example

```
'use strict' ;
class Parent {
        show() {
                console.log("It is the show() method from the parent class");
        }
}
class Child extends Parent {
        show() {
                console.log("It is the show() method from the child class");
        }
}
var obj = new Child();
obj.show();
```

Output

```
It is the show() method from the child class
```

In the above example, the implementation of the superclass function has changed in the child class.

The super keyword
It allows the child class to invoke the properties, methods, and constructors of the immediate parent class. It is introduced in ECMAScript 2015 or ES6. The super.prop and super[expr] expressions are readable in the definition of any method in both object literals and classes.

Syntax

```
super(arguments);
```

In this example, the characteristics of the parent class have been extended to its child class. Both classes have their unique properties. Here, we are using the super keyword to access the property from parent class to the child class.

```
'use strict' ;
class Parent {
        show() {
                console.log("It is the show() method from the parent class");
        }
}
class Child extends Parent {
        show() {
                super.show();
        console.log("It is the show() method from the child class");
        }
}
var obj = new Child();
obj.show();
```
Output
```
It is the show() method from the parent class
It is the show() method from the child class
```

ES6 Functions

A function is the set of input statements, which performs specific computations and produces output. It is a block of code that is designed for performing a particular task. It is executed when it gets invoked (or called). Functions allow us to reuse and write the code in an organized way. Functions in JavaScript are used to perform operations.

In JavaScript, functions are defined by using function keyword followed by a name and parentheses (). The function name may include digits, letters, dollar sign, and underscore. The brackets in the function name may consist of the name of parameters separated by commas. The body of the function should be placed within curly braces {}.

The syntax for defining a standard function is as follows:

```
function function_name() {
//body of the function
}
```

To force the execution of the function, we must have to invoke (or call) the function. It is known as function invocation. The syntax for invoking a function is as follows:

```
function_name()
```
Example
```
function hello() //defining a function
{
console.log ("hello function called");
}
hello(); //calling of function
```
Output
```
hello function called
```

In the above code, we have defined a function hello(). The pair of parentheses {} define the body of the function, which is called a scope of function.

Let us try to understand different functions.

Parameterized functions

Parameters are the names that are listed in the definition of the function. They are the mechanism of passing the values to functions.

The values of the parameters are passed to the function during invocation. Unless it is specified explicitly, the number of values passed to a function must match with the defined number of parameters.

Syntax

```
function function_name( parameter1,parameter2 ,.....parameterN) {
//body of the function
}
```

Example

In this example of parameterized function, we are defining a function mul(), which accepts two parameters x and y, and it returns their multiplication in the result. The parameter values are passed to the function during invocation.

```
function mul( x , y) {
        var c = x * y;
        console.log("x = "+x);
        console.log("y = "+y);
        console.log("x * y = "+c);
}
mul(20,30);
```

Output

```
x = 20
y = 30
x * y = 600
```

Default Function Parameters

In ES6, the function allows the initialization of parameters with default values if the parameter is undefined or no value is passed to it.

You can see the illustration for the same in the following code:

For example

```
function show (num1, num2=200)
{
        console.log("num1 = " +num1);
        console.log("num2 = " +num2);
}
show(100);
```

Output

```
100 200
```

In the above function, the value of num2 is set to 200 by default. The function will always consider 200 as the value of num2 if no value of num2 is explicitly passed.

The default value of the parameter 'num2' will get overwritten if the function passes its value explicitly. You can understand it by using the following example:

For example

```
function show(num1, num2=200)
{
        console.log("num1 = " +num1);
        console.log("num2 = " +num2);
}
show(100,500);
```

Output

```
100 500
```

Rest Parameters

Rest parameters do not restrict you to pass the number of values in a function, but all the passed values must be of the same type. We can also say that rest parameters act as the placeholders for multiple arguments of the same type.

For declaring the rest parameter, the name of the parameter should be prefixed with the spread operator that has three periods (not more than three or not less than three).

You can see the illustration for the same in the following example:

```
function show(a, ...args)
{
        console.log(a + " " + args);
}
show(50,60,70,80,90,100);
```

Output

```
50,60,70,80,90,100
```

Note: The rest parameters should be at last in the list of function parameters.

Returning functions

The function also returns the value to the caller by using the return statement. These functions are known as Returning functions. A returning function should always end with a return statement. There can be only one return statement in a function, and the return statement should be the last statement in the function.

When JavaScript reaches the return statement, the function stops the execution and exits immediately. That's why you can use the return statement to stop the execution of the function immediately.

Syntax

A function can return the value by using the return statement, followed by a value or an expression. The syntax for the returning function is as follows:

```
function function_name() {
//code to be executed
return value;
}
```

Example

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

Output

```
30
```

In the above example, we are defining a function add() that has two parameters a and b. This function returns the addition of the arguments to the caller.

Generator functions

Generator (or Generator function) is the new concept introduced in ES6. It provides you a new way of working with iterators and functions.

ES6 generator is a different kind of function that may be paused in the middle either one or many times and can be resumed later.

Anonymous function

An anonymous function can be defined as a function without a name. The anonymous function does not bind with an identifier. It can be declared dynamically at runtime. The anonymous function is not accessible after its initial creation.

An anonymous function can be assigned within a variable. Such expression is referred to as function expression. The syntax for the anonymous function is as follows.

Syntax

```
var y = function( [arguments] )
{
//code to be executed
}
```

Example

```
var hello = function() {
        console.log('Hello World');
        console.log('I am an anonymous function');
}
hello();
```

Output

```
Hello World
I am an anonymous function
```

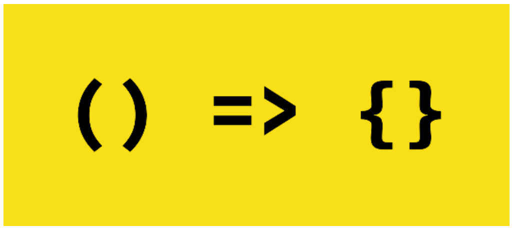
Anonymous function as an argument
One common use of the anonymous function is that it can be used as an argument to other functions.

Example
Use as an argument to other function.

```
setTimeout(function()
{
        console.log('Hello World');
}, 2000);
//When you execute the above code, it will show you the output after two seconds.
```
Output
```
Hello World
```

Parameterized Anonymous function
Example
```
var anon = function(a,b)
{
        return a+b
}
function sum() {
        var res;
        res = anon(100,200);
        console.log("Sum: "+res)
}
sum()
```
Output
```
Sum: 300
```

Arrow functions
Arrow functions are introduced in ES6, which provides you a more accurate way to write the functions in JavaScript. They allow us to write smaller function syntax. Arrow functions make your code more readable and structured.



Before Arrow Function
```
Hello = function() {
  return "Hello World!";
}
```

With Arrow Functions

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

Arrow Functions Return By Default

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

Arrow Function With Parameters:

```
hello = (val) => "Hello " + val;
```

Arrow Function Without Parentheses:

```
hello = val => "Hello " + val;
```

What About this in Arrow Functions?
The handling of “**this”** is also different in arrow functions compared to regular functions.
In short, with arrow functions there are no binding of “**this”**.

In regular functions the **“this”** keyword represented the object that called the function, which could be the window, the document, a button or whatever.

With arrow functions the “**this”** keyword always represents the object that defined the arrow function.

Function Hoisting
As variable hoisting, we can perform the hoisting in functions. Unlike the variable hoisting, when function declarations get hoisted, it only hoists the function definition instead of just hoisting the name of the function.

Let us try to illustrate the function hoisting in JavaScript by using the following example:

Example
In this example, we call the function before writing it. Let's see what happens when we call the function before writing it.

```
hello();
function hello() {
        console.log("Hello world ");
}
```
Output
```
Hello world
```

In the above code, we have called the function first before writing it, but the code still works.

However, function expressions cannot be hoisted. Let us try to see the illustration of hoisting the function expressions in the following example.

```
hello();
var hello = function() {
        console.log("Hello world ");
}
```

When you execute the above code, you will get a "TypeError: hello is not a function." It happens because function expressions cannot be hoisted.

Output

```
TypeError: hello is not a function
```

JavaScript Functions and Recursion

When a function calls itself, then it is known as a recursive function. Recursion is a technique to iterate over an operation by having a function call itself repeatedly until it reaches a result.

It is the best way when we require to call the same function regularly with different parameters in a loop.

Example

```
function fact(num) {
        if (num <= 1) {
                return 1;
        } else {
                return num * fact(num - 1);
        }
}
console.log(fact(6));
console.log(fact(5));
console.log(fact(4));
```

Output

```
720
120
24
```

Function Expression v/s Function Declaration
The fundamental difference between both of them is that the function declarations are parsed before the execution, but the function expressions are parsed only when the script engine encounters it during the execution.

Unlike the function declarations, function expressions in JavaScript do not hoists. You cannot use the function expressions before defining them.

The main difference between a function declaration and function expression is the name of the function, which can be omitted in the function expressions for creating the anonymous functions.

ES6 Promises
A Promise represents something that is eventually fulfilled. A Promise can either be rejected or resolved based on the operation outcome.

ES6 Promise is the easiest way to work with asynchronous programming in JavaScript. Asynchronous programming includes the running of processes individually from the main thread and notifies the main thread when it gets complete. Prior to the Promises, Callbacks were used to perform asynchronous programming.

Callback
A Callback is a way to handle the function execution after the completion of the execution of another function.

A Callback would be helpful in working with events. In Callback, a function can be passed as a parameter to another function.

Why Promise required?
A Callback is a great way when dealing with basic cases like minimal asynchronous operations. But when you are developing a web application that has a lot of code, then working with Callback will be messy. This excessive Callback nesting is often referred to as Callback hell.

To deal with such cases, we have to use Promises instead of Callbacks.

How Does Promise work?
The Promise represents the completion of an asynchronous operation. It returns a single value based on the operation being rejected or resolved. There are mainly three stages of the Promise, which are shown below:

ES6 Promises
Pending - It is the initial state of each Promise. It represents that the result has not been computed yet.

Fulfilled - It means that the operation has completed.

Rejected - It represents a failure that occurs during computation.

Once a Promise is fulfilled or rejected, it will be immutable. The Promise() constructor takes two arguments that are rejected function and a resolve function. Based on the asynchronous operation, it returns either the first argument or second argument.

Creating a Promise
In JavaScript, we can create a Promise by using the Promise() constructor.

Syntax

```
const Promise = new Promise((resolve,reject) => {....});
```

Example

```
let Promise = new Promise((resolve, reject)=>{
let a = 3;
if(a==3){
        resolve('Success');
}
else{
        reject('Failed');
}
})
Promise.then((message)=>{
        console.log("It is then block. The message is: ?+ message)
}).catch((message)=>{
        console.log("It is Catch block. The message is: ?+ message)
})
```

Output

```
It is then block. The message is: Success
```

Promise Methods
The Promise methods are used to handle the rejection or resolution of the Promise object. Let's understand the brief 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.

Let's understand with the following example how to handle the Promise rejection and resolution by using .then() method.

Example

```
let success = (a) => {
	console.log(a + " it worked!")
}
let error = (a) => {
	console.log(a + " it failed!")
}
const Promise = num => {
	return new Promise((resolve,reject) => {
		if((num%2)==0){
			resolve("Success!")
		}
	reject("Failure!")
	})
}

Promise(100).then(success, error)
Promise(21).then(success,error)
```

Output

```
Success! it worked!
Failure! it failed!
```

.catch()

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

Let's understand with the following example how to handle the Promise rejection and failure by using .catch() method.

Example

```
const Promise = num => {
	return new Promise((resolve,reject) => {
	if(num > 0){
		resolve("Success!")
	}
	reject("Failure!")
	})
}
Promise(20).then(res => {
throw new Error();
console.log(res + " success!")
}).catch(error => {
console.log(error + " oh no, it failed!")
})
```

Output

```
Error oh no, it failed!
```

.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.

```
Promise.resolve('Success').then(function(val) {
          console.log(val);
          }, function(val) {
});
```

.reject()

It returns a rejected Promise object with the given value.

Example

```
function resolved(result) {
          console.log('Resolved');
}

function rejected(result) {
          console.error(result);
}
Promise.reject(new Error('fail')).then(resolved, rejected);
```

Output

```
Error: fail
```

.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.

Example

```
const PromiseA = Promise.resolve('Hello');
const PromiseB = 'World';
const PromiseC = new Promise(function(resolve, reject) {
          setTimeout(resolve, 100, 1000);
});

Promise.all([PromiseA, PromiseB, PromiseC]).then(function(values) {
          console.log(values);
});
```

Output

```
[ 'Hello', 'World', 1000 ]
```

.race()
This method is used to return a resolved Promise based on the first referenced Promise that resolves.

Example

```
const Promise1 = new Promise((resolve,reject) => {
        setTimeout(resolve("Promise 1 is first"),1000)
})

const Promise2= new Promise((resolve,reject) =>{
        setTimeout(resolve("Promise 2 is first"),2000)
})

Promise.race([Promise1,Promise2]).then(result => {
        console.log(result);
})
```

Output

```
Promise 1 is first
```

ES6 Promise Chaining
Promise chaining allows us to control the flow of JavaScript asynchronous operations. By using Promise chaining, we can use the returned value of a Promise as the input to another asynchronous operation.

Promise chaining is helpful when we have multiple interdependent asynchronous functions, and each of these functions should run one after another.

Let us try to understand the concept of Promise chaining by using the following example:

Example

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Document</title>
</head>
<body>
<script>
const PromiseA = () =>{
        return new Promise((resolve,reject)=>{
        resolve("Hello Promoise A");
        });
}
```

```
const PromiseB = () =>{
        return new Promise((resolve,reject)=>{
        resolve("Hello Promise B");
        });
}

const PromiseC = () =>{
        return new Promise((resolve,reject)=>{
        resolve("Hello Promise C");
        });
}
PromiseA().then((A)=>{
        console.log(A);
        return PromiseB();
}).then((B)=>{
        console.log(B);
        return PromiseC();
}).then((C)=>{
        console.log(C);
});
</script>
</body>
</html>
```

Output:

```
Hello Promoise A
Hello Promoise B
Hello Promoise C
```

Express JS
Express.js is a web framework for Node.js. It is a fast, robust and asynchronous in nature.

Our Express.js tutorial includes all topics of Express.js such as Express.js installation on windows and linux, request object, response object, get method, post method, cookie management, scaffolding, file upload, template etc.

What is Express.js
Express is a fast, assertive, essential and moderate web framework of Node.js. You can assume express as a layer built on the top of the Node.js that helps manage a server and routes. It provides a robust set of features to develop web and mobile applications.

Let's see some of the core features of Express framework:

- It can be used to design single-page, multi-page and hybrid web applications.
- It allows to setup middlewares to respond to HTTP Requests.
- It defines a routing table which is used to perform different actions based on HTTP method and URL.
- It allows to dynamically render HTML Pages based on passing arguments to templates.

Why use Express
- Ultra fast I/O
- Asynchronous and single threaded
- MVC like structure
- Robust API makes routing easy

How does Express look like

Let's see a basic Express.js app.

File: basic_express.js

```
var express = require('express');
var app = express();
app.get('/', (req, res)=>{
        res.send('Welcome to Geetanjali!');
});
var server = app.listen(8000, function () {
var host = server.address().address;
var port = server.address().port;
console.log('Example app listening at http://%s:%s', host, port);
});
```

Setting up an app with ExpressJS
Install Express.js
Firstly, you have to install the express framework globally to create web application using Node terminal. Use the following command to install express framework globally.

```
npm i express
```

Installing Express
Use the following command to install express:

```
npm install express –save
```

The above command install express in node_module directory and create a directory named express inside the node_module. You should install some other important modules along with express. Following is the list:

body-parser: This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.

cookie-parser: It is used to parse Cookie header and populate req.cookies with an object keyed by the cookie names.

multer: This is a node.js middleware for handling multipart/form-data.
npm install body-parser –save

```
npm install cookie-parser --save
npm install multer –save
```

Express.js App Example

Let's take a simple Express app example which starts a server and listen on a local port. It only responds to homepage. For every other path, it will respond with a 404 Not Found error.

File: express_example.js
```
var express = require('express');
var app = express();
        app.get('/', (req, res)=> {
        res.send('Welcome to Geetanjali);
})
var server = app.listen(8000, function () {
var host = server.address().address
var port = server.address().port
console.log("Example app listening at http://%s:%s", host, port)
})
```

Routing in ExpressJS,

Express.js Routing

Routing is made from the word route. It is used to determine the specific behavior of an application. It specifies how an application responds to a client request to a particular route, URI or path and a specific HTTP request method (GET, POST, etc.). It can handle different types of HTTP requests.

Let's take an example to see basic routing.

File: routing_example.js

```
var express = require('express');
var app = express();

app.get('/', (req, res)=> {
console.log("Got a GET request for the homepage");
res.send('Welcome to Geetanjali!');
})

app.post('/', (req, res)=>  {
console.log("Got a POST request for the homepage");
res.send('I am Impossible! ');
})

app.delete('/del_student', (req, res)=> {
console.log("Got a DELETE request for /del_student");
res.send('I am Deleted!');
})

app.get('/enrolled_student', (req, res)=> {
console.log("Got a GET request for /enrolled_student");
res.send('I am an enrolled student.');
})

// This responds a GET request for abcd, abxcd, ab123cd, and so on
app.get('/ab*cd', (req, res)=> {
console.log("Got a GET request for /ab*cd");
res.send('Pattern Matched.');
})

var server = app.listen(8000, function () {
var host = server.address().address
var port = server.address().port
console.log("Example app listening at http://%s:%s", host, port)
})
```

You see that server is listening.
Now, you can see the result generated by server on the local host http://127.0.0.1:8000
Output:

Note: The Command Prompt will be updated after one successful response.

You can see the different pages by changing routes. http://127.0.0.1:8000/enrolled_student

Updated command prompt:

This can read the pattern like abcd, abxcd, ab123cd, and so on.

Next route http://127.0.0.1:8000/abcd

Next route http://127.0.0.1:8000/ab12345cd

Updated command prompt:

Connecting views with templates configurations

Express.js Template Engine
What is a template engine
A template engine facilitates you to use static template files in your applications. At runtime, it replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. So this approach is preferred to design HTML pages easily.

Following is a list of some popular **template engines** that work with Express.js:
Ejs (formerly known as jade), mustache, dust, atpl, eco, ect, **ejs**, haml, haml-coffee, handlebars, hogan, jazz, jqtpl, JUST, liquor, QEJS, swig, templayed, toffee, underscore, walrus, whiskers

In the above template engines, ejs (formerly known as jade) and mustache seems to be most popular choice. Ejs is similar to Haml which uses whitespace. According to the template-benchmark, ejs is 2x slower than Handlebars, EJS, Underscore. ECT seems to be the fastest. Many programmers like mustache template engine mostly because it is one of the simplest and versatile template engines.

Using template engines with Express
Template engine makes you able to use static template files in your application. To render template files you have to set the following application setting properties:

Views: It specifies a directory where the template files are located.
For example: app.set('views', './views').

view engine: It specifies the template engine that you use. For example, to use the Ejs template engine: app.set('view engine', ejs).

Let's take a template engine ejs

Ejs Template Engine
Let's learn how to use ejs template engine in Node.js application using Express.js. Ejs is a template engine for Node.js. Ejs uses whitespaces and indentation as the part of the syntax. Its syntax is aesy to learn.

Install ejs
Execute the following command to install ejs template engine:

```
npm i ejs
```

Express.js template engine
Ejs template must be written inside .ejs file and all .ejs files must be put inside views folder in the root folder of Node.js application.

Note: By default, Express.js searches all the views in the views folder under the root folder. you can also set to another folder using views property in express. For example:

```
app.set('views','MyViews').
```

The ejs template engine takes the input in a simple way and produces the output in HTML. See how it renders HTML:

Simple input:

```
<!DOCTYPE html>
<html>
<head>
<title>A simple ejs example</title>
</head>
<body>
<h1>This page is produced by ejs template engine</h1>
<p>some paragraph here..</p>
</body>
</html>
```

Express.js can be used with any template engine. Let's take an example to deploy how ejs template creates HTML page dynamically.

Example:
Create a file named index.ejs file inside views folder and write the following ejs template in it:

```
var express = require('express');
var app = express();
//set view engine
app.set("view engine","ejs")
app.get('/', (req, res)=> {
```

```
res.render('index');
});
var server = app.listen(5000, function () {
console.log('Node server is running..');
});
```

Error handling process in Express.js refers to the condition when errors occur in the execution of the program which may contain a continuous or non-continuous type of program code. The error handling in Express.js can easily allow the user to reduce and avoid all the delays in the program execution like AJAX requests proceeded by HTTP requests etc. These types of program handling types observe and wait for the error to arrive in the program execution process and soon as the error occurs like time limit exceeded or space limit exceeded in a computer, then it will remove all these errors by following a specific set of instructions which are named as the Error handling process in Express.js framework.

Types of Errors in Express.js:

Syntax Errors: The Syntax Errors occur in the Express.js web framework because these types of errors cannot be understood by the computer in the machine language and hence the program doesn't get executed and compiled successfully. The Syntax Errors are due to the wrong type of command or declaration of function by the user which is not accepted by the computer. For example – the wrong indentation in Python can cause a syntax error.
Runtime Errors: The Runtime Errors occur in the Express.js web framework because these types of errors can be only detected by the compiler when the program runs or the command gets executed. Before that, it is not easy to identify the run-time errors in the program.

Logical Errors: The Logical Errors in the Express.js web framework because these types of errors mainly occur when the programmer wants to implement a certain command in the program, but due to some logical error like applying some other logic gate, the program undergoes a logical error in the program which the computer is not able to understand in the machine language.
There are basically two types of program codes in Express.js due to which the error handling process in the Express.js can arrive:

Asynchronous Code: The Asynchronous Code in the Express.js program framework leads to calling the function which was declared in the previous set of instruction cycles. It means that these programs are not dependent on the clock cycle of the CPU for the execution of the program. But in the Asynchronous Code program, the processor has to indulge in multiple tasks at the same time as it has to also draw the next set of instructions in the cycle in an asynchronous code along with executing the current task. The Asynchronous Functions is the program code that collects all the data in advance because they are not clocked according to the clock cycle of the processor. So, to resolve the errors that might occur in an Asynchronous code in a program, we declare a new function which we resolve the error in the program as soon as it starts executing.

Syntax:

```
const Asyncfunction = fn => {
fn(req, res, next).catch(err => next(err));
};

export.error_catch = catchAsync(async (req, res, next) => {
const error = await middle.create(req.body);

res.status(149).json({
status: '',
data: { stop: error_catch }
});
});
```

Synchronous Code: The Synchronous Code in the Express.js program framework leads to calling the function which is declared in the current cycle of instructions execution. It means that these programs are dependent on the clock cycle of the processor for the execution of the program. During a Synchronous Code program, the processor first uses its processing power to execute the current running program and after it finishes, it goes to execute the next program.

Syntax:

```
app.begin('/', (req, res) => {
wait for new Error("Report error!")
})
```

Example: The Error Handling Procedure in the Express.js framework deals with the removal and prevention of the errors in the program code during the execution process in a synchronous and asynchronous program code. The Error Handling Procedure is mainly done by using these techniques – callbacks, middleware modules, or wait and proceed.