1. What are the basic requirements for building an End to End Web Application?

- HTML
- CSS
- Client Side Script

  JavaScript, TypeScript
- Server Side Script

  JSP, PHP, Node JS, ASP
- Database

  Oracle, MySql, MongoDb
- Middleware

  Express , JsX
- Publishing Tools

  Photoshop, Flash
- IDE's

  Eclipse, WebStrom, VS code, sublime

  Edit ..

"Tim Berners Lee" introduced - Web - HTML - 1990

What are the Challenges in modern Web Development?

> 90 % of web users
  are using web from smart devices
  [mobile, tabs]

1. Unified UX

- Unified User Experience

- An application should have same feel, appearence and functionality across any device.

- Application should not optimize.

- Mobile users must get access to everything.

2. Fluid UX

- User will stay on only one page and can get access to everything on to the page.

- New details are added to page without reloading the complete page.

3. Loosely coupled and extensible

- Without disturbing the existing application we must able to push in new components.

4. Simiplified Deployment and upgrades

What is the solution?

- Better build SPA or Progressive web application instead of tradional web applications.

How to build SPA and Progressive Application? Can we continue with HTML, CSS, JavaScript and JQuery?

(or)

Can we build SPA and Progressive web Applications using JavaScript and JQuery?

"Yes"

Why we are looking into New Technologies for building SPA and Progressive Web Application? What are the issues with JavaScript and JQuery?

1. Lot of DOM manipulations.

   [DOM - Document Object Model]

2. Lot of references

3. Lot of coding

4. Heavy pages

5. Slow Rendering

6. Navigation issues

7. Data Binding issues [depend on lot of events]

8. JavaScript is a Language and JQuery is an Library

   [Language and Library can build Application but can't

   control application flow]

"Better use a Framework"

- A framework is an software architectural pattern that can build the application and also control the application flow.

Do we have any Framework to use client side?

"Yes"

- Knockout JS
- Ember Js
- Vue
- Backbone Js
- Angular Js

Client Side "MVC, MVP, MVVM"

Angular JS is an open source and cross platform framework introduced by Google and maintained by a large community of organizations and developers to address the issues in SPA.

Key Features of Angular

- Angular is Cross Platform

*It uses frameworks like Cordova, Ionic, NativeScript

*You can build cross platform mobile apps.

*You can also build cross platform desktop installed apps.

[window, Mac and others ...]

*Angular application can run on any device and any OS.

- Modular and Asynchronous

* Only the library that is required for the current situation

is loaded, which means that it is not using Legacy library.

* Application specific framework will be used.

* Asynchronous uses non-blocking technique

[Makes a request and will not wait for response, it proceeds to next]

* It make application perform fast.

- Frameworks for Building App

* It can use frameworks like MVC, MVP, MVVM client side

* MVC, MVP, MVVM - Reusability, Extensibility, Testability

Maintainability.

* Separtion and Resuability concerns
* These frameworks compliment SPA and Progressive apps
* These frameworks will resolve data binding issues.

- Speed and Performance

* As it is modular it is light weight
* Less startup time
* Uses only the framework required for the situation
* Angular application run with 10x speed than angular js app.
* It can uses techniques like

    a) Code Splitting

    b) Dynamic Loading

* Angular 9 is now using IVY a rendering engine which can provide the stratergy of "Build-Run"
* Angular provide tools like CLI : building fast, adding component and tests and deploy with simple commands.

- End to End Solution

* It provides all the tools required for a developer to build, debug, test and deploy applications.
* Angular supports high level animations [Html, CSS]
* Angular supports plugins and widgets by using "Materials", which are ready to integerate and deoply.
* Pre-defined templates to implement.

Angular - MVC, MVVM, MVP

Model, View, Controller

Model - Data [Data- Client Side]

Views- UI

Controller - Application Logic

TypeScript

Problems in JavaScript:

- JavaScript is not a strongly typed language.

```
var x = 10;   // x is initialized with 10 - number
x = "John"; // valid
```

- JavaScript allows contradictionary values.
- Explicitly developer have to handle types and their restrictions.
- JavaScript supports OOP but not completely OOP language.
- Contracts, Templates, Code Security, Dynamic Polymorphism are not completely supported.

```
class JavaScriptClass {
 private name = "John"; // private is not supported
}
```

- JavaScript is not strictly typed by default. You have to turn on strict mode explicitly.

Strictly Typed = Follow Coding Rules

[Visit PMD or Sonar for Coding Rules]

https://pmd.github.io/

- JavaScript handles schema less data, which will a problem when you are dealing with Schema based.

- Google Angular Team Decided to develop a new Language in order overcome the issues with JavaScript and they started a script called "AtScript" in 2013

- Microsoft already developed a similar script for their ".NET" languages called "TypeScript".

- TypeScript designed by "Anders Hejlberg" who developed a language called "C#" for Microsoft.

- TypeScript is an Open Source and Cross platform language development by Microsoft and it is being used by Angular and other technologies related to web.

- TypeScript is strictly superset of JavaScript.

- TypeScript is in strict mode by default.

- TypeScript is Strongly Typed. [Duck Typing]

- TypeScript is an OOP. It supports all features of OOP.

- TypeScript transcompiles everything into JavaScript.

- TypeScript speeds up your development experience by catching errors and providing fixes before you even run your code.

- TypeScript runs on any browser, any OS, and any where JavaScript runs.

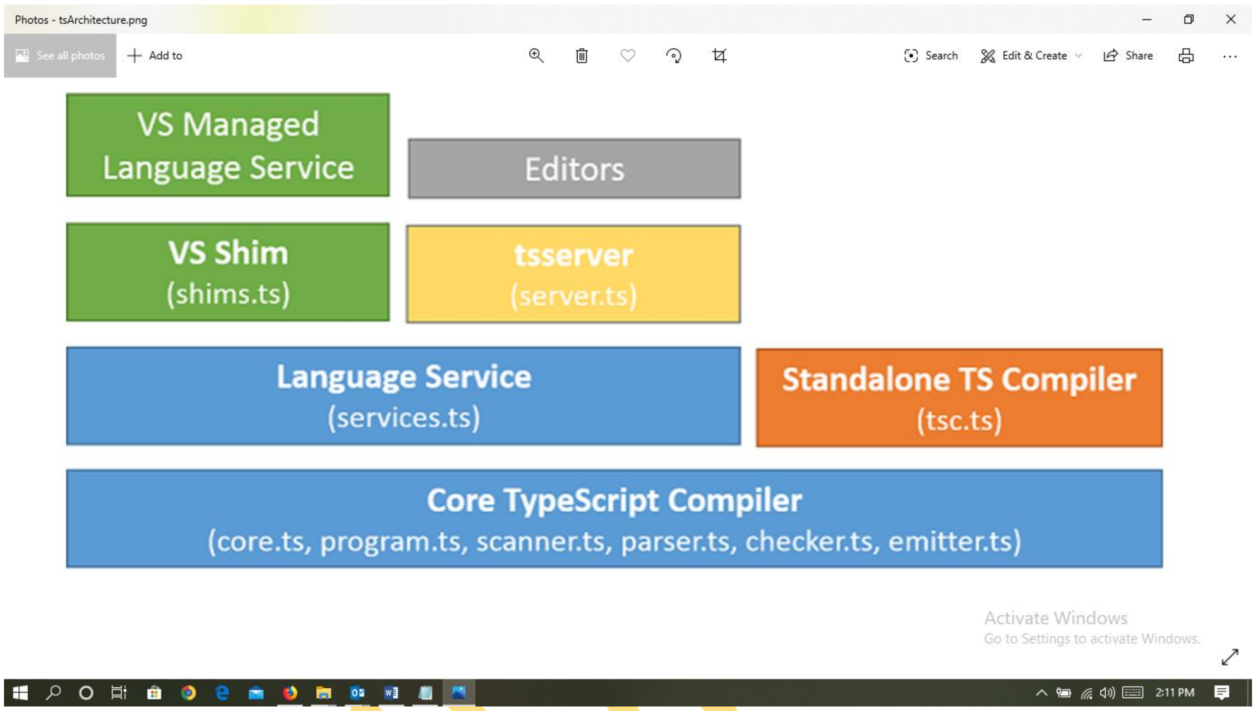
- TypeScript supports low level features. So that it can directly interact with hardware services.

- It can operate on very low memory devices.

- TypeScript is completely built by using TypeScript.

- TypeScript can use all JavaScript library.

TypeScript Architecture


Photos - tsArchitecture.png
See all photos
Add to
Search
Edit & Create
Share
VS Managed Language Service
Editors
VS Shim (shims.ts)
tsserver (server.ts)
Language Service (services.ts)
Standalone TS Compiler (tsc.ts)
Core TypeScript Compiler (core.ts, program.ts, scanner.ts, parser.ts, checker.ts, emitter.ts)
Activate Windows
Go to Settings to activate Windows.
2:11 PM

What are the Features of TypeScript?

TypeScript Architecture

[TypeScript built by using TypeScript]

Compiler : It used as translator

1. Core TypeScript Compiler

- Developer write code in TypeScript
- Core TypeScript compiler verifies the keywords,
  Syntax, Blocks, DataTypes, memory allocation.
- It reports the errors.

- If it is compiled successfully then it is ready for transcompiling.
- The key components in TypeScript core compiler are:

    core.ts
    program.ts
    scanner.ts
    emitter.ts
    parser.ts
    checker.ts

2. Standalone Compiler

- It is a compiler used to transcompile the type script code into JavaScript.
- "TSC" is a transcompiler for TypeScript, which generates a JavaScript file.

demo.ts → tsc.ts [compiler] → demo.js

Service: It is a pre-defined business logic with re-usable functions.

3. TypeScript Language Service: It provides all the repository [library] for typescript language. Keywords, functions, syntax all are verified from language service.

"services.ts"

ex:

array.length = "A"; // invalid

[number] = can't assign string

4. tsserver [server.ts]: It is responsible for hosting , compiling, processing and generating output for typescript programs. It is the location where typescript programs are compiled and processed.

5. VS Shim: It is responsible for translating your typescript code into platform independent code. [byte code]

6. VS Managed Language Service: It contains the library, which is cross platform.

7. Editors: Contains the rules for an IDE [Integrated Development Environments] in order to handle typescript. The IDE can be

Visual Studio Code

Sublime

Web Strom

DW, Eclipse etc..

Settingup Environment for TypeScript

1. Install Node JS on your PC

- We are installing Node Js for a Package Manager called "NPM"

- Package Manager is an software solution for installing library on your PC.
- Various Package Managers are used by developers like
    - NPM, Bower, Ruby Gems, NuGet etc..

a) Visit

https://nodejs.org/en/download/

b) Download ".msi" for your windows PC

c) Install from ".msi"

Note: Install Node JS 10+

e) Verify Installed Node JS and NPM

- Open Command Prompt

```
C:\> node -v
C:\> npm -v
```

2. Install TypeScript [Latest version is 3.9]

- Open Command Prompt
- Type the command

```
>npm install -g  typescript@latest
```

g [Global]

- Check the installed version by using the command

```
> tsc -v
```

Working with TypeScript in a WebSite:

1. Make sure that your computer have a Web Server

    Tomcat, IIS, [Internet Information Services]

- Every windows PC have a web server called IIS
- Goto Control Panel
- Open Administrative Tool
- Look for "Internet Information Services Manager"

Note: It IIS is missing in your Control Panel then Add from "Programs and Features"

- Goto Programs and Features
- Click on "Turn Windows Features ON or OFF
- Select "Internet Information Services"
- OK

2. Test your WebServer
    - Open Browser
    - Type the following URL

http://localhost (or) http://127.0.0.1

3. Create a new Website on Local Server

[Technically Web is a Virtual Directory on WebServer]

- Open Local IIS

  Run → inetmgr

  [Control Panel → Administrative Tools → IIS]

- Expand Local Computer [http://localhost]

- Expand "Sites" folder

- Right Click on "Default Website"

- Select the option

  "Add Virtual Directory" [Website]

  Alias : [SiteName] : TypeScriptOnlineProject

  Physical Path : C:\TypeScriptOnlineProject

- OK

Site Virtual Path:

http://localhost/TypeScriptOnlineProject

Physical Path:

C:\TypeScriptOnlineProject

4. Create an Index Page for Website

- Open NotePad and Create new HTML document

```
      <!DOCTYPE html>
<html>
```

```
<head> <title> TypeScript | Home </title> </head>
<body>
  <h2> TypeScript Online Project </h2>
</body>
</html>
```

- Save into website Physical Path by name
  "index.html"

5. Lets Create a TypeScript file for Project

- Create a new folder in Website Physical Path
  "code"
- Open Notepad and type the "TypeScript code"

```
var msg = "Welcome to TypeScript";
document.write(msg);
```

- Save into "Code" folder by name
  "Welcome.ts"
- Transcompile TypeScript file into JavaScript

```
C:\TypeScriptOnlineProject\Code> tsc welcome.ts
```

- This Generates "welcome.js"

- Use Welcome.js in your HTML Page.

```
<!DOCTYPE html>
<html>
<head> <title> TypeScript | Home </title>
  <script src="code/welcome.js"> </script>
 </head>
<body>

</body>
</html>
```

IDE [Integrated Development Environment] for TypeScript

- IDE provides a platform for developer to build, debug, test and deploy application.

- You can use various IDEs

  Sublime

  WebStrom

  Visual Studio Code

  Visual Studio

  Eclipse etc..

Get Visual Studio Code and Install from

  https://code.visualstudio.com/

1. Open Visual Studio Code IDE

2. Install support for JavaScript

[You can find in Home Screen Customize panel]

3. Goto "Extentions" and install following extentions

    [Ctrl + Shift + X]

    a) TsLint

    b) Live Server

    c) vscode-Icons

4. Open your project in VS Code

    - File → Open → Choose your website phyical path

        C:\TypeScriptOnlineProject

5. Open Index.html

    - Right Click in Index.html

    - Open with Live Server

6. Goto "Code" folder and add a new File [+]

        "hello.ts"

```
var msg = "Hello ! from TypeScript";
document.write(msg);
```

7. Open Terminal [Terminal Menu → New Terminal]

    Ctrl + ` [backtick]

8. Compile hello.ts

```
> tsc hello.ts
```

Note: If Terminal is using "PowerShell" and it is unable to execute "tsc" then:

- Goto Terminal Window
- Choose the option "Select Default Shell"
- Choose "Command Prompt [cmd.exe] as your terminal shell.
- Then click on New Terminal [+] icon
- In new terminal try your "tsc"

Practice Your TypeScript from Console Mode:

1. Create a new TypeScript file

"welcome.ts"

```
console.log("Welcome to TypeScript");
console.log("Hello ! World");
```

2. Compile in Terminal

```
> tsc  welcome.ts
```

3. Run the JavaScript file

```
> node welcome.js
```

TypeScript Language

1. Variables
2. DataTypes
3. Operators
4. Statements
5. Functions

Variables in TypeScript

- Variables are storage locations in memory where you can store a value and use it as a part of any expression.
- Variables have 3 configuration states:
    1. Declaration

        var x;

    2. Rendering / Assignment

        x = 10;

    3. Initialization

        var x = 10;

- JavaScript can handle variables without declaration and initialization when it is not in strict mode.
- If JavaScript is in strict mode then it is mandatory to declare or initialize variable.
- TypeScript is in strict mode by default.
- Hence declaring or initialization of variables in mandatory.
- Variables can be declared or initialized by using following keywords
    1. var
    2. let
    3. const

Var

- Var is used to define a function scope variable.

- You can declare or initialized in any block of a function and access from another block with in the same function.

- Var allows declaration, rendering, initialization.

Syntax:

```
 function f1() {
 var x = 10; → Initialized
 if(x==10) {
     var y;             → Declared
     y=20;              → Assigned / Rendered
 }
 console.log(`x=${x}\n y=${y}`);   //y is valid as it is declared
                                   as function scope with var
}
f1();
```

- Var allows shadowing.

- Shadowing is the process of re-declaring or initializing the same name identifier within the scope.

Syntax:

```
function f1() {
  var x = 10;
  if(x==10) {
      var y=20;
      var y=30;          // y - shadowing
  }
  console.log(`x=${x}\n y=${y}`);
}
f1();
```

- Var allows hoisting.

- Hoisting is the process of using a variable in code before the declaration or initialization.

Syntax:

```
function f1() {
  x = 10;
  console.log(`x=${x}`);
  var x;                    // x is hoisted
}
f1();
```

Let:

- It is used to define a block scope variable.
- If a variable is declared or initialized inside a block it is accessible only within the block.
- let allows declaring, rendering, initialization
- let will not allow shadowing.
- let will not allow hoisting.

Syntax:

```
function f1() {
    x = 20;
    console.log(`x=${x}`);
    let x;        // invalid
  }
  f1();
```

const:

- It is also block scope like let.
- It allows only initialization.
- It will not allow declaration and rendering.
- It will not allow shadowing
- It will not allow hoisting.

TypeScript Data Types

- Primitive Types

  1. number type

     Syntax:

     ```
     let variableName:Datatype = value;
     let float:number = 4.6;
     ```

  2. string type

String Type in TypeScript

- String is literal with group characters [alpha numeric and special chars] enclosed in

  a) Double Quotes

    "string"

  b) Single Quotes

    'string'

  c) Back Ticks

    `string`

- Double and Single quotes are used to swap between outer and inner string vice versa.

```
let msg:string ="<a href='home.html'>Home</a>";
let msg:string ='<a href="home.html">Home</a>';
```

- Back Tick is available from ES5
- Back Tick allows an Embeded Expression inside string.
- Single and Double quotes will not allow embeded expression they use concatination.
- String with embeded expression is defined by using

  "${ }" - it is possible only for backtick

Ex:

```
let username:string = "John"
let age:number = 20;
console.log("Hello !"+ " " + username+" "+"You will be"+ " " + (age+1) + " "
+"Next Year");
```

console.log(`Hello ! ${username} You will be ${age+1} Next Year`);

- In a string literal you can define special chars. But few special characters are not printable. The non printable character usually escape printing.

- You can print any non-printable character by using "\".

```
let path:string = "D:\Pics\flower.jpg";
console.log(path);

 O/P:   D:Picsflower.jpg

let path:string = "D:\\Pics\\flower.jpg";

 O/P:   D:\Pics\flower.jpg
```

Ex:

```
let path:string = "\"D:\\Images\\Cars\\Ferrari.jpg\"";
console.log(`Path=${path}`);
```

String Handling

-TypeScript can use all JavaScript string functions to handle any string dynamically.

-TypeScript can use ES5 and ES6 latest features.

1. charAt()   : It returns the character at specified

index.

```
let msg:string = "Welcome to TypeScript";
msg.charAt(0);      // W
```

2. charCodeAt()   : It returns the Unicode of the character
at specified index. [UTF]

```
let name:string = "Ajay";
name.charCodeAt(0);    // 65
```

Ex:

```
let username:string = "John";
let firstCharCode:number = username.charCodeAt(0);
if(firstCharCode>=65 && firstCharCode<=90){
   console.log(`Hello ! ${username}`);
} else {
   console.error(`Error: Name must start with UpperCase Letter`);
}
```

3. startswith()] These are boolean functions used to verify
4. endswith()]   whether the give string is starting with or
ending with specified char(s) and return

true or false.

Syntax:

```
let name:string = "btnSubmit";
name.startsWith("btn");     // true
```

Ex: startsWith()

```
let className:string = 'bg-primary';
if(className.startsWith("form")) {
   console.log(`You Defined a Form Class`);
} else if(className.startsWith("btn")) {
   console.log(`You Defined a Button Class`);
} else {
   console.log(`You are using Miscelaneous class ${className}`);
}
```

Ex: endsWith()

```
let className:string = 'text-primary';
if(className.endsWith("center")) {
   console.log(`You Defined an Alignment Class`);
} else if(className.endsWith("primary")) {
   console.log(`You Defined Contextual Class`);
} else {
    console.log(`You are using Miscelaneous class ${className}`);
}
```

5. indexOf() : It returns the index number of specified character in a string. It returns the first occurance index. It returns -1 if the given char not found.

6. lastIndexOf() : It is similar to indexOf() but gets the last occurance index number of specified char.

Ex:

```
let email:string = "john@gmail.com";
if(email.indexOf("@")==-1) {
    console.error(`Error: Invalid email [@ Missing]`);
} else {
    console.log(`Your email ${email} Verified..`);
}
```

1. charAt()
2. charCodeAt()
3. startsWith()
4. endsWith()

5. indexOf()

6. lastIndexOf()

7. includes() : It is used to verify whether the given chars are present in string. It returns boolean true/false.

```
        yourString.includes("searchString"); // true/false
```

Ex:

```
let msg:string = "Welcome to JavaScript String Methods.";
if(msg.includes("TypeScript")){
   console.log(`You are using TypeScript`);
} else if(msg.includes("JavaScript")) {
   console.log(`You are using JavaScript`);
}
```

JavaScript have obsolete methods and properties.
 [no-longer in use but still available in library]

8. search() : It can search for specified chars and return the starting index. so that we can know where the given chars occur. It returns -1 if the given char not found.

Syntax:

```
yourString.search("searchString");
```

EX:

```
let msg:string = "Welcome to JavaScript String Methods.";
console.log(msg.search("TypeScript"));
```

9. replace(): It can search for give chars in a string and
replace with new chars and returns a new string.
The new string contains updated values.

Syntax:

```
yourString.replace("oldString", "newString");
```

Note:

- replace() is case sensitive.
- replace() will only change the first occurance
- You have to use "RegularExpression" to replace any global value or case-insensitive.
- Regular Expression is defined in "/ /"
- It uses the meta chars

+	one or more occurance
?	zero or one occurance
*	zero or more occurance
d	numeric
w	alpha numberic

g	global
i	case-insensitive

Ex:

```
let msg:string = "Welcome to javaScript. You are learning javaScript String
Methods.";
console.log(msg);
let newString:string = msg.replace(/javascript/gi, "TypeScript");
console.log(newString);
```

g - global

i - case-insensitive

10. match(): It checks whether the given string is matching
with specified regular expression and returns
boolean true or false.

Syntax:

```
yourString.match(regExp);   // true/false
```

Ex:

```
let mobile:string = "+919876543210";
let regExp:any = /\+91[0-9]{10}/;
if(mobile.match(regExp)) {
   console.log(`Your Mobile ${mobile} Verified`);
} else {
   console.log(`Invalid Mobile:+91 with 10 digits`);
}
```

10. match()

Regular Expression

- Regular expression uses a pattern for validating the format of value.
- Regular expression is built by using
  a) Meta Characters and
  b) Quantifiers

Metacharacter	Description
?	It defines zero or one occurance of char
+	It defined one or more occurances
*	It defines zero or more occurances

```
colour]     colou?r
color  ]
```

Ex:

```
let word:string = "color";
let regExp:any = /colou?r/;
if(word.match(regExp)){
    console.log(`Your Spelling ${word} is Valid`);
} else {
    console.log(`Your Spelling ${word} is incorrect [color or colour] is valid`);
}
```

Metacharacter	Description
\^	Starts with
^	Excluding the given all others allowed
$	Ends with
\	Special chars
d	Only numeric allowed
w	Alpha Numeric with underscore allowed. [uppercase, lowercase, number and underscore]
i	Ignore case
g	global
[ ]	Range and specific set of chars
[A-Z]	Only uppercase allowed
[a-z]	Only lowercase allowed
[a-Z] or [a-zA-Z]	Both upper and lowercase allowed
[0-9]	Only numeric allowed
[a-zA-Z0-9]	Alpha numeric
[a,d,m]	Only specified chars allowed
[^a,d,m]	Excluding specified chars all others allowed.

[a-mA-M4-9]	Only chars in specified range allowed
\+ \- \. \@ \%	Special chars individually.
(?=.*[A-Z])	Atleast one uppercase letter
(?=.*[0-9])	At least one numeric
(?=.*[!@#$%])	At least one special char

[A-Z] only uppercase letters allowed

How Many chars allowed?

by default only 1.

Quantifier	Description
{n}	Exactly n-number of chars
	{4} only 4 allowed
{n,m}	Minimum-n and Maximum-m
	{4,10}
{n, }	Minimum-n and Maximum-any
	{4, }

Ex:

[A-Z]{4}	Exactly 4 uppercase letters only
	JOHN - valid
	JOH - Invalid
	JOHNK- Invalid

john - Invalid

[A-Z]{4,10}   Uppercase between 4 to 10

Ex:

```
// Password 4 to 15 chars alpha numberic with at least 1 uppercase letter
let password:string = "john123G";
let regExp:any = /(?=.*[A-Z])\w{4,15}/;
if(password.match(regExp)) {
   console.log(`${password} - Strong Password`);
} else {
   if(password.length<4) {
      console.log(`${password} - Poor Password`);
   } else {
      console.log(`${password} - Weak Password`);
   }
}
```

11. substring() : It can extract the a portion of string using start and end index. It can read bi-directional.

```
Syntax:   msg.substring(0,7);
          msg.substring(7,0);
```

12. substr() : It is similar to substring() but allows only from

start to end, where end index must be always
a value after the start index.

```
msg.substr(0,7);
msg.substr(7,0);  // not valid
```

split()   : It can split the string at specified char and return
an array of strings.

Syntax:

```
    yourString.split('delimeter');
```

Ex:

```
let msg:string = "SamsungTV,45000.55,InStock";
let data:string[] = msg.split(',');
for(var value of data) {
   console.log(value);
}
console.log(`Name=${data[0]}`);
```

trim(): It returns a new string by removing the leading spaces.

Ex:

```
let password:string = " john ";
```

```
if(password.trim()=="john"){
   console.log('Verified..');
} else {
   console.log('Invalid Password');
}
```

Primitive Types

- number
- string
- boolean

What is the purpose of storing a boolean value in memory? [true/false]

Boolean Type

- boolean is used in decision making.
- The boolean values are used to control the execution flow.
- TypeScript uses "boolean" as data type to store a bool value in memory.
- It can handle on "true or false".
- Like JavaScript it is not 0 or 1 it is directly true or false.

```
JavaScript  1 = true
            0 = false
```

- Traditionally boolean variable name must start with "is".

Ex:

let Name:string = "Samsung TV";

let Price:number = 45000.55;

let isInStock:boolean = true;

console.log(`Name=${Name}\nPrice=${Price}\nStock=${(isInStock==true)?"Available":"Out of Stock"}`);

Undefined Type

undefined : Variable is available but no value defined.

not-defined: There is no reference for the variable you specified.

```
let x:number;          //x is undefined.
console.log(`x=${x}\ny=${y}`); // y is not defined
```

TypeScript supports Union of Types, which allows a variable to handle multiple types.

```
let x:string|number = 10;
```

Ex:

let page:string ="about.html";

let msg:string|number;

if(page=="home.html"){

```
  msg=`You requested for ${page} page`;
} else {
   msg = 404;
}
if(msg==400) {
   msg = "Bad Request";
} else if (msg==404) {
   msg = "Not Found";
}
console.log(msg);
```

Ex: Undefined

```
let Name:string = "Samsung TV";
let Price:number|undefined=50000.34;
if(Price==undefined) {
   console.log(`Name=${Name}`);
} else {
   console.log(`Name=${Name}\nPrice=${Price}`);
}
```

Primitive Data Types

- number
- string
- boolean

- undefined
- null

```
        * compile time
        let x:number;
        console.log("x=" + x);     // x=undefined

        * run time
         let x:number = prompt("Enter Number");
         console.log("x=" + x );    // x = null
```

null is an exception type , which indicates that there is no value provided into reference.

Ex:

1. Create a new File

strdemo.ts

```
let userName:string|null = prompt("Enter Name");
if(userName==null) {
   document.write(`You Canceled..`);
} else if(userName=="") {
   document.write(`Name can't be Empty`);
} else {
   document.write(`Hello ! ${userName}`);
}
```

2. Transcompile

```
> tsc strdemo.ts
```

3. Add a new HTML file "home.html"

```
<head>
 <script src="strdemo.js"> </script>
</head>
```

Primitive Type

- They are Immutable
- Immutables are the object whose state can't be changed once the object is created.
- string, number, boolean, null, undefined are immutable.
- They have a fixed range for values.

Non-Primitive Types

- They are Mutable
- Mutable is a type of variable that can be changed.
- A Mutable object is an object whose state can be modified after it is created.
- Without disturbing the exisiting state it can point to an new memory to handle a different value.
- It can change the memory as per requirement.
- Hence no fixed range for values.
- Value range varies according to memory available.
- TypeScript Non-Primitive Types are

1. Array
2. Object
3. Date
4. Regular Expression

Array

- In Computer Programming Arrays are used to reduce overhead and reduce complexity.

- Arrays will reduce overhead by storing values in sequential order.

- Arrays will reduce complexity by storing multiple value under one name.

- Arrays can different types of values.

- Array size can be changed dynamically.

- Array can have the behavior of stack, queue, dictionary type collection.

Declaring Array

- You can use var, let or const.

- Array uses the data type with meta character [ ], which specifies a range of values.

```
let  arrayName:string[];
```

- TypeScript can configure Arrays of same type or various types. You can use "any" as the type to handle various values.

```
let arrayName:any[];
```

- If array is designed to initialize various types of values then it is often known as "Tuple".

- You have to initialize or assign memory for array.

Syntax:

```
let values:number[];
values[0]=10;
values[1]=20;
console.log(values[0]);    // Type Error: No property - 0
```

- You can initialize or assign memory for array by using meta character "[ ]" or Array constructor "Array()".

Initializing:

```
let values:string[] = [ ];
let values:string[] = new Array();
```

Assigning:

```
let values:string[];  → declaring
values = [ ]; → Assigning
values = new Array()→ Assigning
```

Ex:

```
let values:number[] = [];
values["0"]=30;
values["1"]=50;
console.log(values["1"]);
```

FAQ: What is difference between [ ] and Array()?

A: Meta character [] will allow to configure a Tuple if data type is "any" type.

Array() constructor will not allow various types of values even when the type is defined as "any" type during initialization. It will allow to assign.

Ex: Valid

```
let values:any[] = new Array();
values[0]= 10;
values[1]= "A";
values[2]= true;
console.log(`${values[0]}\n${values[1]}\n${values[2]}`);
```

Ex: Invalid

```
let values:any[] = new Array(10, "A", true);
console.log(`${values[0]}\n${values[1]}\n${values[2]}`);
```

- Array can handle any type of data in memory , which can be primitive type, non-primitive type, or a function.

- Technically array is an object.
- Object comprises of data and logic.

Ex:

```
let newArray:any[] = [0,"",true,[],function(){}];
newArray[0] = 10;
newArray[1] = "John";
newArray[2] = true;
newArray[3] = ["A", "B"];
newArray[4] = function(){
    console.log("Function in Array");
};
for(var item of newArray[3]) {
    console.log(item);
}
newArray[4]();
```

Array Manipulation

- TypeScript can use all JavaScript Array functions to manipulate array.

Read Element from Array

Function	**Description**
toString()	It returns all array elements separated with ",".
join()	It returns all array elements by using a custom delimiter. [separator].
filter()	It uses a condition and returns all elements that are matching with condition.
find()	It is similar to filter but returns only the first occurance value that matches the condition.
slice()	It can extract elements based on index number.

Example:

```
let products:string[] = ["TV","Mobile","Shoe"];
console.log(`To String: ${products.toString()}`);
console.log(`Join : ${products.join("-->")}`);
console.log(`Slice : ${products.slice(1,2)}`);
```

Syntax:

```
        arrayName.find(function(val){
            return val|condition;
            })
           arrayName.filter(function(val){
            return  val|condition;
            })
```

Ex: filter()

```
let sales:number[] = [2000, 5000, 12000, 56000, 2100];
console.log(sales.filter(function(sale){return sale>=10000}).toString());
```

Ex: find()

```
let sales:number[] = [2000, 5000, 12000, 56000, 2100];
console.log(sales.find(function(sale){return sale>=10000}).toString());
```

- You can use loops and Iterations

  * Loop required condition, initialization and counter

```
        for(var i=0; i<length; i++)
        {
        }
```

  * Iterator is a software design pattern used to read elements from a collection in sequential order. It doesn't require explicit initializer, condition or counter.

        for..in   → Iterates through properties

        for..of   → Iterates through values

Ex:

```
let categories:string[] = ["Electronics","Fashion","Footwear"];
```

```
//loop
console.log(`-----Loop------`);
for(var i=0;i<categories.length; i++) {
    console.log(categories[i]);
}

//Iterate the properties
console.log(`-----Iterate Properties----`);
for(var property in categories) {
    console.log(property);
}

//Iterate the values
console.log(`----Iterate Values-----`);
for(var value of categories) {
    console.log(value);
}

//Iterate over properties and values

for(var property in categories) {
    console.log(`${property} : ${categories[property]}`);
}
```

Add Elements into Array

push()	Add new Element(s) as last item(s), towards end.
unshift()	Add new Element(s) as first item(s).
splice()	Add new Element(s) at specific position.

Syntax:

```
  arrayName.splice(startIndex, deleteCount, "NewItems,...")
```

Ex:

```
let categories:string[] = ["Electronics","Fashion"];
function PrintList(){
    for(var property in categories) {
        console.log(`${property}:${categories[property]}`);
    }
}
PrintList();
categories.splice(1,0,"Mobiles","Footwear");
console.log(`---New Categories Added----`);
PrintList();
```

Array Manipulation

- TypeScript can use all JavaScript Array functions to manipulate array.

Read Element from Array

Function	Description
toString()	It returns all array elements separated with ",".
join()	It returns all array elements by using a custom delimeter. [separator].
filter()	It uses a condition and returns all elements that are matching with condition.
find()	It is similar to filter but returns only the first occurance value that matches the condition.
slice()	It can extract elements based on index number.

Ex:

```
let products:string[] = ["TV","Mobile","Shoe"];
console.log(`To String: ${products.toString()}`);
console.log(`Join : ${products.join("-->")}`);
console.log(`Slice : ${products.slice(1,2)}`);
```

Syntax:

```
          arrayName.find(function(val){
           return val|condition;
           })
          arrayName.filter(function(val){
           return  val|condition;
           })
```

Ex: filter()

```
let sales:number[] = [2000, 5000, 12000, 56000, 2100];
console.log(sales.filter(function(sale){return sale>=10000}).toString());
```

Ex: find()

```
let sales:number[] = [2000, 5000, 12000, 56000, 2100];
console.log(sales.find(function(sale){return sale>=10000}).toString());
```

- You can use loops and Iterations
  * Loop required condition, initialization and counter

```
        for(var i=0; i<length; i++)
        {
        }
```

  * Iterator is a software design pattern used to read elements from a collection in sequential order. It doesn't require explicit initializer, condition or counter.

  for..in ? Iterates through properties

for..of ? Iterates through values

Ex:

```
let categories:string[] = ["Electronics","Fashion","Footwear"];

//loop
console.log(`-----Loop------`);
for(var i=0;i<categories.length; i++) {
   console.log(categories[i]);
}

//Iterate the properties
console.log(`-----Iterate Properties----`);
for(var property in categories) {
   console.log(property);
}

//Iterate the values
console.log(`----Iterate Values-----`);
for(var value of categories) {
   console.log(value);
}

//Iterate over properties and values
```

```
for(var property in categories) {
    console.log(`${property} : ${categories[property]}`);
}
```

Add Elements into Array

push()	Add new Element(s) as last item(s), towards end.
unshift()	Add new Element(s) as first item(s).
splice()	Add new Element(s) at specific position.

Syntax:

```
arrayName.splice(startIndex, deleteCount, "NewItems,...")
```

Ex:

```
let categories:string[] = ["Electronics","Fashion"];
function PrintList(){
    for(var property in categories) {
        console.log(`${property}:${categories[property]}`);
    }
}
PrintList();
categories.splice(1,0,"Mobiles","Footwear");
console.log(`---New Categories Added----`);
PrintList();
```

- Reading Elements from Array
- Adding Elements into Array
- Remove Elements from Array

Functions	Description
pop()	It removes and returns the last element
shift()	It removes and returns First element
splice()	It removes elements from specific index and returns an array of removed items.

Ex: pop()

```
let categories:string[] = ["Electronics","Fashion","Footwear"];
function PrintList(){
    for(var property in categories) {
        console.log(`${property}:${categories[property]}`);
    }
}
PrintList();
let removedItem:string = categories.pop();
console.log(`${removedItem} Removed From Collection`);
PrintList();
```

Ex: shift()

```
let categories:string[] = ["Electronics","Fashion","Footwear"];
function PrintList(){
   for(var property in categories) {
      console.log(`${property}:${categories[property]}`);
   }
}
PrintList();
let removedItem:string = categories.shift();
console.log(`${removedItem} Removed From Collection`);
PrintList();
```

Ex: splice(startIndex, deleteCount)

```
let categories:string[] = ["Electronics","Fashion","Footwear"];

function PrintList(){
   for(var property in categories) {
      console.log(`${property}:${categories[property]}`);
   }
}
PrintList();
let removedItem:string[] = categories.splice(1,1);
```

```
console.log(`${removedItem[0]} Removed From Collection`);
PrintList();
```

Sorting Array:

Function	Description
sort()	It arranges the elements in ascending order.
reverse()	It arranges the elements in reverse order. [right to left (or) bottom to top]

Ex:

```
let values:string[] = ["A","D","C","B"];
function PrintList(){
    for(var property in values) {
        console.log(`${property}:${values[property]}`);
    }
}
PrintList();
values.sort();
values.reverse();
console.log(`------Sorted List-----`);
PrintList();
```

Search for any Value in Array:

Function	Description
indexOf()	It can search for value in array and return its index. If value not found then it returns -1
lastIndex()	It can search for values in array and return the last occurance index.
find()	
filter()	

Array Destruction

- It is a technique used to desctruct the structure of array and retrive the elements from array in order to store in individual memory references.

Ex:

```
let values:number[] = [10, 20, 30];
let [a,b,c] = values;
console.log(`A=${a}\nB=${b}\nC=${c}`);
```

Ex:

```
let Factory:any[] = [function(a,b){return a+b},function(a,b){return a*b}];
let [Add, Mul] = Factory;
console.log(`Addition=${Add(10,20)}\nMultiply=${Mul(5,4)}`);
```

Object Type

- Object in computer programming comprises of data and logic.

- Alan Kay introduced the concept of object into computer programming in the early 1967. [Small Talk]

- Keep all related type of data and logic together.

- Data is stored in properties.

- Logic is defined in functions.

- TypeScript object uses Type Inference

[Type Inference is a technique used in TypeScript language, where the data of any property is determined according to the value defined. It will not allow explicit Types]

- The members of any object are accessible with in the object by using "this" keyword. And outside object by using "Object Name".

Ex:

```
let product:object = {
    Name:"",
    Price:0,
    InStock:true,
    Qty:0,
    Total:function(){
        return this.Qty * this.Price;
    },
    Print:function(){

console.log(`Name=${this.Name}\nPrice=${this.Price}\nStock=${(this.InStock==
true)?"Available":"Out of Stock"}\nQty=${this.Qty}\nTotal=${this.Total()}`);
    }
}
product.Name = "Samsung TV";
product.Price = 34000.55;
product.Qty = 2;
product.InStock = true;
product.Print();
console.log(`-----------------`);
product.Name = "Nike Casuals";
product.Price = 3000.55;
product.Qty = 3;
product.InStock = false;
product.Print();
```

Object Format with Key and Value is JSON

Array of Objects:

-It is a collection objects.

-It is mostly used to represent our data.

-It is format of data which often resembles JSON.

[JavaScript Object Notation]

- There is no specific datatype of JSON type data collection.

- We have to use "any" as type.

Ex:

```
let data:any[] = [
   {Name: "TV", Price: 30000.44, Category: "Electronics"},
   {Name: "Shirt", Price:3100.33, Category: "Fashion"},
   {Name: "Mobile", Price:12000.33, Category: "Electronics"},
   {Name: "Nike Casuals", Price: 5000.55, Category: "Footwear"}
];
let searchResults:any[] = data.filter(function(obj){
   return obj.Category=="Electronics" || obj.Category=="Fashion";
})
for(var obj of searchResults) {
   console.log(`${obj.Name} - ${obj.Price}`);
}
```

Ex: LAMBDA

```
let data:any[] = [
    {Name: "TV", Price: 30000.44, Category: "Electronics"},
    {Name: "Shirt", Price:3100.33, Category: "Fashion"},
    {Name: "Mobile", Price:12000.33, Category: "Electronics"},
    {Name: "Nike Casuals", Price: 5000.55, Category: "Footwear"}
];
let searchResults:any[] = data.filter(obj=>obj.Category=="Electronics");

for(var obj of searchResults) {
    console.log(`${obj.Name} - ${obj.Price}`);
}
```

Ex:

```
let data:any[] = [
    {Name: "TV", Price: 30000.44, Category: "Electronics", ShippedTo:
["Delhi","Hyd"]},
    {Name: "Shirt", Price:3100.33, Category: "Fashion", ShippedTo:
["Hyd","Mumbai"]},
    {Name: "Mobile", Price:12000.33, Category: "Electronics",
ShippedTo:["Chennai","Hyd"]},
    {Name: "Nike Casuals", Price: 5000.55, Category: "Footwear", ShippedTo:
["Hyd","Mumbai"]}
];
let searchResults:any[] = data.filter(obj=>obj.Category=="Electronics");

for(var obj of searchResults) {
   console.log(`${obj.Name} - ${obj.Price} - ${obj.ShippedTo.toString()}`);
}
```

Ex:

```
let data:any[] = [
   [{Name:"TV", Price:30045.55}],
   [{Name:"Shirt", Price:5600.55}]
];
console.log(data[1][0].Price);
```

Syntax: Union of Types for Arrays

```
let data:string[]|number[] = [];
data = ["A", "B", "C"];
data = [10, 20, 30];
console.log(data[0]);
```

Regular Expression Type

- It allocates memory to store a regular expression, which is a pattern enclosed in "/ /".
- Regular Expression uses a pattern to verify the format of input value.
- Patterns are built by using Meta Characters and Quantifiers

- TypeScript doesn't provide any reserved datatype for Regular Expression. you have to use "any" type.

```
                    = [ ]  → Array
                    = { } → Object
                    = / /   → Regular Expression
  let  regExp:any = /pattern/;
```

- You value is verified with regular expression by using "match()" function. It is a boolean function that returns true or false.

Meta Characters	Description
?	Zero or One occurance of char
+	One or More occurances of char
*	Zero or many occurances of char

Quantifier

{n}	Exactly n number of chars
{n, m}	Minimum n and max m
{n, }	Minimum n and max any

Ex:

```
let regExp:any = /[A-Z]{4,10}/;
let username:string = "JOHN";
if(username.match(regExp)){
    console.log(`Hello ! ${username}`);
} else {
    console.log(`Name 4 to 10 Uppercase Only`);
}
```

Date Type

- JavaScript and TypeScript library doesn't provide any specific reserved type for Date.

- "any" type is used for date values.
- To store a date value in memory, you have initialize memory by using "Date()" [constructor of Date object]

```
    let  manufactured:any = new Date();  → loads current date
```

- Initialize date value into memory

```
        = new Date("YY-MM-DD");
```

- You can read date value from memory by using various date functions

Function	Description
getHours()	Returns the hour number 0-23
getMinutes()	Returns minutes number 0-59
getSeconds()	Seconds number 0-59
getMilliseconds()	Milliseconds number 0-999
getDay()	returns weekday number 0-6 0-Sunday
getMonth()	returns the month number 0-11 0-January
getDate()	returns the date of day [1 - 31]
getYear()	Obsolete - It returns as per Y2K 1999 - 99 2000 - 100
getFullYear()	It returns full year number

toDateString()
toLocaleDateString()
toTimeString()
toLocaleTimeString()
	pre defined formats for printing date and time

EX:

```
let Mfd:any = new Date("2020-03-22");

let weekdays:string[] = ["Sunday","Mon","Tue","Wed","Thu","Fri","Sat"];
let months:string[] = ["January","Feb","March","April","May","June"];
```

console.log(`Manufactured Month=${months[Mfd.getMonth()]}\nWeekDay=${weekdays[Mfd.getDay()]}\nYear=${Mfd.getFullYear()}`);
console.log(`Manufactured=${Mfd.toDateString()}`);

You set date values dynamically:

Function	Description
setDate()	
setFullYear()	
setHours()	
setMinutes()	
setMilliseconds()	
setMonth()	
setTime()	
setSeconds()	

Ex:

```
let Mfd:any = new Date("2020-03-22");

let weekdays:string[] = ["Sunday","Mon","Tue","Wed","Thu","Fri","Sat"];
let months:string[] = ["January","Feb","March","April","May","June"];
Mfd.setFullYear(2021);
Mfd.setMonth(5);
console.log(`Manufactured Month=${months[Mfd.getMonth()]}\nWeekDay=${weekdays[Mfd.getDay()]}\nYear=${Mfd.getFullYear()}`);
console.log(`Manufactured=${Mfd.toDateString()}`);
```

TypeScript Operators
- In computer programming Operator is an object
- Object comprises of data and logic
- Data is stored in operands
- Functionality is handled by using a Symbol or Keyword

Operand1 [symbol/keyword] Operand2

```
x          +          y
userName   instanceof UserClass
```

- TypeScript can use all JavaScript operators [ES6]

- Based on how many operands that an operator can handle they are classified into following types:

1. **Unary Operator**
   - Single operand
   ex: ++, --
2. **Binary Operator**
   - Two operands
   ex: +, * , /
3. **Ternary Operator**
   - Three operands
   ex: ? :

- Based on the type of value an operator returns they are classified into following:

1. Arithematic Operators
2. Assignment Operators
3. Comparison Operators
4. Logical Operators
5. Bitwise Operators
6. Special Operators

Arithematic Operators:

Operator	**Description**
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (Division Remainder)
**	Exponentation [Math.pow()]
++	Increment
--	Decrement

Example:

```
let x:number = 2;
let y:number = 3;
console.log(`Exponentation : ${x**y}`);     // Math.pow(2,3) =8
```

```
JavaScript:
        true + true =   ?  = 2
        10  + true   =   ?  = 11
TypeScript:
        true + true = ?  //invalid
        10 + true  =   ? // invalid
        true + "A" =  ?  // trueA
```

Increment and Decrement:
======================

```
Post:
        x = 10;
        y = x++;              X=11   ,  Y=10
        y = x--;              X=9,      Y=10

Pre:
        x=10
        y=++x;                x=11, y=11
        y=--x;                x=9    y=9
```

Assignment Operators:

Operator	Description
=	Equal
+=	Add and Assign
-=	Subtract and Assign
*=	Multiply and Assign
/=	Divide and assign
%=	Modulus and Assign
**=	Exponent and Assign

```
x=10;
y=20;
        x+=y;          x = x + y;
                       x = 10+20;  30
                       y = 20
```

Comparision Operators

Operator	**Description**
==	Equal
===	Identical Equal
!=	Not Equal
!==	Not Indentical
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal

JavaScript:

```
        x = "10";
        y = 10;
        x == y   ?    true    [Type Inference]
        x === y ?    false [ both are not same type]

        x="10A";
        y=10;
        x == y  ? false
```

=== can compare only values of same type.
==   can compare values of different types.

Note: TypeScript is strongly typed, it will not allows to compare values of different types even with "==".

Logical Operators

&&	Logical AND
||	Logical OR
!	Logical NOT

(condition) &&(condition) = true - when both conditions true
(condition) || (condition) = true - when any one condition true

x = true;
y = !x;   y=false, x=true

Bitwise Operators

&	AND
|	OR
~	NOT
^	XOR
<<	Zero fill left shift
>>	Signed right shift
>>>	Zero fill right shift

Special Operators

1. Ternary Operator: [? :]

(condition)?statement_if_true:statement_if_false

Ex:
```
let inStock:boolean = false;
console.log((inStock==true)?"Available":"Out of Stock");
```

2. typeof : It is a special operator used to know the data type of value stored in any specific memory reference.

Syntax:
```
        typeof  refName;    // number, string
```

Ex:

```
let product:any = {
    Name: "Samsung TV",
    Price: 45000.55,
    InStock:true,
    ShippedTo: ["Delhi","Hyd"],
    Total: function(){},
    Mfd: new Date()
}
console.log(`Name is ${typeof product.Name}\n Price is ${typeof product.Price}\n
InStock is ${typeof product.InStock}\n ShippedTo is ${typeof
product.ShippedTo}\nTotal is ${typeof product.Total}\n Mfd is ${typeof
product.Mfd} `);
```

O/P:

```
 Name is string
 Price is number
 InStock is boolean
 ShippedTo is object
 Total is function
 Mfd is object
```

3. instanceof operator : It is a boolean operator which verifies whether the object is derived from specific class.

Syntax:

```
        objectName   instanceof   className;  // true-false
```

Ex:

```
class Employee {

}
let emp = new Employee();
let product = new Array();
console.log(`emp is instance of Array ? ${emp instanceof Array}\nemp instance of
Employee ? ${emp instanceof Employee}\n Product instance of Array ? ${product
instanceof Array}\nProduct instance of Object? ${product instanceof Object}`);
```

O/P:

```
emp is instance of Array ? false
emp instance of Employee ? true
Product instance of Array ? true
Product instance of Object? true
```

4. delete operator : It is a special operator used to remove any property from an object. You can't remove readonly properties.

Syntax:

```
        delete  objectName.Property;
        delete  Math.PI;     // invalid
```

Ex:

```
let product:any = {
   Name: "TV",
   Price: 34000.44
};
delete product.Price;
if(product.Price==undefined) {
   console.log(`Name=${product.Name}`);
} else {
console.log(`Name=${product.Name}\nPrice=${product.Price}`);
}
```

5. in operator : It is a special operator used to verify whether the given property is a member of specified object. It returns boolean true or false.

Syntax:

```
        "PropertyName"  in  objectName
```

Ex:

```
let product:any = {
   Name: "TV",
   Price: 34000.44
};
//delete product.Price;
if("Price" in product) {
   console.log(`Name=${product.Name}\nPrice=${product.Price}`);
} else {
   console.log(`Name=${product.Name}`);
}
```

6. of operator : It is a special operator used to access the values from a collection using any iterator.

Syntax:

```
        for(var item of collection) { }
```

Ex:

```
let products:string[] = ["TV","Mobile","Shoe"];
let data:any[] = [
   {Name: "TV", Price: 45000.55},
   {Name: "Mobile", Prie: 30000.44}
];
for(var product of products) {
   console.log(product);
}
for(var item of data) {
   console.log(item);
}
```

7. void : It is a special operator used to designate any functionality that will not return a value for that specific state.

Ex:

```
<a
href="javascript:void(window.document.body.style.backgroundColor='yellow')">
Home</a>
```

Statements in TypeScript

In Computer programming statements are used to control the execution flow.

Statements	Keywords
Selection	if, else, switch, case, default
Iteration & Looping	for, while, do while, for..in, for..of
Jump	break, continue, return
Exception Handling	try, catch, throw, finally

OOP in TypeScript

In real world development we have 3 types of programming systems
1. POPS [Process Oriented Programming System]
2. OBPS [Object Based Programming System]
3. OOPS [Object Oriented Programming System]

POPS
- Process Oriented Programming System
- It supports low level features.
- It can directly interact with hardware services
- It is faster in communication.
- It requires very less memory.
    Ex: C, PASCAL, COBOL
- Code reusability concerns
- Code separation concerns
- Dynamic Memory allocation concerns

OBPS
- Object Based Programming System
- It supports code re-usability
- It supports code separation
- It supports dynamic memory allocations
    Ex: JavaScript [before ES5], VB
- Extensibility concerns
- Code Security concerns
- Will not support dynamic polymorphism.

OOPS
- Object Oriented Programming System
- Object design by "Alan Kay"
- 1967 by "Johan Olay, Kristian Nygaard" - On SIMULA 67
    Code Reusability
- 1970's "Trygve" introduces MVC to address the issues related to code - separation. It was formulated with "Small Talk"

Java	Spring
PHP	Cake PHP, Code Igniter
Perl	Catalyst, Dancer
Python	Django, Flask , Grok

Ruby	Ruby on Rails
JavaScript	SPINE, Angular JS, Angular
.NET	ASP.NET MVC

Features of OOP:	Characterstics
- Code reusability	- Inheritence
- Code Separation	- Abstraction
- Code Extensibility	- Encapsualtion
- Mantainability	- Polymorphism
- Testability	
- Loosely Coupled	
- Code Security	

DrawBacks of OOP:
- Uses more memory
- Slow
- Can't directly interact with hardware services

Contracts in OOP
- In OOP every component is designed as per the specified contract.
- Contract defines set of rules for designing a component.
- Technically contracts are defined as "interfaces"
- Interface defines set of rules for designing components in OOP.
- The keyword "interface" is used to create an Interface.

Syntax:
```
interface  InterfaceName
{
  // rules
}
```

- Every rule in a contract must contain only declaration. It should not have any initialization, rendering or definition.

- The properties and methods defined in a contract must contain only declaration.

```
interface IProduct {
  Name:string = "TV";    invalid
  Total():number { };          invalid
```

```
        }

        interface IProduct {
          Name:string;         valid
          Total():number;           valid
        }
```

- The contract member can't have any access restriction for modification. You can't use access modifier for contract member. Type Script access modifiers are:
    - public
    - private
    - protected
- The contract member can have access specifier : readonly
- Every Member defined in a contract is mandatory by default. i.e Every rule defined in contract must have an implementation.

Ex:

```
interface IProduct
{
  Name:string;
  Price:number;
  Qty:number;
  Total():number;
}
let product:IProduct = {
   Name:"TV",
   Price:34000.44,
   Qty:3,
   Total: function(){
      return this.Qty*this.Price;
   }
}
console.log(`Name=${product.Name}\nPrice=${product.Price}\nQty=${product.Qty}\nTotal=${product.Total()}`)
```

Optional Rules in Contract
======================

- A Contract can contain optional rules.
- Rules can be optional because your component can have goals.
- Every component will have objective and goal.
- Objective must be achieved and time bound.

- Objective is non-nullable.
- Every member in contract is by default non-nullable.
- Goal is not mandatory to achieve. It is optional.
- Contract can be defined with optional rules by using a Null reference character "?" .
- "?" is used to define a non-nullable rule into nullable.

Syntax:

```
        interface InterfaceName {
            property?:type;
            method?():type;
        }
```

Ex: Optional member

```
interface IProduct
{
  Name:string;
  Price:number;
  Qty:number;
  Vendor?:string;
  Total():number;
}
let tv:IProduct = {
   Name:"TV",
   Price:34000.44,
   Qty:3,
   Total: function(){
      return this.Qty*this.Price;
   }
}
let mobile:IProduct = {
   Name:"Mobile",
   Price:12000.44,
   Qty:2,
   Vendor: "Samsung",
   Total: function(){
      return this.Qty*this.Price;
   }
}
```

```
function Print(obj) {
   console.log(`Name=${obj.Name}\nPrice=${obj.Price}\nQty=${obj.Qty}\nTotal
=${obj.Total()}\nVendor=${(obj.Vendor==undefined)?"Anonymous
Vendor":obj.Vendor}`);
}
Print(tv);
Print(mobile);
```

- Every contract member can be assigned with a value after initialization.

Syntax:

```
        tv.Name = "LG TV";
        Print(tv);
```

"Readonly Members in Contract"
==============================

- A readonly member will not allow to re-define a value or functionality after initialization.
- "readonly" is an access specifier

Ex:

```
interface IProduct
{
  readonly Name:string;
  Price:number;
  Qty:number;
  Vendor?:string;
  Total():number;
}
let tv:IProduct = {
   Name:"TV",
   Price:34000.44,
   Qty:3,
   Total: function(){
      return this.Qty*this.Price;
   }
}
let mobile:IProduct = {
   Name:"Mobile",
   Price:12000.44,
    Qty:2,
    Vendor: "Samsung",
    Total: function(){
        return this.Qty*this.Price;
    }
}
function Print(obj) {
    console.log(`Name=${obj.Name}\nPrice=${obj.Price}\nQty=${obj.Qty}\nTotal
=${obj.Total()}\nVendor=${(obj.Vendor==undefined)?"Anonymous
Vendor":obj.Vendor}`);
}
tv.Name = "LG TV";       // invalid - Name is readonly
Print(tv);
Print(mobile);
```

-Contracts
-Readonly Properties
-Optional Properties

Extending Contracts

- A contract can be extended with rules.
- Its intention is to extended the contract without disturbing the existing contracts.
- "extends" is a keyword that configures relation between contracts.
- Existing contract is known as Super contract.
- Newly created contract is known as Derived contract.
- You can implement Derived contract to access the rules from both Super and Derived.

Syntax:

```
interface      Super {
     // super rules
}
interface  Derived extends Super {
    // derived rules
}
let  obj : Derived = {
    //rules of both super and derived
}
```

Ex:
```
interface IProduct {
   Name:string;
   Price:number;
}
interface ICategory extends IProduct {
   CategoryName:string;
}
let tv:ICategory = {
   Name:"TV",
   Price:45500.55,
   CategoryName:"Electronics"
};
```

- A Contract can extend multiple contracts by using
  a) Multi Level Hierarchy
  b) Multiple Hierarchy

-Multi Level Hierachy uses incremental model i.e extending the dreived contract and add new rules to contract.

Ex:
```
interface IModel {
   ModelName:string;
}
interface IMemory extends IModel {
   RAMsize:string;
}
interface ICamera extends IMemory {
   Mpx:string;
}
interface IMobile extends ICamera {
   Print():void;
}
let mob:IMobile = {
   ModelName: "Samsung",
   RAMsize: "8GB",
   Mpx:"20px",
   Print:function (){
        console.log(`Name=${this.ModelName}\nRAM=${this.RAMsize}\nCamera=
${this.Mpx}`);
    }
}
mob.Print();
```

- Multiple Hierarchy allow a single contract to extend all existing contracts.

Ex:

```
interface IModel {
    ModelName:string;
}
interface IMemory {
    RAMsize:string;
}
interface ICamera {
    Mpx:string;
}
interface IMobile extends IModel, IMemory, ICamera {
    Print():void;
}
let mob:IMobile = {
    ModelName: "LG",
    RAMsize: "8GB",
    Mpx:"20px",
    Print:function (){
        console.log(`Name=${this.ModelName}\nRAM=${this.RAMsize}\nCamera=
${this.Mpx}`);
    }
}
mob.Print();
```

Templates in OOP

Projects
1. Implementation
2. Support
3. Rollout

4. Upgrade

Class in OOP
- In computer programming class is a program template.
- It provides a set of properties and methods to store data and handle manipulation.
- Class is used as a Model to represent the data we are working with.
- Class is used as an Logical Enity to represent business.
- Class is used as a Blue print for creating similar type of objects.
- Class comprises of a set of members which includes
  - a) Properties
  - b) Methods
  - c) Accessors
  - d) Constructor
- Class is designed by using the keyword class.
- A class can be configured in 2 ways
  - a) Class Declaration
  - b) Class Expression

Class Declaration:
- It comprises of class with a name.
- Members are encapsualted in { }.
- You can load the class into memory by using class name.

Syntax:
```
class  className {
    // members
}
```

Class Expression:
- It comprises of class without a name.
- It is loaded into memory when required.
- It contains a reference name, which you can use to access its members.

Syntax:
```
let  emp = class {
    //members
}
```

Ex:
```
//Class Declaration
class Product {

}
//Class Expression
let Employee = class {

}
```

Static and Non-Static Members

- A class comprises of static and non-static members.

Static
- uses continous memory.
- memory allocated for first object will continue for other objects.
- It occupies more memory.
- Security issues
- But good for continous operations.

Non-Static
- uses a discrete memory
- memory is newly allocated for every object.
- It disposes the memory after using.
- Can't handle continous operations
- But uses less memory.

TypeScript static members in a class are defined by using "static" keyword. And they are accessed by using class name.

TypeScript non-static members are the members without static marking. And they are accessed with in the class by using "this" keyword and outside the class by using an instance of class.

Ex:
```
class Demo {
   static s=0;
   n=0;
   constructor(){
        Demo.s = Demo.s + 1;
        this.n = this.n + 1;
    }
    Print(){
        console.log(`s=${Demo.s}  n=${this.n}`);
    }
}
let obj1 = new Demo;
obj1.Print();
let obj2 = new Demo;
obj2.Print();
let obj3 = new Demo;
obj3.Print();
```

Declare Class
Types of Members in Class
Static and Non-Static Members

Access Modifiers in TypeScript

- Access modifiers defines the restriction about how a member is accessed from various locations.
- TypeScript supports 3 access modfiers for members
  1. public
  2. private
  3. protected

public : It defines accessbility for a member in class. so that it is accessible with in the class and by using an object of class from any location.

private : It defines accessbility for a member in class. It is not accessible through any medium out side the class.

protected: It defines accessbility for a member in class. It can be access outside the super class and within the derived class only by using a derived class object.

Syntax:

```
class SuperClass {
   public Name:string;
   private Price:number;
   protected Quantity:number;
}
class DerivedClass extends SuperClass {
    Print(obj1:SuperClass, obj2:DerivedClass):void{
        obj1.Name;  // super class public member
        obj2.Name;  // super class public member
        obj2.Quantity; //supper class protected
    }
}
let object1 = new Derived();
object1.Name;    // public
object1.Price;    // invalid - private
object1.Quantity; // invalid - protected
```

Accessors in TypeScript

- TypeScript supports getters/setters as a way of intercepting access to a member of classs.
- Accessors allow to define authorized access to any member.
- Accessors give a fine-grained control over member and its accessibility.
- You can restrict getting and setting of values according to the state of application.

Syntax:

```
          get PropertyName() {
            return PropertyValue;
          }
          set PropertyName(newValue) {
              oldValue = newValue;
          }
```

Ex:

```
let password:string = "admin123";
class Product {
    private _productName:string;

    get ProductName():string {
        return this._productName;
    }
    set ProductName(newName:string) {
        if(password && password=="admin123") {
            this._productName = newName;
        } else {
            console.log(`UnAuthorized : You are not authorized to set Product Name`);
        }
    }
}
let tv = new Product();
tv.ProductName = "Samsung TV";
if(tv.ProductName==undefined) {
    console.log(`Name is Readonly for this User`);
} else {
console.log(`Name = ${tv.ProductName}`);
}
```

- Constructor
- Properties
- Methods
- Accessors

Class
Class Declarations
Class Expression
Static and Non-Static Members
Access Modifiers
Accessors

Class Members

- A typical typescript class comprises of
    a) Constructor
    b) Properties
    c) Methods
    d) Accessors

Constructor
- Constructor is a special type of method.

- Constructor method loads and executes automatically when memory is allocated for class.
- Normal methods require an explicit call.
- Constructor initializes memory in a class where it can load functionality that executes implicitly.
- TypeScript constructor is designed by using "constructor" keyword.
- It is anonymous method i.e no name for constructor required explicitly.
- Class name is implicitly used for constructor.

Syntax:
```
    class Demo
    {
        constructor() {
         }
    }
   let obj = new Demo;     // Parameterless constructor is
                              called.
   let obj = new Demo();   // Args for constructor can be
                              passed.
```
Ex:
```
class Database
{
    constructor(){
        console.log(`Connected to Database`);
    }
    public Insert(){
        console.log(`Record Inserted..`);
    }
}
let oracle = new Database;
oracle.Insert();
```

- Constructor can be parameterized
- The args are passed into parameterized constructor at the time of allocating memory for class.

Ex:
```
class Database
{
    constructor(dbName?:string){
        if(dbName==undefined){
            console.log(`Connected to Unknown Database..`);
        } else {
        console.log(`Connected to ${dbName} Database`);
        }
    }
    public Insert(){
        console.log(`Record Inserted..`);
    }
}
let oracle = new Database("Oracle");
oracle.Insert();
```

- By default constructor is "public" in access.

1. What is Constructor?
2. Purpose of Constructor in Class?
3. Can we use multiple constructors?
- Multiple constructor implementations are not allowed
4. Can we define access modifiers for constructor?
- Yes
5. Can interface have a constructor?
A. No
6. Why interface can't have a constructor?
A. Constructor requires implementation of functionality. And Interface will not allow.
7. Why multiple inheritence is not supported for classes?
A. Constructor Deadlock
8. Can a constructor have parameters?
A. Yes
9. Can it have multiple parameters?
A. Yes

Properties:
- Properties are used to store data.

FAQ:
1. What is difference between Variable and Property?

```
        let  Name:string = "TV";  Variable
```

public Name:string = "TV"; Property

A.

Variables

-Variables are Immutable.
-Their state can't be changed after created.
- Less Secured
- Lack of code security

Properties

-Properties are Mutable.
-Their state can be changed after created.
-More Secured
-Have code security
-More grained control by using accessors.

```
 <img src="">
   tag  src          ? attribute
var pic = new Image();
 pic.src = ""       ? property
```

- Class will not allow variables.
- Class can contain only properties.
- Properties can be defined with accessors.

FAQ: What is the purpose of accessors?
A. Provides authorized access to properties.

- Properties can be defined with access modifiers like
public, private, protected

- Property can be configured as "readonly".

Syntax:

```
class Demo {
  public  Name:string = "TV";
}
```

- Property can be static or non-static.
- Static property is accessible with in the class or outside class by using class name.

- Non-static property is accessbile with in the class by using "this" keyword and outside the class by using "instance" of class.

Methods in Class

- Method defines the actions to be performed.
- Logic is defined in Method.
- The behavior of object is controlled by using methods.

FAQ: What is difference between a "function, method & procedure"?
A.

function	- Always intended to return a value.
method	- Not intended to return a value.
procedure	- May or may not return value.

- functions are defined in TypeScript by using "function" keyword or "Function" constructor.

```
        function  f1() {
        }
        let f2 = new Function("a","b","return a+b");
```

Syntax:

```
function f1() {
   console.log(`Function f1`);
}
let f2 = new Function("a","b","return a+b");
f1();
console.log(f2(10,20));
```

FAQ: what is difference between "function" & "Function" ?

- Class will not allow functions directly to be configured.
- Class can call function, but can't have definition for function.

Syntax:

```
function Print(){
  console.log(`Print Function`);
}
class Demo {
  Print() {     // method
     console.log('Print Method');
     Print();  // function call
  }
}
let obj = new Demo();
obj.Print();
```

- Function
- Method
- Procedure

Configuring a Function

====================

- It comprises of
  - a) Declaration
  - b) Definition
  - c) Signature

Syntax:

```
   function  Hello(name:string):void
   {

   }

   function Hello(params):void          → Declaration
   {  }                        → Definition
   Hello(args)                          → Signature
```

  * Declaration is for compiler to understand the type of value and parameters that function uses and returns.
  * Definition comprises of actions to perform.
  * Signature is used to access and use the function.

Ex:

```
function Hello():void {
  console.log(`Hello TypeScript`);
}
Hello();  → Signature
```

- There are various techniques of defining a function
1. function keyword
2. Anonymous type
3. Function Constructor
4. Lambda Notation

function keyword:
It uses "function" keyword and a function name.

Syntax:
```
        function Hello():void {
        console.log("Hello");
        }
```

Anonymous type:
-It refers to a function that doesn't have a name.
-It is loaded into a named location (variable/property) and accessed by using the named location.
-They are used in call back mechanism, where function is called and executed automatically according to the state and situation.

Syntax:
```
let Hello = function(){
  console.log(`Hello TypeScript`);
}
Hello();
```

FAQ: Can we define and access anonymous function without named location?
A. Yes.

Syntax:
```
(function(){
  console.log(`Hello TypeScript`);
})
();
```

Function Constructor:
- It allows to construct a function dynamically.
- It can configure and change the functionality as per the state and situation.

- It is defined with arguments which are string type.
- Arguments include param name, and return value.

Syntax:
```
let param1:string = "a";
let param2:string = "b";
let operation:string = "return a + b";

let f1 = new Function(param1, param2, operation);
console.log(f1(10,2));
```

FAQ: what is difference between "function" & "Function"?
A.
 "function" defines a function statically.
 "Function" defines a function dynamically.

Lambda Notation for Function:
- It is used to minify the logic.
- It is technique used to define function in a short hand way
function
FunctionConstructor
Syntax:
```
let f1 = new Function("console.log('statement1'); console.log('statement2')");
f1();

let f1 = new Function("a+b", "a", "b")
```

Syntax:
```
let f1 = new Function("return a+b","a","b");  // invalid
```

Syntax: invalid can't define the type. It is by default any type, which uses type inference.

```
let f1 = new Function("a:number","b:number","return a + b");
```

LAMBDA
- It is a short hand notation of writing function.

- It allows to Minify the code.

Syntax:

```
          params=>return
          (params)=>return
          (params)=>{ }
```

( )	→ function parameters
=>	→ definition and return
{ }	→ block fo encapsulating statements

Syntax:
Defined in TS : ()=>{}

Returns in JS : (function () { });

- All LAMBDA functions are anonymous.

Syntax:

```
let hello = ()=>{}
hello();
```

Ex: with param

```
let hello = uname=> console.log(`Hello ! ${uname}`);
hello("John");
```

Ex: with multiple params

```
let hello = (name,price)=> console.log(`Name=${name}\nPrice=${price}`);
hello("TV", 45000.44);
```

Ex: without parameters

```
let hello = ()=> console.log(`Welcome to LAMBDA`);
hello();
```

Ex: With strongly typed params

```
let hello = (name:string, price:number)=>
console.log(`Name=${name}\nPrice=${price}`);
hello("TV", 45000.55);
```

Ex: With multiple statements

```
let hello = ()=> {console.log('Statement-1'); console.log('Statement-2');};
hello();


products.filter(function(product) {
                return product.Category=="Electronics"
              });
products.filter(product=>product.Category=="Electronics").length;
```

Ex:

```
let products:any = [
   {Name:"Samsung TV", Category:"Electronics"},
   {Name:"Nike Casuals", Category:"Footwear"},
   {Name:"Speakers", Category: "Electronics"},
   {Name:"Mobile", Category: "Electronics"}
];

let results =
products.filter(product=>product.Category=="Electronics"&&product.Name=="M
obile");
let electronicsCount:number =
products.filter(product=>product.Category=="Electronics"&&product.Name=="M
obile").length;
for(var item of results){
   console.log(item.Name);
}
console.log(`Total No of Electronic Products : ${electronicsCount}`);
```

Function Parameters
Function Return Types
Function Recursions

Function Parameters

- Function can be parameter less
- Function can be parameterized.

What is the purpose of parameter in a function?
- A parameter in function can modify the functionality.
- Same function can be handled for different states.

Syntax: without parameter

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

Syntax: with parameter

```
function PrintNumbers(upto:number) {
   for(var i=1; i<=upto; i++) {
      console.log(i);
   }
}
PrintNumbers(5);
console.log(`---------------`);
PrintNumbers(8);
```

Syntax:

```
        function  Name(param:type)   → Formal Parameter
        {
        }
        Name(10) → Actual Parameter / Arguments
```

- Every parameter defined in function definition is by-default mandatory.
- You can define optional parameters by using "?".
- A required parameter cannot follow an optional parameter.

Syntax:

```
function Product(Id?:number, Name:string){  → invalid

}
Product(1, "TV");
```

Syntax:

```
function Product(Id:number, Name?:string){  → valid

}
```

- If optional parameter is not define then you have handle it by using "undefined" type

Syntax:

```
function Product(Name:string, Price?:number){
    if(Price==undefined) {
        console.log(`Name=${Name}`);
    } else {
   console.log(`Name=${Name}\nPrice=${Price}`);
    }
}
Product("Samsung TV");
console.log(`--------------`);
Product("Mobile", 12000);
```

Note: Parameters have order dependency that is the reason why optional parameters are not allowed to be followed by required parameters.

- A parameter can handle any type of value both primitive, non primitive and even a function.

Ex: Array as parameter

```
function PrintList(list:string[]){
    for(var item of list) {
        console.log(item);
    }
}
PrintList(['TV', 'Mobile']);
```

```
console.log(`--------------`);
PrintList(new Array('Shoe','Watch'));
console.log(`--------------`);
let categories:string[] = ['Electronics', 'Footwear'];
PrintList(categories);
```

- You can define a function with multiple array type parameters along with other parameter in any order.

Ex:

```
function PrintList(list:string[], sales:number[], count:number){
   for(var item of list) {
      console.log(item);
   }
   for(var sale of sales){
      console.log(sale);
   }
   console.log(`Count = ${count}`)
}
PrintList(['A','B'],[10,20],2);
```

- ES6 introduced Rest parameters
- A rest parameter can allow multiple values as arguments.
- One parameter is enough to handle all arguments.
- A rest parameter is defined with "..."
- Every function can be defined with only one rest parameter.
- Rest parameter must be the last parameter in formal list.
- Rest parameter reads upto end of arguments.

Ex:

```
function PrintList(count:number, ...list:any) {
   for(var item of list) {
      console.log(item);
   }
   console.log(`Count=${count}`);
}
PrintList(2,"TV",true)
```

- Object as parameter

Syntax:
```
function PrintList(obj:any) {
    for(var property in obj) {
        console.log(`${property} : ${obj[property]}`)
    }
}
PrintList({Name:'TV', Price:45000});
```

- Function as parameter is never allowed, functions are passed as arguments.

Syntax:
```
let pwd:string = "admin123";
function PrintList(success:any, failure:any) {
    if(pwd=="admin12"){
        success();
    } else {
        failure();
    }
}
PrintList(function(){
    console.log('Login success..');
}, function(){
    console.log('Invalid Password');
})
```

Function Return Types
FAQ: Why a function requires Return Type?
A. To return a value

FAQ: Why and When a function is deisgned to return a Value?
A. When Functions are designed to build any expression then every expression must return a value, which is specified by using return type.

Ex:
```
function Message(str):string {
    return str.toUpperCase();
}
function power(x,y):number {
    return Math.pow(x,y);
}
let msg:string = Message("Welcome to TypeScript");
let pow:number = power(2,3);
console.log(msg);
console.log(pow);
```

- If function is not intended to return any value then it is not defined to return a type. You can configure such functions as "void".

- Void type derived from "C and Algol68"
- It will not provide any result to caller.
- It comprises of function statements to execute.

```
Returning    and                    Rendering
-----------------------------------------------------------------------------
Evaluates a value                   Generating the statements
and return the value                and render output as
to keep in any reference            response
and use in application
```

void → rendering
type → returning

FAQ: Can a function with void as return type can use the "return" keyword?
A. Yes.

FAQ: What is the purpose of return keyword in a void function?
A. It is define un-reachable code with in function. So that compiler can't reach the code. It is mostly used by developers in Unit Testing.

Ex:

```
function PrintMessage():void{
  console.log("Welcome to TypeScript");
  console.log("This is second line");
  console.log("This is incomplete don't execute");
  return;
  console.log("This is pending");
}
PrintMessage();
```

FAQ: Can a function return more than one type of value?
A. Yes. TypeScript supports Union of Type.

Ex:
```
function Demo(param):number | string {
    if((typeof param)=="number") {
        return param + 1;
    } else {
        return `Your Message : ${param}`;
    }
}
console.log(Demo(10));
console.log(Demo("Welcome to TypeScript"));
```

Function Recursion

- Recursion is technique where a function is called with in the context.

Ex:
```
function fact(n){
    if(n==0){
        return 1;
    } else {
        return n * fact(n-1);
    }
}
console.log(`Factorial of 5 is ${fact(5)}`);
```

Class and Its Members
- Properties
- Constructor
- Methods / Functions
- Accessors
- Inheritence
- Abstraction

Polymorphism

- It is one of the key feature of OOP.
- It is a feature that allows a component to work for various situations with different behaviors.

- One component is enough for various purpose.
- In OOP you can design a component with different behaviours by allocating different types of memory.

```
let obj:base = new Dervied();       → Valid
let obj:derived = new Base();       → Invalid
```

Ex:

```
class Employee {
   public FirstName:string;
   public LastName:string;
   public Designation:string;
   public Print(){
      console.log(`${this.FirstName} ${this.LastName} - ${this.Designation}`);
   }
}
class Developer extends Employee {
   public Print(){
      super.FirstName = "Raj";
      super.LastName = "Kumar";
      super.Designation = "Developer";
      super.Print();
   }
}
class Admin extends Employee {
   public Print(){
      super.FirstName = "Kiran";
      super.LastName = "Kumar";
      super.Designation = "Admin";
      super.Print();
   }
}
class Manager extends Employee {
   public Print(){
      super.FirstName = "Tom";
      super.LastName = "Hanks";
      super.Designation = "Manager";
      super.Print();
   }
}
let employees:Employee[] = new Array(new Developer(), new Admin(), new
Manager());
for(var emp of employees) {
   emp.Print();
}
```

Polymorphism?
- A single base class object using the memory of multiple derived classes.

EX:

```
class Employee {
   public FirstName:string;
   public LastName:string;
   public Designation:string;
   public Print(){
      console.log(`${this.FirstName} ${this.LastName} - ${this.Designation}`);
   }
}
class Developer extends Employee {
   public Print(){
      super.FirstName = "Raj";
      super.LastName = "Kumar";
      super.Designation = "Developer";
      super.Print();
   }
}
class Admin extends Employee {
   public Print(){
      super.FirstName = "Kiran";
      super.LastName = "Kumar";
      super.Designation = "Admin";
      super.Print();
   }
}
class Manager extends Employee {
   public Print(){
      super.FirstName = "Tom";
      super.LastName = "Hanks";
```

```
        super.Designation = "Manager";
        super.Print();
    }
}
let employees:Employee[] = new Array();
employees[0]= new Developer();
employees[1]= new Admin();
employees[2] = new Manager();
for(var emp of employees) {
    emp.Print();
}
```

Task

- Create a method that can handle various functionalities
  single method that can accept parameter less, parametereized
    Method() { print message }
    Method(a,b) { addition of a and b }

- Create single method that can handle operations
    Method(10,20, add);
    Method(2,4, mul);

```
class Calculator {

    public  calculate(num1: number, num2: number): void{};

    public static calculateTest(num1: number, num2: number, type: string) {

        let objs: Calculator[] = new Array();

        objs[0] = new Addtion();

        objs[1] = new Mul();

        let [add, mul] = objs;

        let obj = eval(type);

        obj.calculate(num1, num2);
    }
}
class Addtion extends Calculator {
    public calculate(num1: number, num2: number): void {
        console.log(num1 + num2);
    }
}
class Mul extends Calculator {
    public calculate(num1: number, num2: number): void {
        console.log(num1 * num2);
    }
}
Calculator.calculateTest(5, 7, "mul");
Calculator.calculateTest(5, 7, "add");
```

Generics in TypeScript

- Generic is type safe.
- You can configure any component as Type Safe component.
- However it use strongly typed feature for component when it knows the type of value to deal with.
- The datatype initially not know to compiler, when a value is provided according to the value type it can fix the data type of any reference and will not allow any another type.
- TypeScript can use Generics for properties, methods, and classes.

Syntax:

```
        function functionName<T>(param:T): T {
        }
```

Ex:

```
function PrintValue<T>(param:T):T {
   return param;
}
console.log(PrintValue<string>("String Value"));
console.log(PrintValue<number>(10));
```

Note: You can't use any operator on Generic type. You have handle all operations using functions.

Ex:

```
function Sum(a:any, b:any) {
   return a+b;
}
function Concat(a:any, b:any) {
   return `First Name=${a}\nLast Name=${b}`;
}

function PrintValue<T>(a:T, b:T):any{
   if((typeof a)=="number" && (typeof b=="number")){
      return Sum(a,b);
   } else {
      return Concat(a,b);
   }
}
console.log(PrintValue<number>(10,20));
console.log(PrintValue<string>("Tom","Hanks"));
```

- Generics are mostly used even with User Defined Types

Ex: Generic method with user defined type

```
interface IProduct {
   Name:string;
   Price:number;
   InStock:boolean;
}
```

```
interface IEmployee {
    Name: string;
    Designation:string;
}
function GetData<T>(obj:T) {
    for(var property in obj) {
        console.log(`${property} : ${obj[property]}`);
    }
}
console.log(`Product Details:`);
GetData<IProduct>({Name: 'TV', Price: 34000.44, InStock: true});
console.log(`----------------`);
GetData<IEmployee>({Name: 'John', Designation: 'Manager'});
```

Ex: Generic method handling Array Type

```
interface IProduct {
    Name:string;
    Price:number;
    InStock:boolean;
}
function GetData<T>(obj:T) {
    if(obj instanceof Array){
        for(var item of obj) {
            console.log(item);
        }
    } else {
        for(var property in obj){
            console.log(`${property} : ${obj[property]}`);
        }
    }
}
let product:IProduct = {
    Name: 'Nike Casuals',
    Price: 3400.44,
    InStock: true
};

let products:IProduct[] = [
    {Name: "TV", Price: 45000.55, InStock:true},
   {Name: "Mobile", Price: 12000.33, InStock:false}
];
console.log(`Products List:`);
GetData<IProduct[]>(products);
console.log(`-------------`);
GetData<IProduct>(product);
```

Generic Class
- Generic class allows to configure generic type memory of members in a class.
- While allocating memory you can define type for members. so that all members with the same type specified.

EX:

```
class Demo<T> {
   value:T;
   add:(x:T, y:T)=>T;
}
let obj = new Demo<number>();
obj.value = 10;
obj.add = function(x, y) { return x + y };
console.log(`Number=${obj.value}`);
console.log(`Addition=${obj.add(10,20)}`);

let obj2 = new Demo<string>();
obj2.value = "Hello !";
obj2.add = function(x, y) { return `${x} ${y}`};
console.log(obj2.value);
console.log(obj2.add("Raj", "Kumar"));
```

Generic Class with User Defined Type
==============================

```
class Oracle {
   connection:string;
   query:string;
   connectionState:string;
}
class MongoDb {
   connection:string;
   command:string;
}
class DataSource<T> {
   connection:T;
}
let ora = new DataSource<Oracle>();
ora.connection = {connection:"OracleConnection", query:"Select * from
tblproducts", connectionState:"Connected"};
console.log(`Provider=${ora.connection.connection}\nQuery=${ora.connection.qu
ery}\nConnection State=${ora.connection.connectionState}`);

console.log(`--------------------`)
let mongo = new DataSource<MongoDb>();
mongo.connection = { connection:"Mongoose", command:
"db.tblproducts.insert()" };
console.log(`Provider=${mongo.connection.connection}\nCommand=${mongo.co
nnection.command}`);
```

TypeScript Maps and Sets

=====================

- Introduced from ES6 version.
- Maps is used to deal with dictionary type collection.
- A Key / Value collection.
- Object can also have key value type collection
- Object is Schema Less untill or unless you defined as strongly type.
- Map is used to configuring memory map which can handle strongly typed keys and value with structured data.
- Map is object that iterates its elements in insertion order.
- It internally uses "for..of" that returns an array of key and value.

Map	Object
Doesn't contain default keys.	Contains default keys, which can Collide with your own keys.
Key can be any type, including functions, object any primitive type.	Keys must be either a string or a symbol.
The number of items	The number of items in an

in a Map is easily retrived from its size property	Object must be determined manually.
Map is an iterable so that it can be directly iterated	Iterating over an object requires obtaining its keys and values with "for...in"
Performance is better as you can easily add or remove keys dynamically.	Not optimized for frequent additions and removals of key and value paris.

Map in TypeScript

Method	Description
set()	add a new item into Map collection
get()	read value for Map collection using a key reference.
delete()	Remove any item from Map based on key reference.
has()	To verify whether the item available or not. It checks and returns boolean true or false.
clear()	Remove all items from Map collection.
keys()	Return all keys
values()	Return all values
entries()	Return both key and value
size	To return the count of items in collection.

Syntax-1:
```
let product = new Map();
product.set(1, "Samsung TV");
product.set(2, "Nike Casuals");
```

Syntax-2:
```
let product = new Map()
                .set(1, "Samsung TV")
                .set(2, "Nike Casuals");

Syntax-3:
let product = new Map([
   [1, "Samsung TV"],
   [2, "Nike Causuals"]
]);

Ex:
let product = new Map([
   [1, "Samsung TV"],
   [2, "Nike Causuals"]
]);
console.log(product.get(1));
console.log(product.size);
product.delete(1);
product.has(3);        // return false


Ex:
let product = new Map([
   [1, "Samsung TV"],
   [2, "Nike Causuals"]
]);
// read all keys
console.log(`-------Keys---------`);
for(let key of Array.from(product.keys())){
   console.log(key);
}
//read all values
console.log(`-------Values-------`);
for(let value of Array.from(product.values())){
   console.log(value);
}
//read all entries
console.log(`--------Entries-------`);
for(let entry of Array.from(product.entries())){
   console.log(entry);
}
```

Set in TypeScript

- Set is a collection of Keys.
- Functions

```
add()  - to add values
has()  - to verify value
delete()      - delete a value
size   - return the size      etc..
```

Ex:

```
let categories = new Set();
categories.add("Electronics");
categories.add("Footwear");

for(let entry of Array.from(categories)){
   console.log(entry);
}
```

Enum

- It is an Enumerator.
- In computer programming Enum refers to Enumerated type.
- It is also known as enumeration, enum or factor [R language]
- It is a datatype that consists of a set of named values, which are called as elements, members, enumerals or enumerator.
- It comprises of collection of constants.

Indexing Property in Interface

=========================

Ex:

```
interface CustomArray {
   [index:number]:string;
}
let products: CustomArray;
products = ["TV", "Mobile"];
console.log(products[0]);
```

Task-1 Assignment related to Map and Set

```
//adding entries in map using set() method


//let products = new Map();
//products.set(1,{Name:"TV",Price:84000.67});
//products.set(2,{Name:"Mobile",Price:13000.87});
//products.set(3,{Name:"Footwear",Price:3000.78});


//Initializing Map with a an array with key -value pair


let product = new Map(
   [
      [1,{Name:"TV",Price:84000.67}],
      [2,{Name:"Mobile",Price:13000.87}],
      [3,{Name:"Footwear",Price:3000.78}]


]
);
//let Keys=product.keys();
console.log('Adding all keys of Map into Set');
for (var key of Array.from (product.keys()))
{
   //console.log(key);
```

```
    //store map keys in set
     let Keys=new Set();
     Keys.add(key);
    for(var k of Array.from(Keys)){
        console.log(k);
    }


}
console.log('Adding all values of Map into Set');
// let Values=product.values();
 for( let v of Array.from(product.values())){
for (let property in v){
    //console.log(`${property}: ${v[property]}`);
    console.log(`${property}:`);
    let Values =new Set();
    Values.add(v[property]);
    for(let value of Array.from(Values)){
        console.log(value);
    }
}


}
console.log("----Adding product name into set-----");
console.log('Product Names Are');


for( let v of Array.from(product.values())){

    for (let property in v){


        //console.log(`${property}:`);

        if (property == "Name"){

        let Names =new Set();

        Names.add(v[property]);


        for(let value of Array.from(Names)){

            console.log(value);

        }   }   }   }
```

Enums
- Enums can store and access a set of numeric and string constants.
- The keyword "enum" is used to create an Enumeration.
- Enum can dynamically configure its values based on existing values.

Syntax:

```
        enum  EnumName
        {
            Name = value;
        }
        EnumName.Name
```

Ex:

```
enum ErrorCodes
{
   OK = 200,
   Created = 201,
   Accepted = 202,
   NonAuthorative = 203,
   NoContent
}
console.log(`NoContent Status Code: ${ErrorCodes.NoContent}`);          // 204
```

Ex:

```
enum ErrorCodes
{
   OK,
   Created = 201,
   Accepted = 202,
   NonAuthorative = 203,
   NoContent
}
console.log(`NoContent Status Code: ${ErrorCodes.NoContent}`)
console.log(`OK : ${ErrorCodes.OK}`);
```

If no previous value then the value starts with 0.

Ex

```
enum ErrorCodes
{
   OK,
   Created = 201,
   Accepted = 202,
   NonAuthorative,                    // 203
   NoContent=205
}
console.log(`NoContent Status Code: ${ErrorCodes.NoContent}`)
console.log(`OK : ${ErrorCodes.OK}`);
```

Note: Enum can initialize value dynamically only for numeric type constants.

- Enum can be a collection of string constants.
- All string constants must be initialized with a value.
- Enum can't dynamically configure values if they are string type.

- Enum configure a numeric value for string constants it it is defined as starting index value.

Ex:
```
enum Play
{
   Last,                          // valid - 0
   Next = "Next Song",
   Prev = "Previous Song",
   First = "First Song"
}
console.log(`Last Value ${Play.Last}`);
console.log(`Prev : ${Play.Prev}`);
```

Ex:
```
enum Play
{
   Next = "Next Song",
   Prev,                          // invalid - not allowed
   First = "First Song"
}
console.log(`Last Value ${Play.Last}`);
console.log(`Prev : ${Play.Prev}`);
```

- Enum can contain collection of both string and numeric constants.

Ex:
```
enum Play
{
   Last = 0,
   Next = "Next Song",
   Prev = 1,
   First = "First Song"
}
```

- Enum values can't be re-defined as they are configured as constants in memory.

```
        Play.Prev = 2; // invalid
```

FAQ:
1. Can we declare enum with "const" as access specifier?
A. Yes. Const enums are used to configure enums with expressions?

2. Can enum have an expression?
A. Yes

3. Can your defined a boolean type constants?
A. Directly the boolean true and false are not allowed, JavaScript can handle them with 1 and 0.

```
const enum Expressions
{
   A = true,
   B = false, // invalid
};
```

4. Can we use all type of expressions in Enum?
A. No. Enum support only following operators

```
        +, - , *, / , %
        <<  , >> , >>>
        &, | , ^
```

5. What is Heterogeneous enum?
A. Combination of both string and numeric type

6. What is Ambinet Enum?
A. All enums are by default Non-Ambient enums,
All are constants and have initializer.
Ambinent enum refers to non-constant values in a collection, they doesn't have initializer.

Syntax:

```
  enum ErrorCode {
     NotFound = 404,
     TimeOut,        // ambient
     MethodNotAllowed = 405
  }
  TimeOut = 405
  Method Not Allowed = 405
```

7. What is Enum Reverse Mapping?
A. It is a technique used to access the name of enum constant based on its value.

Ex:
```
enum ErrorCode
{
   NotFound = 404
}
let a = ErrorCode.NotFound;
console.log(`a = ${a}`);
let statusText = ErrorCode[a]; // revers mapping
console.log(`Status Text= ${statusText}`);
```

Enums, Modules and Namespace

Ex:
```
enum Size {
   Height,
   VSize
}
interface Table {
   Vertical:Size.Height;
   Color:string;
}
interface Div {
   Vertical:Size.VSize;
}
let empTable: Table = {
   Vertical:Size.VSize,    // invalid
   Vertical:Size.Height,   // valid
   Color: "Red"
};
let navDiv: Div = {
   Vertical:Size.VSize     // valid
   Vertical:Size.Height    // invalid
}
```

Modules in TypeScript

- Modules introduced into JavaScript with ECMAScript 2015

- TypeScript uses modules of JavaScript.
- Modules is like a component in any framework with collection of various members.
- You can import modules from external source and use in your application.
- Modules are executed with in their own scope, not in the global scope, the variables, functions, classes all are declared in a module and they are not accssible outside.
- To make them accessible outside you have to mark them with "export"
- You can import any module by using "import" reference.
- JavaScript requires AMD [Asynchronouse Module Distribution] and Common JS like systems to handle modules.
- TypeScript can implicitly handle modules.

Modules in TypeScript

- Modules introduced into JavaScript with ES 5
- Modules are executed within their own scope and not in the global scope.
- A module is a collection of variables, functions and classes etc.
- A module is used to maintain a repository of functions and class for a library.
- Modules are used to build library.
- In order to handle modules JavaScript based language uses AMD [Asynchronous Module Distribution] and Common JS system.
- Module have its own scope. In order to make the members of a module accessible outside the module you have to mark them as export.

```
export class className {
 }
```

- To implement the module members in any another module you have to "import" the module and classes.

Ex:
1. Create a new folder by name "Modules"
2. Add sub folders
    - Contracts
    - Templates
    - Services
    - App
3. Controls folder add file
        "IProduct.ts"

```
export interface IProduct {
    Name:string;
    Price:number;
    Qty:number;
    Total():number;
    Print():void;
}
```

4. Templates folder add file
          "AbstractProduct.ts"

```
import { IProduct } from '../Contracts/IProduct';

export abstract class ProductTemplate implements IProduct {
   public Name:string = '';
   public Price:number = 0;
   public Qty:number = 1;
   public Total():number {
       return this.Qty * this.Price;
   }
   public abstract Print():void;
}
```

5. Services folder add file
           "ProductService.ts"

```
import { ProductTemplate } from '../Templates/AbstractProduct';

export class Product extends ProductTemplate {
    public Name:string = "";
    public Price:number = 0;
    public Qty:number = 1;
    public Total():number {
        return this.Qty * this.Price;
    }
    public Print():void {
        console.log(`Name=${this.Name}\nPrice=${this.Price}\nQty=${this.Qty}\nT
otal=${this.Total()}`);
    }
}
```

6. App folder add the file
           "ProductApp.ts"

```
import { Product } from "../Services/ProductService";
```

```
let tv = new Product();
tv.Name = "Samsung TV";
tv.Price=  45000.55;
tv.Qty= 2;
tv.Print();

   > tsc  ProductApp.ts
   > node ProductApp.js
```

declare - is not a keyword
Syntax:

```
declare let x;
let x : any;
```

default - You can export many variables from same file
default is used only one in while file. You can
import without using brackets. { }

Exports are of 2 types:
1. Named Exports
- Zero or more exports per module.

Ex:

```
export default interface IProduct {

}
export interface ICategory {

}

import { ICategory } from 'fileName';
import { IProduct} from 'fileName';  invalid
import  IProduct from 'fileName';     valid
```

2. Default Exports
- One per module

Namespace

- A Namespace is a collection of releated type of sub namespaces and classes.
- You can configure and import namespace in order to access the members of library.
- Anything defined in a namespace is not global in access, you have to use "export".

1. **Add a new folder**

   Project

2. **Add following folders into project folder**

- Contracts
- Templates
- Services
- App

3. **Add following file into contracts**

   "ProductContract.ts"

```
namespace Project
{
    export namespace Contracts
    {
        export interface IProduct
        {
            Name:string;
            Price:number;
            Qty:number;
            Total():number;
            Print():void;
        }
    }
}
```

4. **Add file into Templates folder**

"ProductTemplate.ts"

```
///<reference path="../Contracts/ProductContract.ts" />

import contracts = Project.Contracts;
namespace Project
{
    export namespace Templates
    {
       export abstract class ProductTemplate implements contracts.IProduct
        {
        public Name:string;
        public Price:number;
        public Qty:number;
        public Total():number {
            return this.Qty * this.Price;
        }
        abstract Print():void;
       }
    }
}
```

5. **Add file into Services folder**

"ProductService.ts"

```
///<reference path="../Templates/ProductTemplate.ts" />

import templates = Project.Templates;

namespace Project
{
    export namespace Services
    {
        export class Product extends templates.ProductTemplate
        {
            public Name:string = '';
            public Price:number = 0;
            public Qty:number = 1;
            public Total():number {
                return this.Qty * this.Price;
            }
            public Print():void {
                console.log(`Name=${this.Name}\nPrice=${this.Price}\nQty=${this.Qty
}\nTotal=${this.Total()}`);
            }
        }
    }
}
```

6. Add File into APP folder

MyApp.ts

```
///<reference path="../Services/ProductService.ts" />

import services = Project.Services;

let tv = new services.Product();
tv.Name = "Samsung TV";
tv.Price = 34000.44;
tv.Qty = 2;
tv.Print();
```

7. Compile

```
> tsc --outFile  myapp.js  myapp.ts
```

declare - is not a keyword

Syntax:

```
declare let x;
let x : any;
```

default - You can export many variables from same file
default is used only one in while file. You can
import without using brackets. { }

Exports are of 2 types:

1. Named Exports
   - Zero or more exports per module.

Ex:

```
export default interface IProduct {

}
export interface ICategory {

}

import { ICategory } from 'fileName';
import { IProduct} from 'fileName';  invalid
import  IProduct from 'fileName';     valid
```

2. Default Exports
   - One per module

Namespace

- A Namespace is a collection of releated type of sub namespaces and classes.
- You can configure and import namespace in order to access the members of library.
- Anything defined in a namespace is not global in access, you have to use "export".

1. **Add a new folder**

   Project

2. **Add following folders into project folder**

- Contracts
- Templates
- Services
- App

3. **Add following file into contracts**

   "ProductContract.ts"

```
namespace Project
{
   export namespace Contracts
   {
      export interface IProduct
      {
         Name:string;
         Price:number;
         Qty:number;
         Total():number;
         Print():void;
      }
   }
}
```

4. **Add file into Templates folder**

"ProductTemplate.ts"

```
///<reference path="../Contracts/ProductContract.ts" />

import contracts = Project.Contracts;
namespace Project
{
   export namespace Templates
   {
      export abstract class ProductTemplate implements contracts.IProduct
       {
       public Name:string;
       public Price:number;
       public Qty:number;
       public Total():number {
          return this.Qty * this.Price;
       }
       abstract Print():void;
      }
   }
}
```

5. **Add file into Services folder**

"ProductService.ts"

```
///<reference path="../Templates/ProductTemplate.ts" />

import templates = Project.Templates;

namespace Project
{
   export namespace Services
   {
       export class Product extends templates.ProductTemplate
       {
          public Name:string = '';
          public Price:number = 0;
          public Qty:number = 1;
          public Total():number {
             return this.Qty * this.Price;
          }
```

```
            public Print():void {
                console.log(`Name=${this.Name}\nPrice=${this.Price}\nQty=${this.Qty
}\nTotal=${this.Total()}`);
            }
        }
    }
}
```

6. Add File into APP folder

MyApp.ts

```
///<reference path="../Services/ProductService.ts" />

import services = Project.Services;

let tv = new services.Product();
tv.Name = "Samsung TV";
tv.Price = 34000.44;
tv.Qty = 2;
tv.Print();
```

7. Compile

```
> tsc --outFile  myapp.js  myapp.ts
```

Templates in OOP

- Template comprises of sample design with sample data, which you can customize according to your requirements and implement in your application.
- A developer need to design templates for client so that he can implement according to his requirements.
- Templates are mostly uses for "Rollouts"
    a) End to End / Implementation
    b) Upgrade
    c) Rollout
    d) Support
- Abstract class is used to design templates.
- Abstract class comprises of functionalities both implemented and to be implemented.
- Technically a class should have all members functionality implemented.

- If any functionality is not-implemented then it should be designed as "Abstract Class".
- Abstract classes are defined by using "abstract" keyword
- Abstract class contains both incomplete and implemented members.
- You have to implement an abstract class and defined all in-complete functionality.
- You can't create an object for abstract class as it is incomplete.

What is difference between a contract and template?

Contract:
- Contract comprises of only rules.
- You can't define functionality for rules in contract.
- It contains only declaration not definition.
- You can't define access modifiers for contract members.

Template:
- Template comprises of members without implementation and as implemented.
- Can contain both complete and incomplete.
- It can contain only declaration or declaration with definition.
- You can defined access modifers for members.

- "abstract" indicates that its functionality will be implemented later.
- If any one member is a class is abstract, then entire class must be marked as abstract.

"The mechanism of hiding the members in a class and providing implementation for the complete methods is known as "Abstraction".

Ex:
```
interface IProduct
{
   Name:string;
   Price:number;
   Qty:number;
   Total():number;
   Print():void;
}
abstract class ProductTemplate implements IProduct
```

```
{
   public Name:string;
   public Price:number;
   public Qty:number;
   public Total():number {
      return this.Qty * this.Price;
   }
   public abstract Print():void;
}
class Product extends ProductTemplate
{
   public Print():void {
      super.Name = "TV";
      super.Price = 34500.55;
      super.Qty = 2;
      console.log(`Name=${super.Name}\nPrice=${super.Price}\nQty=${super.Qt
y}\nTotal=${super.Total()}`);
   }
}
let tv = new Product();
tv.Print();
```