[**https://www.tutorialstonight.com/js/javascript-star-pattern.php**](https://www.tutorialstonight.com/js/javascript-star-pattern.php)

**https://www.youtube.com/watch?v=wZHtZ\_VJGKI&list=PL8p2I9GklV47TMMnPzqnkCtSOS3ebr4O7&index=1**

**Data types Data can be stored and manipulated within a program**

**🡪 let, const, var, tuples, enum**

**Primitive (Primary,** ప్రాథమిక **) String, Number, and Boolean**

**Composite (Reference,** మిశ్రమ**) Object, Array, and Function**

**Special data types Undefined and Null**

**Stringy – Parsing**

**JSON.stringify()**  takes a JavaScript **object** and then transforms it into a JSON **string**.

**JSON.parse()**  takes a JSON **string** and then transforms it into a JavaScript **object**. (**Remove Colons**)

Method 1 :

// Store JSON data in a JS variable

var json = '{"name": "Peter", "age": 22, "country": "United States"}';

// Converting JSON-encoded string to JS object

var obj = JSON.parse(json);

// Accessing individual value from JS object

alert(obj.name); // Outputs: Peter

alert(obj.age); // Outputs: 22

alert(obj.country); // Outputs: United States

Method 2 :

/\* Storing multi-line JSON string in a JS variable using the new ES6 template literals \*/ var json = `{ "book": { "name": "Harry Potter and the Goblet of Fire", "author": "J. K. Rowling", "year": 2000, "characters": ["Harry Potter", "Hermione Granger", "Ron Weasley"], "genre": "Fantasy Fiction", "price": { "paperback": "$10.40", "hardcover": "$20.32", "kindle": "$4.11" } } }`;

// Converting JSON object to JS object

var obj = JSON.parse(json);

// Define recursive function to print nested values

function printValues(obj) {

for(var k in obj) {

if(obj[k] instanceof Object) {

printValues(obj[k]);

} else {

document.write(obj[k] + "<br>");

};

}

};

// Printing all the values from the resulting object

printValues(obj);

document.write("<hr>");

// Printing a single value

document.write(obj["book"]["author"] + "<br>"); // Prints: J. K. Rowling document.write(obj["book"]["characters"][0] + "<br>"); // Prints: Harry Potter document.write(obj["book"]["price"]["hardcover"]); // Prints: $20.32

**try...catch**

JavaScript provides the try-catch statement to **trap,handle the runtime errors**, and handle them gracefully.

try {  
    // Code that may execute / cause an error  
} catch(error) {  
    // Action to be performed when an error occurs  
}

try {

var greet = "Hi, there!"; document.write(greet);

// Trying to access a non-existent variable

document.write(welcome);

// If error occurred following line won't execute

alert("All statements are executed successfully.");

} catch(error) {

// Handle the error alert("Caught error: " + error.message);

} // Continue execution

document.write("<p>Hello World!</p>");

**Throwing Errors**

Possible to throw an error manually by using the throw statement.

throw 123;

throw "Missing values!";

throw true;

throw { name: "InvalidParameter", message: "Parameter is not a number!" };

throw new Error("Something went wrong!");

**Boolean ( true/false)**

var a = true;

console.log(a); //true

var a = 17, b = 18, c = 20;

if(a<b){

console.log('not eligible'); //false

}else if(a<c){

console.log('not eligible'); //false

}else{

console.log('not eligible');

}

**Null (empty)**

**(Ex: water 🡪 empty 🡪 Milk ) ( Ex: Old Value 🡪Null 🡪 New Value)**

var a = "Jhon";

var b = "Doe";

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

console.log('before null'+' '+a);

a=null; //empty

a='new Jhon';

console.log('after null'+' ' +a);

emptyArray= [ ] //empty array

emptyObject={} //empty Object

**Undefined ( value not assigned to variable)**

var c;

console.log(c);

console.log(d); // d is *not defined* in object/document

**JavaScript is an Object Oriented Programming (OOP) language**. A programming language can be called object-oriented if it provides four basic capabilities to developers −

* **Encapsulation** − the capability to store related information, whether data or methods, together in an object.
* **Aggregation** − the capability to store one object inside another object.
* **Inheritance** − the capability of a class to rely upon another class (or number of classes) for some of its properties and methods.
* **Polymorphism** − the capability to write one function or method that works in a variety of different ways.

Objects are composed of attributes. If an attribute contains a function, it is considered to be a method of the object, otherwise the attribute is considered a property.

Attribute contains 🡪 composed Object

Attribute contains 🡪 Function 🡪 Method/Property of Object

1. **Object properties** can be any of the three **primitive data types**, or any of the abstract data types, such as another object. Object properties are usually variables that are used internally in the object's methods
2. **Object method** is attached to an object and can be referenced by the **this** keyword.
3. **All user-defined objects** and built-in objects are descendants of an object called **Object**. The **new** operator is used to create an instance of an object
4. A constructor is a function that creates and initializes an object. JavaScript provides a special constructor function called **Object()** to build the object. The return value of the **Object()** constructor is assigned to a variable.
5. The **‘with’** keyword is used as a kind of shorthand for referencing an object's properties or methods.

**Object ( store collections of data types** **)**

**(Ex: Reference = { employes : role } )**

var a = {

x:10,

y:5,

z: function(){

return this.x+ '' +this.y;

},

b:function(){

return this.x+ '' +this.y;

}

}

console.log(a.y);

console.log(a.z());

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

delete a.y; // Deleting property

a.a(); // Outputs: Peter

a["z"](); // Outputs: Peter

// Iterating over object properties

for(var i in a) {

document.write(a[i] + "<br>"); // Prints: x, y,z and b

}

// Setting a new property

a.country = "United States";

document.write(a.country); // Prints: United States

// Assign person variable to a new variable

var user = a;

user.name = "Harry";

document.write(user.name); // Prints: Harry

**Functions (group of statements)( Print return value)**

* **Functions reduces the repetition of code within a program**
* **Functions makes the code much easier to maintain**
* **Functions makes it easier to eliminate the errors**

**Returning Values from a Function**

// Defining function

function getSum(num1, num2) {

var total = num1 + num2;

return total;

}

// Displaying returned value

alert(getSum(6, 20)); // 0utputs: 26

// Function Declaration / Function Statement

Hoisted to top, can be declared anywhere in the file (i.e also after it is used)

function getSum(num1, num2) {

var total = num1 + num2;

return total;

}

**Working with Function Expressions**

Hoisted to top but not initialized/defined, can not be decalred anywhere in the file(i.e, not after its used)

// Function Expression

var getSum = function(num1, num2) {

var total = num1 + num2;

return total;

};

var sum = getSum(7, 25);

alert(sum); // 0utputs: 32

**Defining and Calling a Function**

// Defining function

function sayHello() {

alert("Hello, welcome to this website!");

}

// Calling function

sayHello(); // 0utputs: Hello, welcome to this website!

var greeting = function(){

return "Hello World!";

}

alert(typeof greeting) // function

**Adding Parameters to Functions**

// Defining function

function displaySum(num1, num2) {

var total = num1 + num2;

alert(total);

}

// Calling function

displaySum(6, 20); // 0utputs: 26

**Default Values for Function Parameters**

function sayHello(name = 'Guest') {

alert('Hello, ' + name);

}

sayHello(); // 0utputs: Hello, Guest

**Arrays (store more than one value/group of values)( Print return value)**

1. **Single array**
2. **Multiple array**
3. **Associated array**

var myArray = [*element0*, *element1*, ..., *elementN*];

var fruits = ["Apple", "Banana", "Mango", "Orange", "Papaya"]; // Iterates over array elements

for(var i = 0; i < fruits.length; i++) {

document.write(fruits[i] + "<br>"); // Print array element

}

pop() 🡪 To remove the last element from an array

shift() 🡪 To remove the first element from an array

splice() 🡪 To add or remove elements from any index

( arr.splice(startIndex, deleteCount, elem1, ..., elemN) )

join() 🡪 To create a string by joining the elements of an array

slice( ) 🡪 To extract out a portion of an array (arr.slice(startIndex, endIndex))

concat() 🡪 To merge or combine two or more arrays

indexOf() 🡪 To the method returns the first one found and To search an array for a specific value

lastIndexOf() 🡪 To the lastIndexOf() returns the last one found

includes() 🡪 This method to find out whether an array includes a certain element or not

find() 🡪 findIndex() 🡪 This method returns the value of the first element

filter()🡪 This method to find out all the matched elements

sort() 🡪 for sorting array elements in alphabetical order

reverse() 🡪 To reverse the order of the elements of an array.

apply() 🡪This method in combination with the Math.max() and Math.min() to find the maximum and minimum value inside an array

## JavaScript Arithmetic Operators

## + - \* / %

## JavaScript Assignment Operators

## = += -= \*= /= %=

## JavaScript String Operators

## + +=

## JavaScript Incrementing and Decrementing Operators

## ++x x++ --x x—

## JavaScript Logical Operators

## && || !

## JavaScript Comparision Operators

## == === != !== < > >= <=

## Different type of Loops in Javascript

## while — loops through a block of code as long as the condition specified evaluates to true.

1. **do…while** — loops through a block of code once; then the condition is evaluated. If the condition is true, the statement is repeated as long as the specified condition is true.
2. **for** — loops through a block of code until the counter reaches a specified number.
3. **for…in** — loops through the properties of an object.
4. **for…of** — loops over iterable objects such as arrays, strings, etc.

switch(expression) {  
  case x:  
    *// code block*    break;  
  case y:  
    *// code block*    break;  
  default:  
    // code block  
}

## Closures

## This is accomplished by creating a function inside another function. closures are the primary mechanism used to enable data privacy.

## function makeCounter() {

## var counter = 0; // Nested function

## function make() {

## counter += 1;

## return counter;

## }

## return make;

## }

## /\* Execute the makeCounter() function and store the returned value in the myCounter variable \*/

## var myCounter = makeCounter();

## console.log(myCounter()); // Prints: 1

## console.log(myCounter()); // Prints: 2

## Creating the Getter and Setter Functions

## we will create a variable secret and protect it from being directly manipulated from outside code using closure.

## var getValue, setValue;

## // Self-executing function

## (function() {

## var secret = 0;

## // Getter function

## getValue = function() { return secret; };

## // Setter function

## setValue = function(x) {

## if(typeof x === "number") {

## secret = x; } };

## }());

## // Calling the functions

## getValue(); // Returns: 0

## setValue(10);

## getValue(); // Returns: 10

## setValue(null);

## getValue(); // Returns: 10

## Function Hoisting

## function before it is defined, but the code still works.

## // Calling function before declaration

## sayHello(); // Outputs: Hello, I'm hoisted!

## function sayHello() {

## alert("Hello, I'm hoisted!");

## }

## alert(str); // Outputs: undefined

## var str; // Declare and initialize

## str = "Hello World!";

**ECMAScript 6 (or ES6)**

New features such as, **block-scoped variables**, **new loop for iterating over arrays and objects**, **template literals**

**1. block-scoped Variables :**

Let 🡪  if you declare a variable with the let keyword **inside a loop**,

it does not exist outside of the loop(console.log = undefined)

for(let i = 0; i < 5; i++) {

console.log(i); // 0,1,2,3,4

}

console.log(i); // undefined

**const**  🡪 Constants are **read-only**, you **cannot reassign new values** to them and you can still change **object properties or array** elements

const PI = 3.14;

console.log(PI); // 3.14

PI = 10; // error

#### ****Tuples 🡪****Tuples are a data type unique to TypeScript.  Think of them as arrays with a fixed number of elements.  This data type is best used when you know exactly how many variables you should have.  It is possible to reassign the value of the indices but not the amount of elements in the tuple.

Variables of data type tuple are declared just like an array:

// define our tuple  
let ourTuple: [number, boolean, string];  
// initialize correctly  
ourTuple = [5, false, 'Coding God was here'];  
// We have no type safety in our tuple for indexes 3+  
ourTuple.push('Something new and wrong');  
console.log(ourTuple);

let mine: [number, string];

#### ****Enum 🡪****Enums are a welcomed (in my humble opinion) addition to the data types.  Think of them as a more user friendly approach to giving names to numeric values.  Here is an example of an enum:

enum Foods {'bacon', 'tomato', 'lettuce'};

console.log(Foods[0]) // yields 'bacon'

console.log(Foods.bacon) // yields 0

console.log(Foods['lettuce']) // yields 2

enum Foods {'bacon'= 18, 'tomato', 'lettuce'};

console.log(Foods['bacon']); // 18

**for...of** 🡪  loops over **iterable objects such as arrays, strings**, etc..  loop is executed for each element of the iterable object

// Iterating over array

let letters = ["a", "b", "c", "d", "e", "f"];

for(let letter of letters) {

console.log(letter); // a,b,c,d,e,f

}

// Iterating over string

let greet = "Hello World!";

for(let character of greet) {

console.log(character); // H,e,l,l,o, ,W,o,r,l,d,!

}

**for...in** 🡪  loops through the properties of an object. If you want to **iterate over the properties of an object** you can use

let arr = [10, 20, 30, 40];

for (var index in arr) {

console.log(index); // prints indexes: 0, 1, 2, 3

console.log(arr[index]); // prints elements: 10, 20, 30, 40

}

## 2.Template Literals

## back-tick (` `) (grave accent)

## // Simple multi-line string

## let str = `The quick brown fox jumps over the lazy dog.`; // String with embedded variables and expression

## let a = 10; let b = 20;

## let result = `The sum of ${a} and ${b} is ${a+b}.`; console.log(result); // The sum of 10 and 20 is 30.

## 3. Default Values for Function Parameters

## function when it is called these default parameters values will be used

## function sayHello(name='World') {

## return `Hello ${name}!`;

## }

## console.log(sayHello()); // Hello World!

## console.log(sayHello('John')); // Hello John!

## 4.Arrow Functions

## [function expressions](https://www.tutorialrepublic.com/javascript-tutorial/javascript-functions.php#function-expressions) by opting out the function and return keywords. the fat arrow (=>) notation.

## // Function Expression

## var sum = function(a, b) { return a + b; } console.log(sum(2, 3)); // 5

## // Arrow function

## var sum = (a, b) => a + b; console.log(sum(2, 3)); // 5

## 5. this  🡪 the current execution context of a function

function Person(nickname, country) {

this.nickname = nickname;

this.country = country;

// Function Expression

this.getInfo = function() {

// Outer function context (Person object)

return () => {

// Inner function context (Person object)

alert(this.constructor.name); // Person

alert(`Hi, I'm ${this.nickname} from ${this.country}`);

};

}

}

let p = new Person('Rick', 'Argentina');

let printInfo = p.getInfo();

printInfo(); // Hi, I'm Rick from Argentina

## Classes

**ES6 classes make it easier to create objects, implement inheritance by using the extends keyword, and reuse the code.**

**class** 🡪 Class have *Class constructor and Class method.*  To create objects, implement inheritance by using the extends keyword, and reuse the code. This keyword followed by a class-name.

**ClassName** 🡪 (i.e. capitalizing the first letter of each word).

**Extends** 🡪 *Child class inherits from Parent*, Classes that inherit from other classes are referred to as derived classes or child classes

**super**() 🡪  *Call parent's constructor* and Square class inherits from the Parent class

**constructor**() 🡪 method is a special method for *creating and initializing an object* created with a **class**

class Rectangle {

// Class constructor

constructor(length, width) {

this.length = length;

this.width = width;

}

// Class method

getArea() {

return this.length \* this.width;

} }

// Square class inherits from the Rectangle class

class Square extends Rectangle {

// Child class constructor

constructor(length) {

// Call parent's constructor

super(length, length);

}

// Child class method

getPerimeter() {

return 2 \* (this.length + this.width);

} }

let rectangle = new Rectangle(5, 10);

alert(rectangle.getArea()); // 50

let square = new Square(5);

alert(square.getArea()); // 25

alert(square.getPerimeter()); // 20

alert(square instanceof Square); // true

alert(square instanceof Rectangle); // true

alert(rectangle instanceof Square); // false

## Modules

## module, in which each module is represented by a separate .js file. you can use the export or import statement in a module to export or import variables, functions, classes or any other entity to/from other modules or files.

## <script type="module" src="app.js"></script>

## main.js

## let greet = "Hello World!";

## const PI = 3.14;

## function multiplyNumbers(a, b) { return a \* b; } // Exporting variables and functions

## export { greet, PI, multiplyNumbers };

## app.js

## import { greet, PI, multiplyNumbers } from './main.js';

## alert(greet); // Hello World!

## alert(PI); // 3.14

## alert(multiplyNumbers(6, 15)); // 90

## The Rest Parameters

## A rest parameter is specified by prefixing a named parameter with rest operator (...) i.e. three dots

## function sortNames(...names) { return names.sort(); } alert(sortNames("Sarah", "Harry", "Peter")); // Harry,Peter,Sarah

## function myFunction(a, b, ...args) { return args; } alert(myFunction(1, 2, 3, 4, 5)); // 3,4,5

## The Spread Operator

## The spread operator, which is also denoted by (...), performs the exact opposite function of the rest operator.

## function addNumbers(a, b, c) {

## return a + b + c;

## }

## let numbers = [5, 12, 8]; // ES5 way of passing array as an argument of a function

## alert(addNumbers.apply(null, numbers)); // 25

## // ES6 spread operator

## alert(addNumbers(...numbers)); // 25

## Destructuring Assignment

## To get an individual value of an array

## let fruits = ["Apple", "Banana"];

## let [a, b] = fruits; // Array destructuring assignment

## alert(a); // Apple

## alert(b); // Banana

## let person = {name: "Peter", age: 28};

## let {name, age} = person; // Object destructuring assignment

## alert(name); // Peter

## alert(age); // 28

## Angular JS

## Install with node commond

## npm install -g @angular/cli

## ng new my-dream-app

## cd my-dream-app

## ng serve

## Commands

## *ng g c componentname*

## *ng g s servicename*

## *npm i bootstrap*

## *npm install jquery*

## *ng build –prod*

## *node server.js*

## cntrl 🡪 space (import line)

## routeLink

## Angular.json 🡪 main.ts 🡪 app.module.ts

* "main": "src/main.ts",
* platformBrowserDynamic().bootstrapModule(AppModule)
* bootstrap: [AppComponent]
* *Just-in-Time* (JIT), compiles your app in the browser at runtime.

## ****just-in-time**** (****JIT****) ****compilation**** (also **dynamic translation** or **run-time compilations**) is a way of executing [computer code](https://en.wikipedia.org/wiki/Computer_code) that involves [compilation](https://en.wikipedia.org/wiki/Compiler) during execution of a program — at [run time](https://en.wikipedia.org/wiki/Run_time_(program_lifecycle_phase)) — rather than prior to execution.

* *Ahead-of-Time* (AOT), compiles your app at build time on the server.

## An ahead-of-time (AOT) compiler converts your code during the build time before the browser downloads and runs that code.

## ng build --prod to build source code bundles which includes assets, JS files (main, vendor, and polyfills), index.html, and CSS. In this step, Angular uses the [Angular compiler](https://www.npmjs.com/package/@angular/compiler-cli) to build source code and they do it in [3 phases](https://angular.io/guide/aot-compiler#compilation-phases) which are code analysis, code generation, and template type checking. In this step, the bundle size will be smaller than bundle size when we build by JIT mode.

## **#1**Interpolation: **{{value}}** **#2**Propertybinding: **[property]=”value”** **#3**Eventbinding: **(event)=”function”** **#4** Two-way data binding(PE): **[(ngModel)]=”value”**

**onInit**

Angular has initialized all data-bound properties of a directive.

[**NgModule**](https://angular.io/api/core/NgModule)

The @[NgModule](https://angular.io/api/core/NgModule) decorator identifies **AppModule** as an [**NgModule**](https://angular.io/api/core/NgModule) class. @[**NgModule**](https://angular.io/api/core/NgModule) takes a metadata object that tells Angular how to compile and launch the application.

* **declarations**—this application's lone component.
* **imports**—import [*BrowserModule*](https://angular.io/api/platform-browser/BrowserModule) to have browser specific services such as DOM rendering, sanitization, and location.
* **providers**—the service providers.
* **bootstrap**—the root component that Angular creates and inserts into the index.html host web page.

## 

## Login Form Process

## FormBuilder initialization in constructor

## With expression function initialize form fields from FormBuilder in ngOnInit

## onSubmit Button get form values in Object and this Object calling in Services response from Services.ts file

## FormsModule and ReactiveFormsModule

1. **Template-driven** forms make use of the "**FormsModule(asynchronous**)", while **Reactive forms** are based on "**ReactiveFormsModule(synchronous**)".
2. **Template-driven** forms are **asynchronous** in nature, whereas **Reactive forms** are mostly **synchronous**.
3. In a template-driven approach, most of the logic is driven from the template, whereas in reactive-driven approach, the logic resides mainly in the component or typescript code. Let us get started by generating a component and then we'll update our form code.
4. [***https://www.pluralsight.com/guides/difference-between-template-driven-and-reactive-forms-angular***](https://www.pluralsight.com/guides/difference-between-template-driven-and-reactive-forms-angular)
5. [***https://www.cloudhadoop.com/2018/08/typescript-how-to-convert-object-to.html***](https://www.cloudhadoop.com/2018/08/typescript-how-to-convert-object-to.html)
6. [***https://blog.angular-university.io/introduction-to-angular-2-forms-template-driven-vs-model-driven/***](https://blog.angular-university.io/introduction-to-angular-2-forms-template-driven-vs-model-driven/)
7. [***https://coryrylan.com/blog/angular-form-builder-and-validation-management***](https://coryrylan.com/blog/angular-form-builder-and-validation-management)
8. [***https://www.pluralsight.com/guides/how-to-display-validation-messages-using-angular***](https://www.pluralsight.com/guides/how-to-display-validation-messages-using-angular)

## FormsModule

## Template – driven forms

## In Module.ts.file

## FomsModule (adding)

## In Add Form tag

## (ngSubmit)="onSubmit(f)" #f="ngForm"

## In TS file

## @ViewChild('f') courseForm: NgForm;

## Html file

## <input type="text" id="courseName" class="form-control" name="courseName" ngModel required #courseName="ngModel">

## Validations

## <div style="color:red" \*ngIf="courseName.errors && (courseName.dirty || courseName.touched)">

## <p \*ngIf="courseName.errors.required"> Course Name is required </p>

## </div>

## fb: FormBuilder

## The [FormBuilder](https://angular.io/api/forms/FormBuilder) provides syntactic sugar that shortens creating instances of a [FormControl](https://angular.io/api/forms/FormControl), [FormGroup](https://angular.io/api/forms/FormGroup), or [FormArray](https://angular.io/api/forms/FormArray)

## Group() 🡪 Construct a new [FormGroup](https://angular.io/api/forms/FormGroup) instance ( this.fb.group )

## Control() 🡪 Construct a new [FormControl](https://angular.io/api/forms/FormControl) with the given state, validators and options.

## Array() 🡪 Constructs a new [FormArray](https://angular.io/api/forms/FormArray) from the given array of configurations, validators and options.

## Modules Added In Application

## BrowserModule (<https://angular.io/api/platform-browser/BrowserModule>)

browser specific services such as DOM rendering, sanitization, and location.

## NgModule (<https://angular.io/guide/ngmodules>)

## configure the injector and the compiler and help organize related things together. An NgModule describes how the application parts fit together.

## HttpClientModule (https://angular.io/api/common/http/HttpClientModule)

## Configures the [dependency injector](https://angular.io/guide/glossary#injector) for [HttpClient](https://angular.io/api/common/http/HttpClient) with supporting services for XSRF

## AppRoutingModule (<https://angular.io/tutorial/toh-pt5>)

## Routes tell the Router which view to display when a user clicks a link or pastes a URL into the browser address bar.

## AppComponent (<https://angular.io/guide/bootstrapping>)

## The default application created by the Angular CLI only has one component, AppComponent, so it is in both the declarations and the bootstrap arrays.

## Router (<https://angular.io/guide/router>)

## The [Router](https://angular.io/api/router/Router) enables navigation by interpreting a browser URL as an instruction to change the view.

## Component, OnInit, ViewChild, AfterViewInit (https://angular.io/guide/lifecycle-hooks)

## The lifecycle continues with change detection, as Angular checks to see when data-bound properties change, and updates both the view and the component instance as needed

## empty, from

## FormBuilder, FormGroup, Validators (<https://angular.io/api/forms/FormBuilder>)

## The [FormBuilder](https://angular.io/api/forms/FormBuilder) provides syntactic sugar that shortens creating instances of a [FormControl](https://angular.io/api/forms/FormControl), [FormGroup](https://angular.io/api/forms/FormGroup), or [FormArray](https://angular.io/api/forms/FormArray). It reduces the amount of boilerplate needed to build complex forms.

## ActivatedRoute (https://angular.io/api/router/ActivatedRoute)

## Provides access to information about a route associated with a component that is loaded in an outlet. Use to traverse the [RouterState](https://angular.io/api/router/RouterState) tree and extract information from nodes.

## HttpClient (<https://angular.io/guide/http>)

## Most front-end applications need to communicate with a server over the HTTP protocol, in order to download or upload data and access other back-end services.

## Injectable (<https://angular.io/api/core/Injectable>)

## Decorator that marks a class as available to be provided and injected as a dependency.

## Environment

## An Angular Application Environment is JSON configuration information that tells the build system which files to change when you use ng build and ng serve

## 

## app.modules

## import { BrowserModule } from '@angular/platform-browser';

## import { NgModule } from '@angular/core';

## import { FormsModule, ReactiveFormsModule} from '@angular/forms';

## import { HttpClientModule } from '@angular/common/http';

## import { AppRoutingModule } from './app-routing.module';

## import { AppComponent } from './app.component';

## User Services modules/component

## import { HttpClient } from '@angular/common/http';

## import { Injectable } from '@angular/core';

## import { environment } from '../environments/environment';

## *private httpclient:HttpClient*

## *return this.httpclient.post(environment.baseUrl, RegObj);*

## *return this.httpclient.get('https://www.w3schools.com/angular/customers.php');*

## *userSignup(RegObj: any) {*

## *const contentHeaders = new HttpHeaders();*

## *contentHeaders.append('Authorization', 'Your token used in app');*

## *contentHeaders.append('Content-Type', 'application/json');*

## *contentHeaders.append('Access-Control-Allow-Origin', 'http://localhost:4200');*

## *return this.httpclient.post('http://localhost:8080/api/auth/signup', RegObj, { headers: contentHeaders });*

## *}*

## Login.modules/component

## import { Component, OnInit, ViewChild, AfterViewInit } from '@angular/core';

## import { FormBuilder, FormGroup, Validators } from '@angular/forms';

## import { empty, from } from 'rxjs';

## import { UserService } from '../user.service';

## import { EditdataComponent } from '../editdata/editdata.component';

## import { Router } from '@angular/router';

## import { ActivatedRoute } from '@angular/router'; //For Updated module

## *private fb: FormBuilder, private uservice: UserService, private router: Router, private route: ActivatedRoute*

## *this.loginForm = this.fb.group({ })*

## *this.uservice.getUserDetails().subscribe((response: any) => {*

## *this.responsedata = response.records;*

## *this.obj1.push(this.responsedata);*

## *}) }*

## *localStorage.setItem('localStorage-editValue', JSON.stringify(editValue));*

## *this.router.navigate(['/editdata', editValue.Name]);*

## *let index = this.responsedata.findIndex(x => x.Name === updatedData);*

## Signup.modules/component

## import { Component, OnInit } from '@angular/core';

## import { FormBuilder, FormGroup, Validators } from '@angular/forms';

## import { UserService } from '../user.service';

## *private fb:FormBuilder, private uservice:UserService*

## *localStorage.setItem('userObj', JSON.stringify(RegObj));*

## *let getLocal = JSON.parse(localStorage.getItem('userObj'));*

## *for(let i=0; i<=getLocal.length; i++){*

## *this.localArray.push(this.getLocal[i]);*

## *}*

## *throw new Error('Method not implemented.');*

## editdata.modules/component

## import { Component, OnInit } from '@angular/core';

## import { FormBuilder, FormGroup, Validators } from '@angular/forms';

## import { ActivatedRoute } from '@angular/router';

## import { UserService } from '../user.service';

## *private fb: FormBuilder, private route: ActivatedRoute, private uservice: UserService*

## Method : 1

## *let Name = this.route.snapshot.paramMap.get('name');*

## *this.uservice.getUserDetails().subscribe((response: any) => { …})*

## *this.responsedata.filter((filterData: any) => { ….})*

## *this.editDataForm.patchValue({…})*

## *this.responsedata.filter((filterData: any) => {*

## *if (filterData.Name == updatedData.Name) {*

## *let index = this.responsedata.findIndex(x => x.Name ===updatedData.Name);*

## *this.responsedata.splice(index,1, updatedData);*

## *}*

## *})*

## Method : 2

## *this.userId = this.route.snapshot.paramMap.get('id');*

## *console.log('ID', this.userId);*

## *if (this.userId) {*

## *debugger;*

## *this.getUserById(this.userId)*

## *}*

## *else {*

## *this.toaster.error('id not available');*

## *}*

## *}*

## *getUserById(userId: any) {*

## *let userIdObj = {*

## *'userId': userId*

## *}*

## *this.userserve.getUsersDetailsById(userIdObj).subscribe((response: any) => {*

## *this.responseData = response.results;*

## *this.Location = this.responseData.Location;*

## *console.log('this.responsedata...', this.responseData);*

## *this.editDataForm.patchValue({*

## *'username': this.responseData.username,*

## *'email': this.responseData.email,*

## *'address': this.responseData.address,*

## *'pincode': this.responseData.pincode*

## *})*

## *})*

## *}*

## Get users-details

## *this.userserve.getUsersDetails().subscribe((response:any)=>{*

## *this.responsedata = response.result;*

## *console.log('get users data', this.responsedata);*

## *});*

## Get users-details

## *userEdit(id: any) {*

## *console.log('user Id', id);*

## *localStorage.setItem('localStorage-editValue', JSON.stringify(id));*

## *this.router.navigate(['/userEdit', id]);*

## *}*

## Delete User Row

## *deleteDataRow(updatedData) {*

## *debugger*

## *console.log('updatedData...', updatedData);*

## *let index = this.responsedata.findIndex(x => x.Name === updatedData);*

## *this.obj1=[];*

## *this.responsedata.slice(index);*

## *this.obj1.push(this.responsedata);*

## *}*

## Get Dashboard-details (From Login)

## *this.userData = localStorage.getItem('userObj');*

## *this.parsedata = JSON.parse(this.userData);*

## *console.log('retrievedObject: ', this.parsedata);*

## File Image upload

## *imageUpload(event: any) {*

## *const file = event.target.files && event.target.files[0];*

## *if (file) {*

## *let formdata = new FormData();*

## *formdata.append('file', file);*

## *this.userserve.userImage(formdata).subscribe((response: any) => {*

## *this.responsedata = response;*

## *this.toaster.success(this.responsedata);*

## *this.Key = response.data.key;*

## *this.Location = response.data.Location;*

## *}, (error: any) => {*

## *alert(error.message);*

## *})*

## *}*

## *else {*

## *return;*

## *}*

## *}*

## Disabled button with Checkbox

## *checked() {*

## *let check = this.signup.controls['checkboxTerms'].value*

## *console.log('value..', check);*

## *if (check) {*

## *this.checkedBtn = false;*

## *}*

## *else {*

## *this.checkedBtn = true;*

## *}*

## *}*

## Toaster messages Script

## *if (this.responsedata.status == 200) {*

## *this.toaster.success(this.responsedata.message);*

## *}*

## *if (this.responsedata.status == 400) {*

## *this.toaster.error(this.responsedata.message);*

## *}*

## Routing Script

## this.router.navigate(['/login'])

## <a [routerLink]="['product/detail/1']">Product 1 Overview</a>

## Common Script

## this.obj1=[];

## this.responsedata.slice(index);

## this.obj1.push(this.responsedata);

## 

## breakpoints: (

## xs: 0,

## sm: 576px,

## md: 768px,

## lg: 992px,

## xl: 1200px,

## xxl: 1400px

## );

## HOST: "localhost",

## USER: "root",

## PASSWORD: "Root@12345",

## git status

## git add .

## git commit -m "Comments"

## git checkout -b "Login-Dashaboard-added"

## git push origin Login-Dashaboard-added

sendMessage

OtpVerification

## Product Image

## Product Type

## Product name

## Quantity (50g, 100g, 250g, 500g, 1000g )

## Quantity Type (gram, Litre, Number)

## Price (50g, 100g, 250g, 500g, 1000g )

**Always keep in mind**: **NO** operator (neither ===, > etc. nor && or ||) changes the variable you might be using in the comparison. In the above examples, the values stored in userName and altName **are NEVER changed**.

===, > etc. just **generate** **new boolean values** which are used in the comparison. || and && **generate NO booleans**, they just treat the **values** **before and after them as conditions** (which therefore need to yield boolean values and are coerced to booleans if required).

Because of the above-described behaviors, you often use || in JavaScript to assign default/ fallback values to variables/ constants:

1. const enteredValue = ''; // let's assume this is set based on some input provided by the user, therefore it might be an empty string
3. const userName = enteredValue || 'PLACEHOLDER'; // will assign 'PLACEHOLDER' if enteredValue is an empty string

## Important Notes -- Shortcuts

## If , else If conditions 🡪 switch or ternary expression

## let isLoggedIn = true; // undefine

## const shoppingCart = isLoggedIn && [‘Books’]; 🡺 true && console

## alert(shoppingCart) // && [‘Books’];

## let isLoggedIn = false; // undefine

## const shoppingCart = isLoggedIn && [‘Books’]; 🡺 false && console

## (first is true/false second value trigger)

## alert(shoppingCart) // false

## const userInput = ‘ ‘ ; //undefined

## const isValidInput = !! userInput // undefined 🡪 false

## alert(isValidInput) // false

## const isValidInput = ! userInput // undefined 🡪 true

## for loop 🡪 Execute code a certain amount of times(with counter variable)

## for(let I = 0; i< 3; i++){ console.log(i);

## for-of loop 🡪 Execute for every element in an array

## for (const el of array) { console.log(el) }

## for-in loop 🡪 Execute for every key in an Object

## for (const key in obj){ console.log(obj[key]); }

## while lopp 🡪 Execute code as long as a condition is true

## while(isLoggedIn){ ……. }

## let randomNumbers = [];

## let finished = false;

## while(!fiinshed){

## const rndNumber = Math.random();

## randomNumbers.push(rndNumbers);

## if(rndNumbers >0.5){

## finished = true;

## console.log(randomNumbers);

## }

## }

## ! lastLogEntry && lastLogEntry !==0 || lastLogEntry < I 🡺 0(false) && 1(true) || var < 1

## Controlling Loop 🡪 break, continue

## for(let I =0; i<5; i++){

## if (i===3) {

## break;

## }

## console.log(i);

## }

## for(let I =0; i<5; i++){

## if (i===3) {

## continue;

## }

## console.log(i);

## }

## The idea is that you wrap try-catch around the code that might fail.

## 

## 0, “ ” (empty string), Null, Nan, Undefined, 🡪 False

## Any number(+ or -), Any other non-empty string (incl. “false”), {}, [], all objects or arrays 🡪 True

## Var 🡪 crates a variable 🡪 Available since .. ever 🡪 Function & Global scope

## Let 🡪 creates a variable 🡪 Available since ES6 🡪 Block Scope

## Const 🡪 Creates a constant 🡪 Available since ES6 🡪 Block Scope

The following resources may be helpful.

* Control Structures (MDN): <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling>
* JavaScript Loops (MDN): <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration>

## Browsers have javacript engine.

## Every Major browser has a Javascript engine. The engine of

## Google browser chrome for example is named 🡪 v8

## Firefox its called 🡪 Spider Monkey

## And these engines do that parsing and execution and they typically consist of two parts 🡪 Interpreter, Compiler(Just-in-time)

## Code 🡪 Parses script & starts execution 🡪 Interpreter basically load our script, reads it. It then kind of translates it to byte code which is a bit easier to excecute you could say and then it goes ahead and starts running our script

## Code 🡪 Parse script & starts execution 🡪 Well the interpreter start executing our script but it does so line-by-line in an optimized way, which means the script execution works but is far from being as fast as possible. To have the best possible performance. Compile it to machine code and hand it off to your machine, to your operating system if you will because that will always be faster and that translation of your interpreted javascript code to machine code is exactly what the compiler does.

## So interpreter does not just start executing your script, it also hands off the byte code to the compiler, so this loaded script in the end, it hands this off to the compiler.

## Now the compiler is also built into the browser, so its part of the JS engine the browser.

## So the compiler, whilst the interpreter already started execution of your script, now compiles the script.

## Hence Just in Time compilation : The compiler starts compiling + executing the compiled code whilst the code is being read/executed your computer.

## Communicatin Bridges between javascript and (C++) Logic built into the Browser.