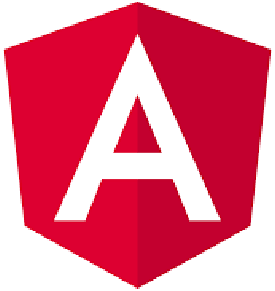
Introduction to TypeScript

Topics Covered

- Introduction to TypeScript
- Features of TypeScript
- Installation and Setup
- Basic Concepts
  - Variables
  - Data types
  - Enum
  - Array
  - Tuples
  - Functions
- OOPs concepts
  - Interfaces
  - Generics
  - Modules
  - Namespaces
- Decorators
- Compiler options
- Project Configuration

Getting Started with TypeScript

Popular Frameworks using TypeScript



Features of TypeScript

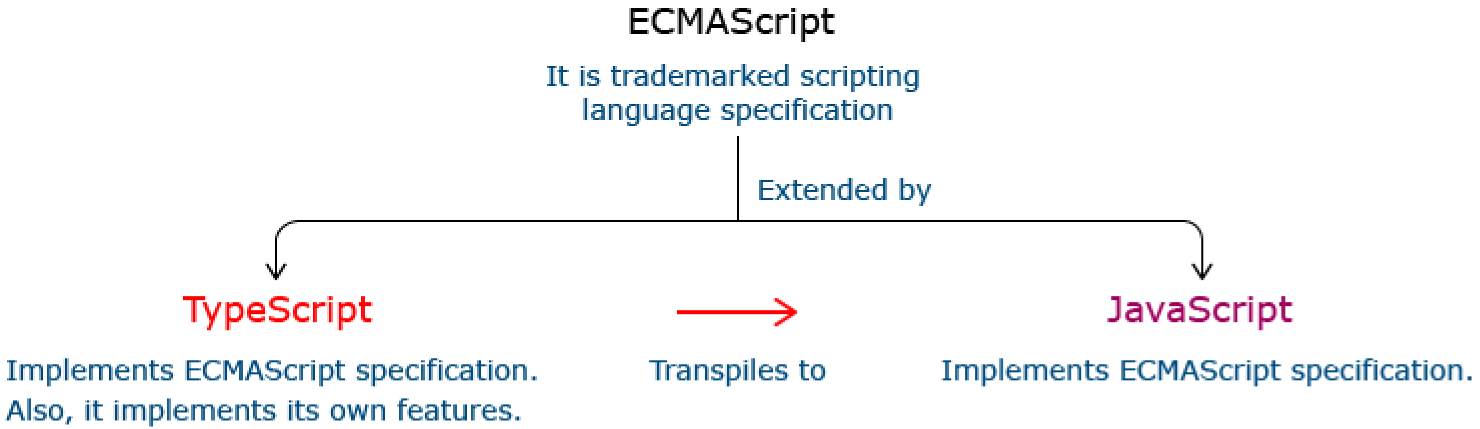
- Static Typing
- Modules Support
- Object Oriented Programming Support
- Open Source
- Cross Platform Compatibility
- Supports tools like Emacs, Vim, Atom, WebStorm

What is Typescript?

Typescript is

- Free and Open source programming language developed and maintained by Microsoft
- Superset of JavaScript and adds optional static typing to the language
- Designed for development of large applications and compiles to JavaScript
- ECMAScript 2015 support:
  - Typescript adds support for features such as classes, modules and an arrow function syntax as proposed in the ECMAScript 2015 standard.

Relationship between TypeScript and JavaScript



ECMAScript

It is trademarked scripting language specification

Extended by

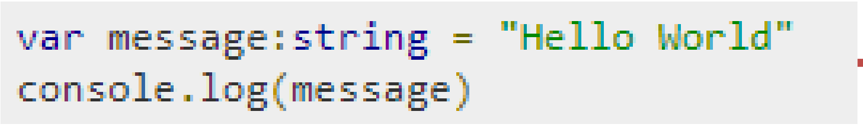
TypeScript

Implements ECMAScript specification.
Also, it implements its own features.

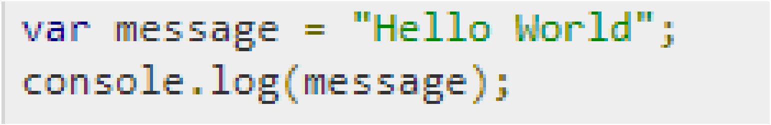
Transpiles to

JavaScript

Implements ECMAScript specification.



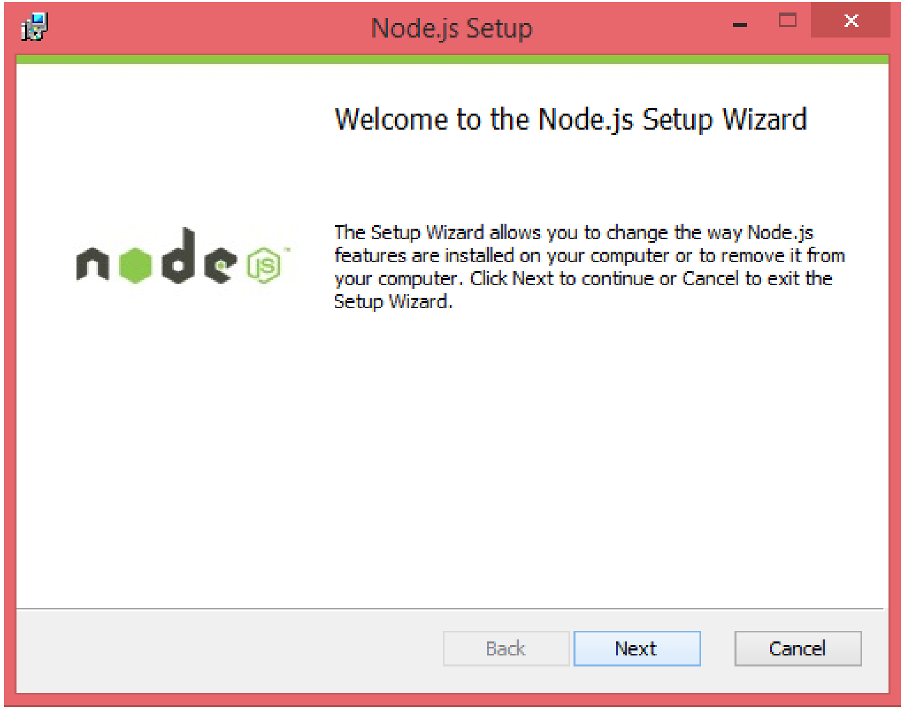
```
var message:string = "Hello World"
console.log(message)
```



```
var message = "Hello World";
console.log(message);
```

Installing TypeScript

- Step 1: Download and run the .msi installer for Node.

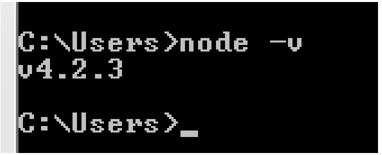


Welcome to the Node.js Setup Wizard

The Setup Wizard allows you to change the way Node.js features are installed on your computer or to remove it from your computer. Click Next to continue or Cancel to exit the Setup Wizard.

Installing TypeScript

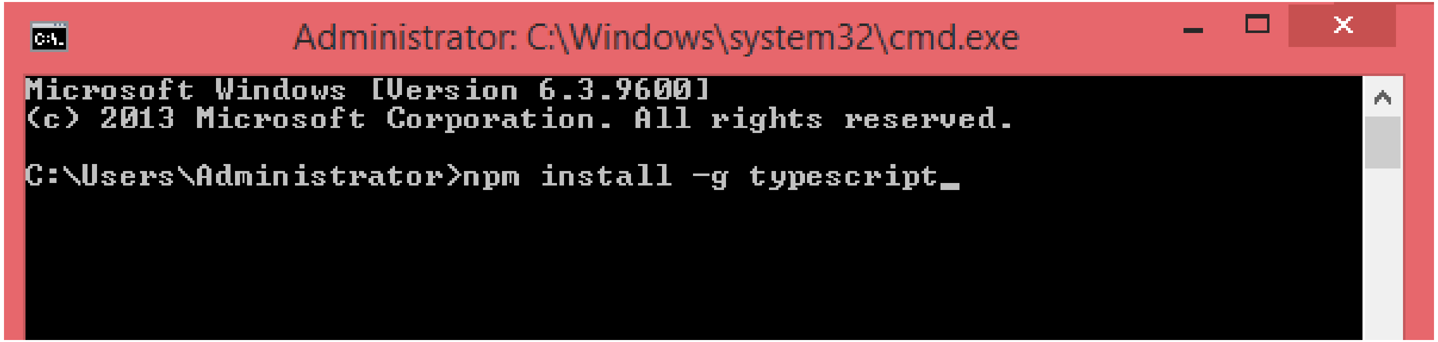
- Step 2: To verify if the installation was successful, enter the commend *node –v* in the terminal window.



```
C:\Users>node -v
v4.2.3

C:\Users>_
```

- Step 3: Type the following command in the terminal window to install TypeScript.

npm install –g typescript



Administrator: C:\Windows\system32\cmd.exe

```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\Administrator>npm install -g typescript_
```

TypeScript Basics

Declaring Variables

- Naming rules for variables in TypeScript
  - Variables contain alphabets and numeric digits
  - They cannot contain spaces and special characters except the underscore(_) and the dollar($) sign
  - Variable names cannot begin with a digit
  - Use var keyword to declare variables

- **Syntax to declare variable:**
  - Declares its type and value in one statement:

    ```
    var <variable name> : <type-annotation>  = <value> ;
    ```

  - Declares it type but no value. Variable will be set to undefined value

    ```
    var <variable name> : <type-annotation> ;
    ```

  - Declare its value but no type. Variable type will be set to any

    ```
    var <variable name> = <value> ;
    ```

  - Declares neither value nor type. Hence, type is any and value is undefined

    ```
    var <variable name> ;
    ```

Declaring Variables

- var and let keywords are used to declare variables in TypeScript like JavaScript.
- Declaring variables using var and let keywords:

```
var itemName = “Tablet”;

let itemName = “Tablet”;
```

- Scope of the variable declared using var keyword declared is outside the block within a function or class in which the block is defined.
- Whereas scope of the variable declared using let keyword is only within the block in which it is been declared.

Difference between var and let keyword

- Re-declaring block – scoped variable using var keyword
  - The var declared variable can be re-declared within the same block.

```
var itemName;
var itemName;
```

- Re-declaring block – scoped variable using let keyword
  - The let declared variable cannot be re-declared within the same block. It will throw a compilation error.

```
let itemName;
let itemName;
```

const Declaration

- The value of a variable declared using const keyword cannot be re-assigned.
- const declared variables are mutable if declared as an array or as an object literal.
- const declaration should be used if value of the variable remains unchanged.

Example:

```
const studentName = “Ram”
studentName = “Sriram”        //cannot reassign value

//Students array is declared using const keyword. Still we will be able to push data to array.
const students: string[ ] = [“John”, “Jack”, “Robin”]
students[3] = “James”;
//Cannot reassign entire array. This throws compilation error.
students = [“Richard”, “Mary”]; //Error
```

Basic Types

- Built-in basic types in TypeScript are:

Data Type	Description	
number	Double precision 64-bit floating point values used to represent integer and fractional values	let itemId: number = 4523;
string	Represents a sequence of characters	let itemName: string = “Books”;
boolean	Represents Boolean values true or false	let isDigit: boolean = true;

void	Void	Let studentId : void = undefined;
		
		function display(): void{
		Console.log(“Display Function”);
		}

Undefined/any	Undefined or any	let totalMarks: any;
		totalMarks = 60;
		//totalMarks is assigned numeric value
		totalMarks = “sixty”
		//totalMarks is assigned string value, which is acceptable since data type is any

User defined Data Types

- User defined data types include
  - Enumerations (enums)
  - Classes
  - Interfaces
  - Arrays
  - Tuple

Enum

- enum in TypeScript is used to organize a collection of related values.
- By default, enum's first item will be assigned with zero as the value and the subsequent values will be incremented by one.

Syntax: enum Enumtype {property1, property2, property3};

Ex: enum tShirtsize{xs, sm, md, l, xl}

//Value of first item will be 0(default value) and subsequent items will have sequential increment from first value

- To get the value from an enum use one of the following:

Syntax: enumName.item or enumName["item"]

Ex: tShirtsize.xs or tShirtsize[xs]

Enum

- In enum, we can set different values for one of the variable and the subsequent values will be incremented by 1.

Ex: enum tShirtsize {xs = 32, sm, md, l, xl}

```
//initial value is set to 32, so subsequent values will be 33, 34, 35, 36 respectively.
```

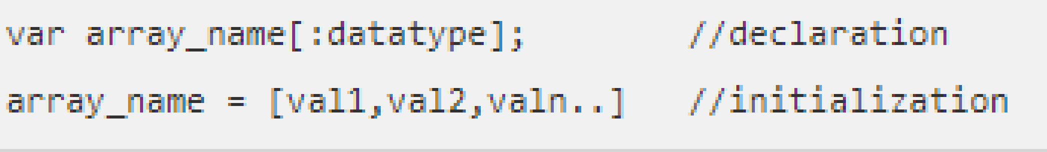
- We can even set different values to different enum items.

Ex: enum tShirtsize {xs = 32, sm=34, md=36, l=38, xl=40}

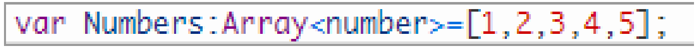
```
//All 5 items assigned with different values
```

Arrays

- Array is a homogeneous collection of values. It is a collection of the same data type.
- An array is allocated with sequential memory blocks. Each memory block representing an array element.
- **Syntax:**

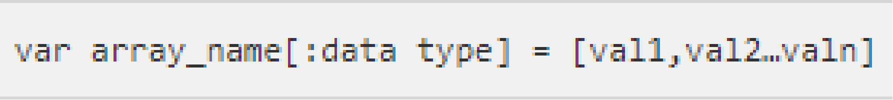


```
var array_name[:datatype];          //declaration
array_name = [val1,val2,valn..]    //initialization
```



```
var Numbers:Array<number>=[1,2,3,4,5];
```

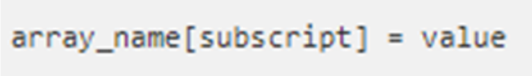
- Arrays can be declared and initialized in a single statement.



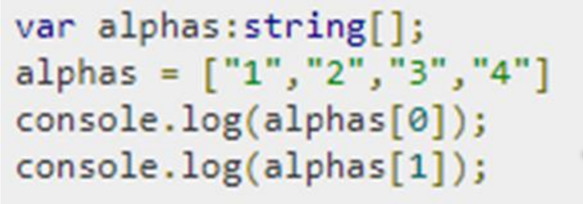
```
var array_name[:data type] = [val1,val2…valn]
```

Arrays

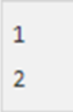
- Accessing array elements



```
array_name[subscript] = value
```



```
var alphas:string[];
alphas = ["1","2","3","4"]
console.log(alphas[0]);
console.log(alphas[1]);
```



```
1
2
```

- Using any[ ] declaration:

Ex: let StudentDetails: any[ ] = [“Ram”, 1001, “Bangalore”] //It accepts any type of data

Arrays

- To add a dynamic value to an array we can either use push function or use the index reference.
- Adding data:
  - **Adding data using push function**. Make sure that the type of pushed data is same as array type, or it will generate compilation error.

```
let states: string[ ] = [“Karnataka”, “Kerala”];
states.push(“Tamilnadu”);
states.push(“Maharashtra”);
```

  - **Adding data using index reference**

```
let states: string[ ] = [“Karnataka”, “Kerala”];
states[2] = “Tamilnadu”
states[3] = “Maharastra”
```

Arrays

- Removing Data:
- Data can be removed from an array using pop function or splice function.
  - **Using pop() function:**

    let states: string[] = [“BLR”, “MLR”, “MYS”]

    states.pop()

    Original Array: [“BLR”, “MLR”, “MYS”]

    After using pop() function: [“BLR”, “MLR”]

  - **Using splice() function:**

    let states: string[] = [“BLR”, “MLR”, “MYS”]

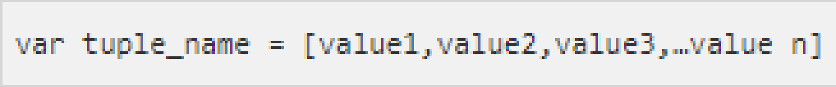
    states.splice(1,2)

    Original Array: [“BLR”, “MLR”, “MYS”]

    After using splice() function: [“BLR”]

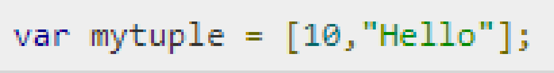
Tuple

- Tuple represents a heterogeneous collection of values.
- **Syntax**:

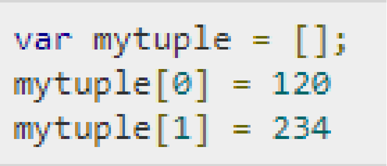


```
var tuple_name = [value1,value2,value3,…value n]
```

- **Example**:



```
var mytuple = [10,"Hello"];
```

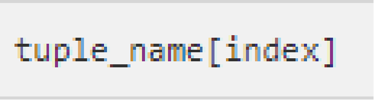


```
var mytuple = [];
mytuple[0] = 120
mytuple[1] = 234
```

Tuple

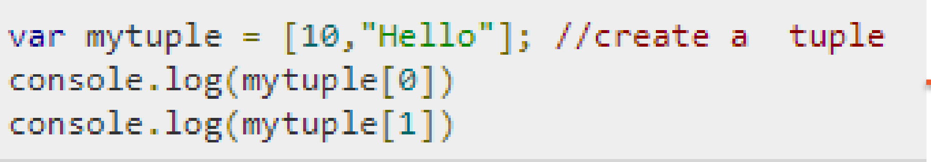
- Accessing values in Tuples:

Syntax:

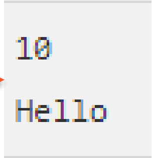


```
tuple_name[index]
```

Example:



```
var mytuple = [10,"Hello"]; //create a  tuple
console.log(mytuple[0])
console.log(mytuple[1])
```



```
10
Hello
```

Functions

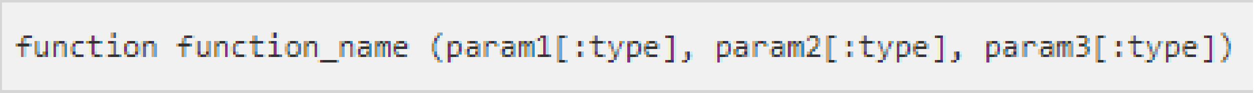
Functions in TypeScript vs JavaScript

- Function is set of statements to perform a specific task.
- They organize the program into logical blocks of code and are reusable.
- Function declaration tells the compiler about function’s name, return type and parameters.
- Function definition contains actual body of the function.

	TypeScript	JavaScript
Types	Supports	Do not support
Required and Optional Parameters	Supports	All parameters are optional
Function Overloading	Supports	Do not support
Arrow functions	Supports	Supported with ES2015
Default parameters	Supports	Supported with ES2015
Rest parameters	Supports	Supported with ES2015

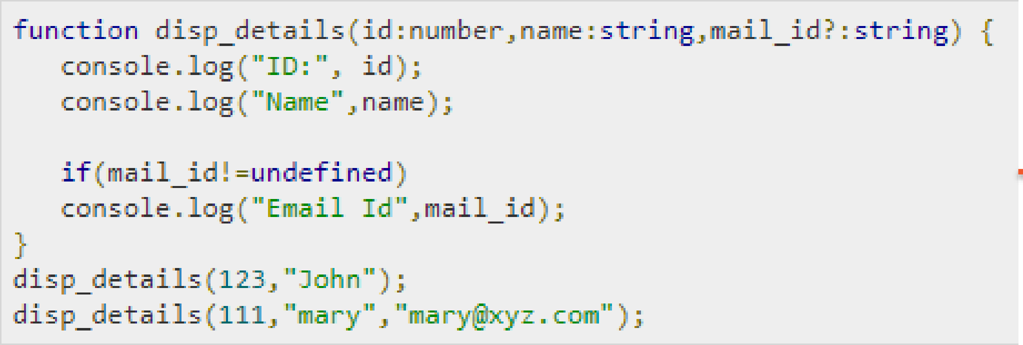
Optional parameters

- Optional parameters are used when all the values to all the parameters need not be passed mandatorily.
- A parameter can be made optional by appending a question mark to its name.
- It should be the last argument in a function.
- **Syntax**:

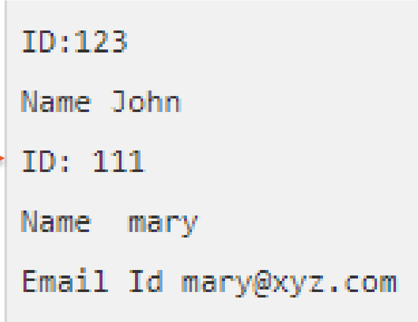


```
function function_name (param1[:type], param2[:type], param3[:type])
```

- **Example**:



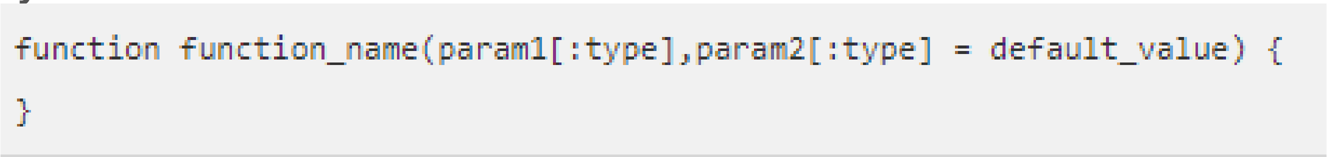
```
function disp_details(id:number,name:string,mail_id?:string) {
   console.log("ID:", id);
   console.log("Name",name);

   if(mail_id!=undefined)
   console.log("Email Id",mail_id);
}
disp_details(123,"John");
disp_details(111,"mary","mary@xyz.com");
```



```
ID:123

Name John

ID: 111

Name  mary

Email Id mary@xyz.com
```

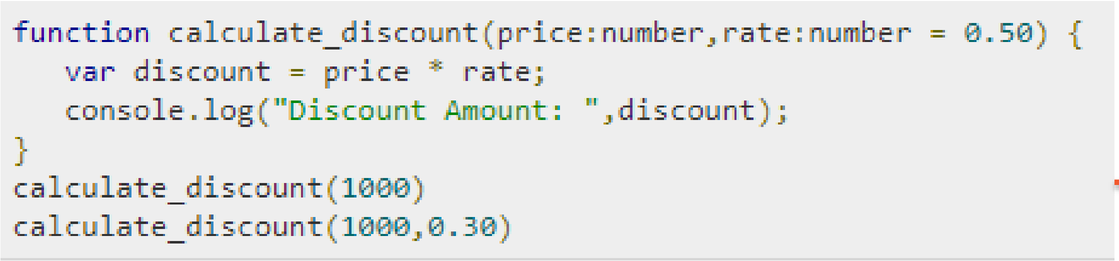
Default parameters

- Parameters in a function definition that can be assigned with default values are default parameters.
- Such parameters can be explicitly passed with other values during function invocation.
- Note: A parameter cannot be declared as both optional and default parameter in a function.
- **Syntax**:

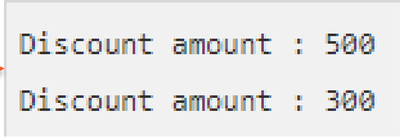


```
function function_name(param1[:type],param2[:type] = default_value) {
}
```

- **Example**:



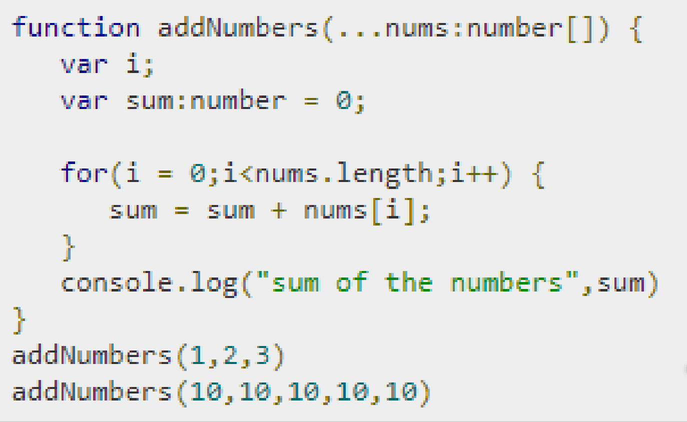
```
function calculate_discount(price:number,rate:number = 0.50) {
   var discount = price * rate;
   console.log("Discount Amount: ",discount);
}
calculate_discount(1000)
calculate_discount(1000,0.30)
```



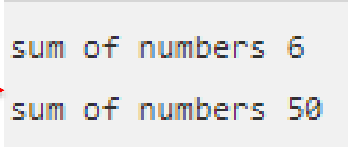
```
Discount amount : 500
Discount amount : 300
```

Rest Parameters

- Rest Parameters are variable-length arguments. They don’t restrict the number of arguments passed to a rest parameter. However, all the values passed should be of same type.
- They are like placeholders for multiple arguments of same type.
- To declare a rest parameter, it should be prefixed with three dots(…).
- All non-rest parameters of a function should come before rest parameters of that function.
- **Example**:



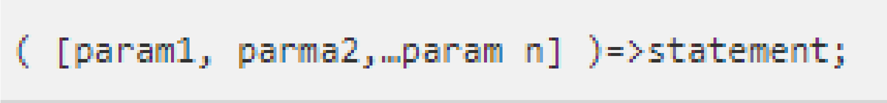
```
function addNumbers(...nums:number[]) {
   var i;
   var sum:number = 0;

   for(i = 0;i<nums.length;i++) {
      sum = sum + nums[i];
   }
   console.log("sum of the numbers",sum)
}
addNumbers(1,2,3)
addNumbers(10,10,10,10,10)
```



```
sum of numbers 6
sum of numbers 50
```

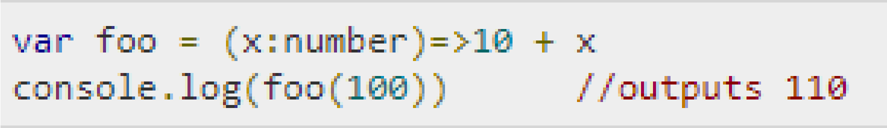
Lambda/Arrow Functions

- It is an anonymous function expression that points to a single line of code.
- Lambda functions have 3 parts:
  - Parameter – parameters are optional in these functions
  - Fat arrow notation/Lambda notation (=>) – called goes to operator
  - Statements – represents the function instruction set.
- **Syntax**:



```
( [param1, parma2,…param n] )=>statement;
```

- **Example**:



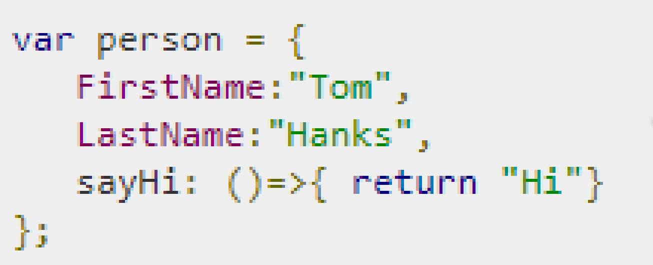
```
var foo = (x:number)=>10 + x
console.log(foo(100))      //outputs 110
```

Interfaces

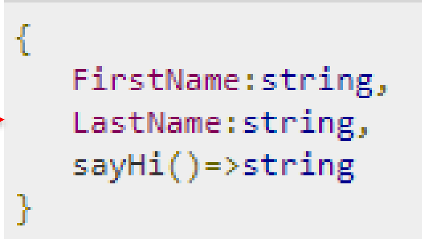
Interface

- Interface defines a syntax that any object created using the interface must adhere to.
- It defines properties and methods which are members of the interface.
- Interface contains only declaration of members and it is the responsibility of deriving class to define the members.
- Interface help in having a standard definition across all the derived classes.

Object ------> **Its Signature**



```
var person = {
   FirstName:"Tom",
   LastName:"Hanks",
   sayHi: ()=>{ return "Hi"}
};
```

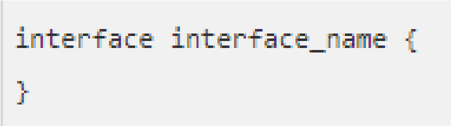


```
{
   FirstName:string,
   LastName:string,
   sayHi()=>string
}
```

Interface

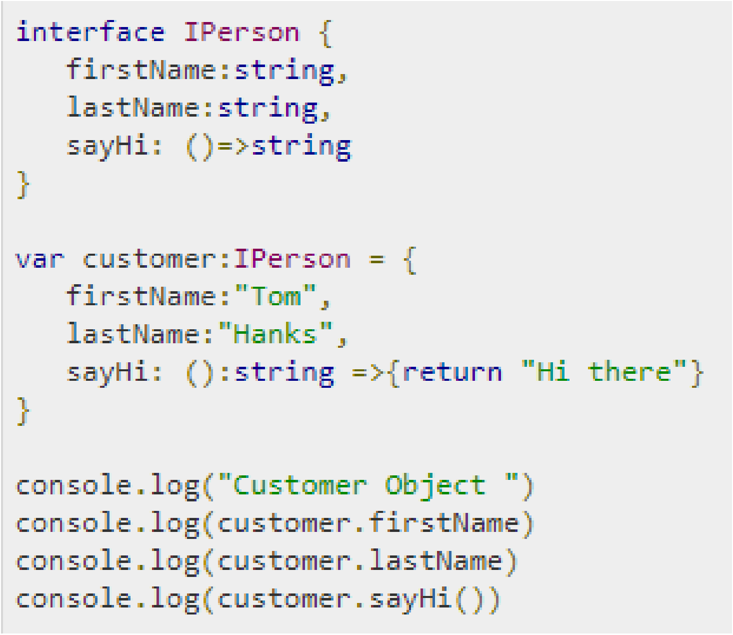
Declaring Interfaces

Syntax:



```
interface interface_name {

}
```

Example:

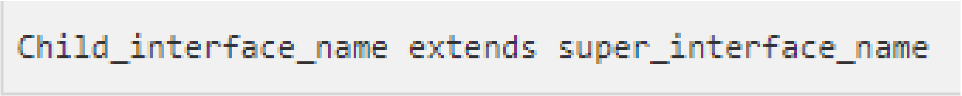


```
interface IPerson {
   firstName:string,
   lastName:string,
   sayHi: ()=>string
}

var customer:IPerson = {
   firstName:"Tom",
   lastName:"Hanks",
   sayHi: ():string =>{return "Hi there"}
}

console.log("Customer Object ")
console.log(customer.firstName)
console.log(customer.lastName)
console.log(customer.sayHi())
```

Extending interfaces – Single inheritance

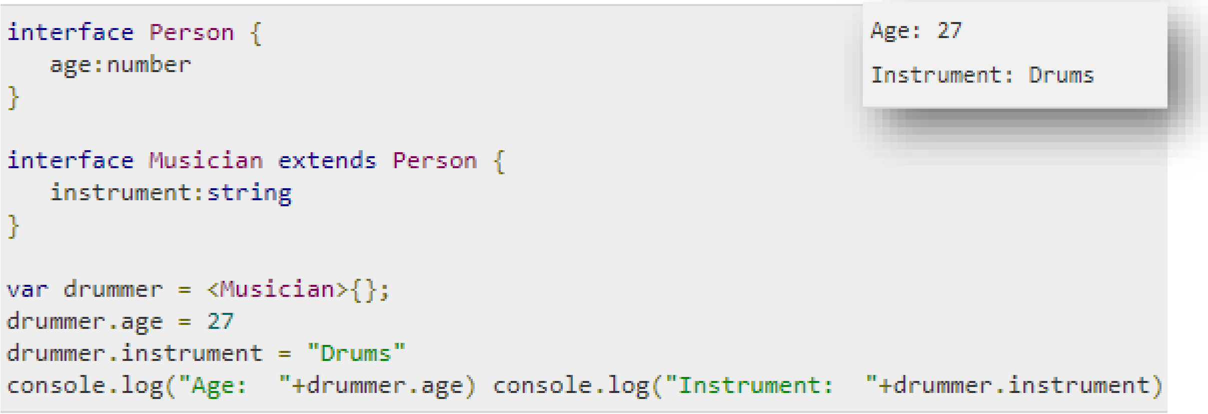
- An interface can be extended from already existing interface using the extends keyword.

Syntax:



```
Child_interface_name extends super_interface_name
```

Example:

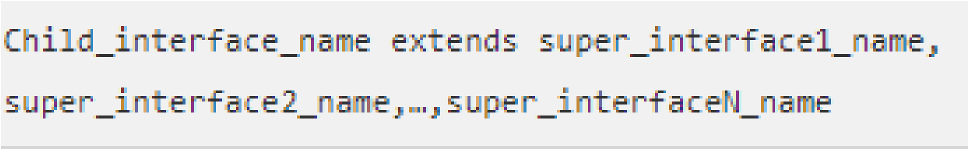


```
interface Person {
   age:number
}

interface Musician extends Person {
   instrument:string
}

var drummer = <Musician>{};
drummer.age = 27
drummer.instrument = "Drums"
console.log("Age:  "+drummer.age) console.log("Instrument:  "+drummer.instrument)
```

```
Age: 27

Instrument: Drums
```

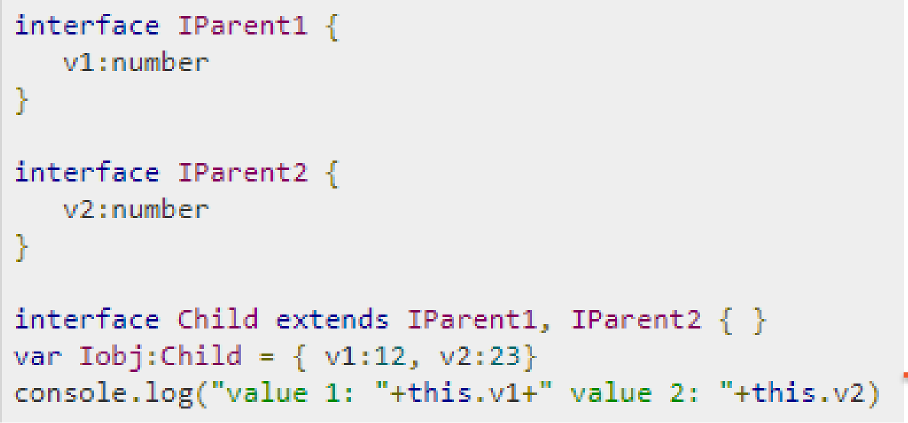
Extending interface – Multiple inheritance

Syntax: Multiple Inheritance

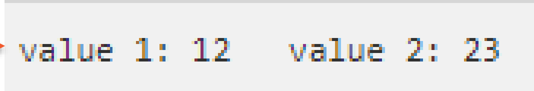


```
Child_interface_name extends super_interface1_name,

super_interface2_name,…,super_interfaceN_name
```

Example:



```
interface IParent1 {
   v1:number
}

interface IParent2 {
   v2:number
}

interface Child extends IParent1, IParent2 { }
var Iobj:Child = { v1:12, v2:23}
console.log("value 1: "+this.v1+" value 2: "+this.v2)
```



```
value 1: 12   value 2: 23
```

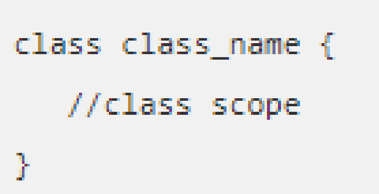
Classes

● ● ●

Infosys
be more

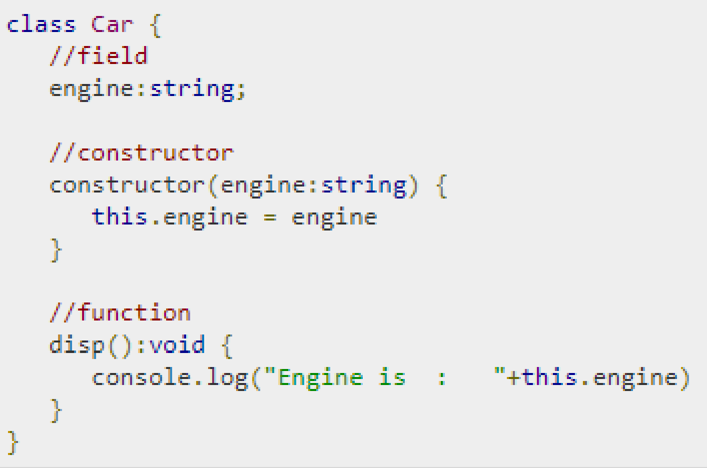
Classes

- Class is a blueprint to create objects.
- Class encapsulates data for the object.
- We can use classes to create reusable components like sign-in, sign-up, Customer, Student and so on.
- A class may include variables, constructors and methods.
- Creating a class:
- **Syntax**:



```
class class_name {
   //class scope
}
```

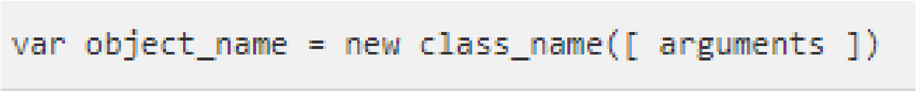
Example:



```
class Car {
   //field
   engine:string;

   //constructor
   constructor(engine:string) {
      this.engine = engine
   }

   //function
   disp():void {
      console.log("Engine is  :   "+this.engine)
   }
}
```

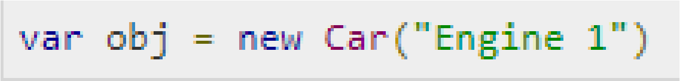
Classes

- Creating Instance objects:
- **Syntax**:



```
var object_name = new class_name([ arguments ])
```

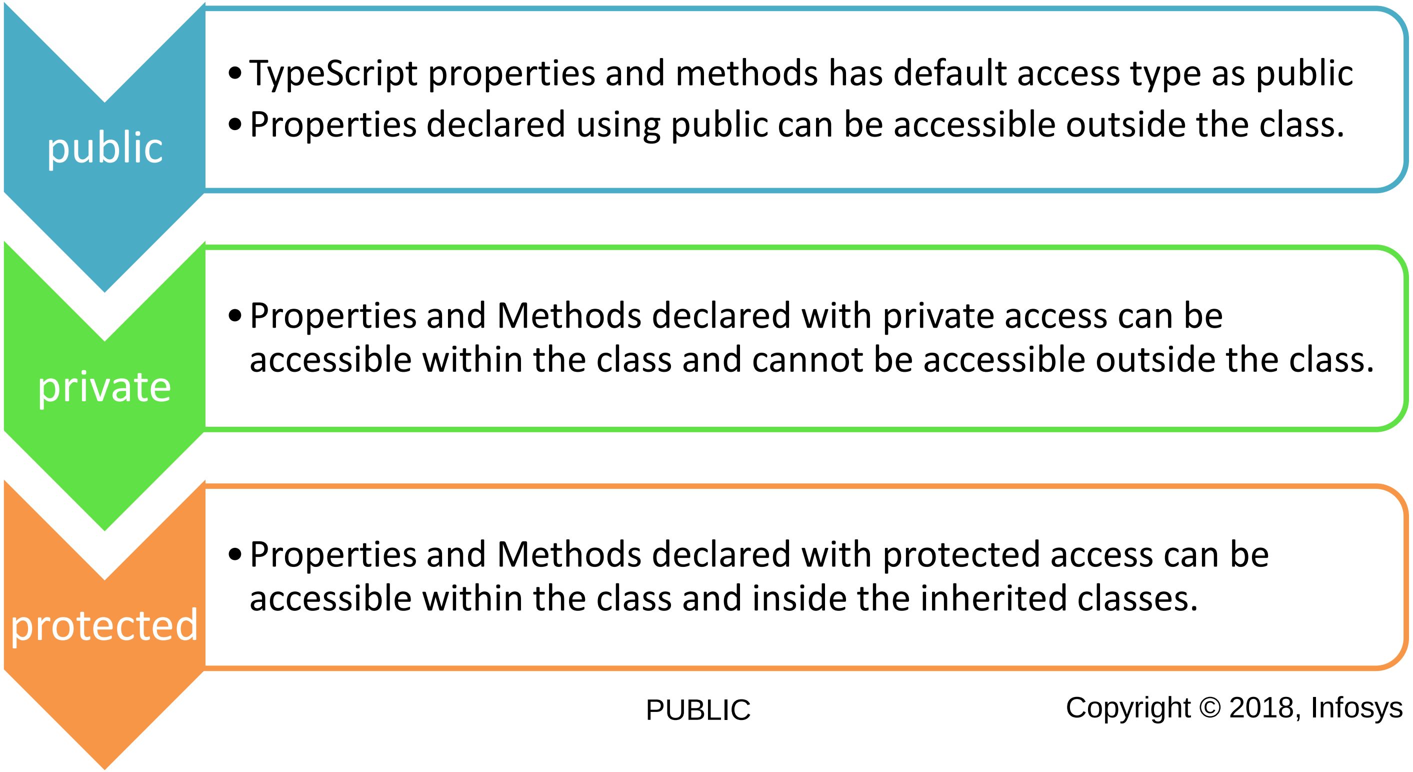
- **Example**: Instantiating a class



```
var obj = new Car("Engine 1")
```

Access Modifiers

- Access modifiers are used to provide certain restriction of accessing the properties and methods outside the class.



public

- TypeScript properties and methods has default access type as public
- Properties declared using public can be accessible outside the class.

private

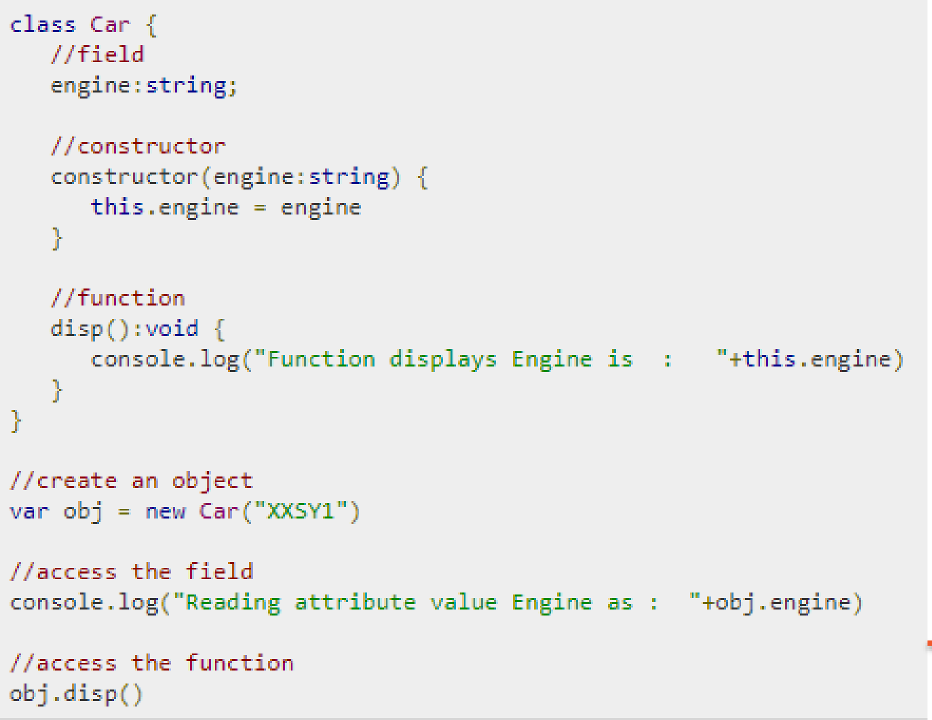
- Properties and Methods declared with private access can be accessible within the class and cannot be accessible outside the class.

protected

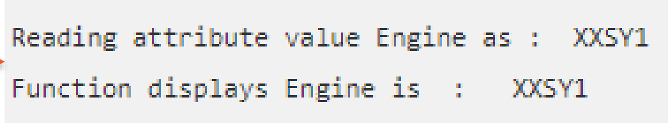
- Properties and Methods declared with protected access can be accessible within the class and inside the inherited classes.

Classes

Instantiating a class and Accessing variables:



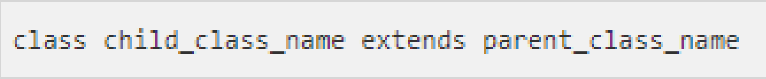
```
class Car {
   //field
   engine:string;

   //constructor
   constructor(engine:string) {
      this.engine = engine
   }

   //function
   disp():void {
      console.log("Function displays Engine is  :   "+this.engine)
   }
}

//create an object
var obj = new Car("XXSY1")

//access the field
console.log("Reading attribute value Engine as :  "+obj.engine)

//access the function
obj.disp()
```



```
Reading attribute value Engine as :  XXSY1
Function displays Engine is  :   XXSY1
```

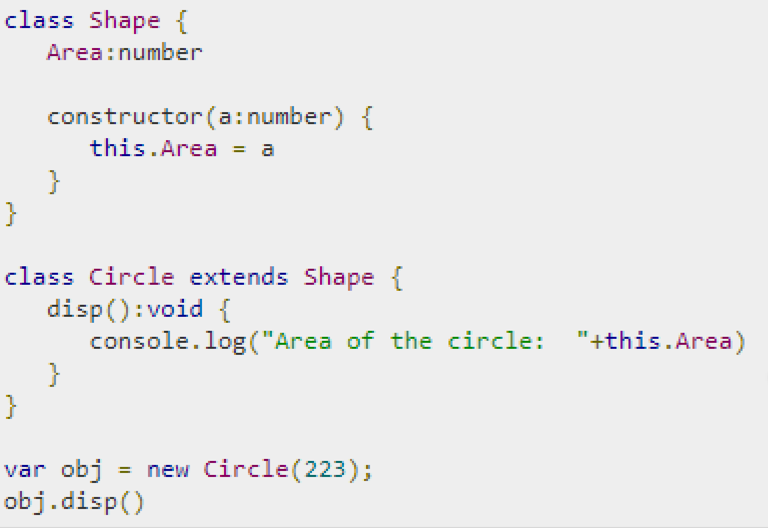
Extending Classes with Inheritance

- A class inherits from other class using ‘extends’ keyword.
- Child class inherits properties and methods except private members and constructors from a parent class.
- **Syntax**:

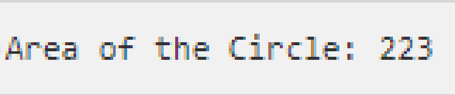


```
class child_class_name extends parent_class_name
```

- **Example**:



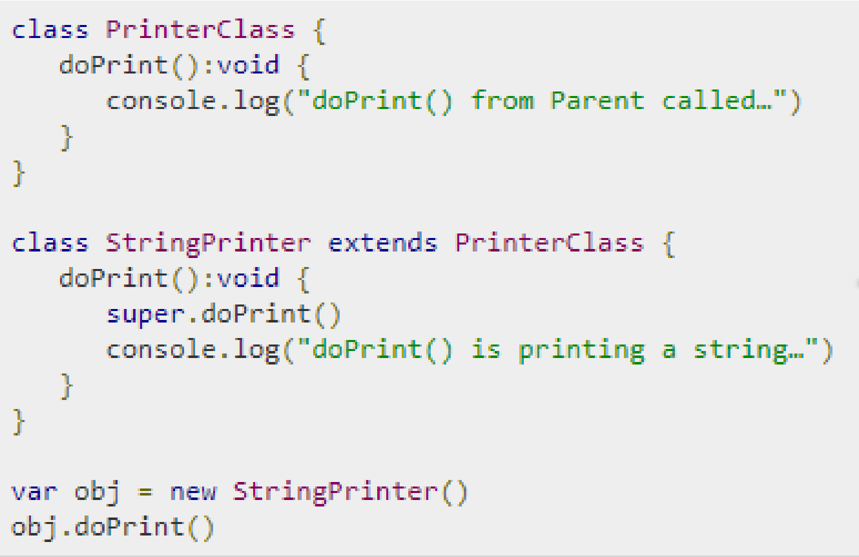
```
class Shape {
   Area:number

   constructor(a:number) {
      this.Area = a
   }
}

class Circle extends Shape {
   disp():void {
      console.log("Area of the circle:  "+this.Area)
   }
}

var obj = new Circle(223);
obj.disp()
```



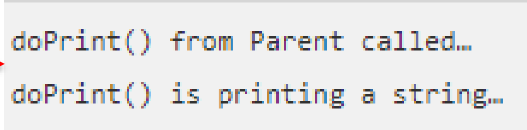
```
Area of the Circle: 223
```

Class Inheritance and Method Overriding

- Method overriding is a mechanism of a child class redefining method of a parent class.
- Super keyword is used to invoke parent class method
- **Example**



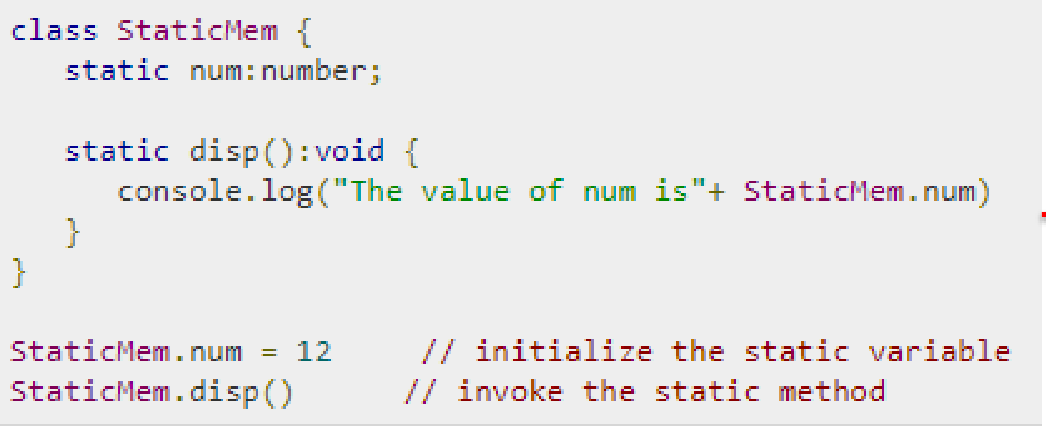
```
class PrinterClass {
   doPrint():void {
      console.log("doPrint() from Parent called…")
   }
}

class StringPrinter extends PrinterClass {
   doPrint():void {
      super.doPrint()
      console.log("doPrint() is printing a string…")
   }
}

var obj = new StringPrinter()
obj.doPrint()
```



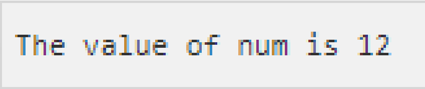
```
doPrint() from Parent called…

doPrint() is printing a string…
```

The Static variable and methods

- Static keyword can be used for variables and methods.
- Static variables retain their value till the end of execution of the program.
- They are referenced using class name.
- **Example**:



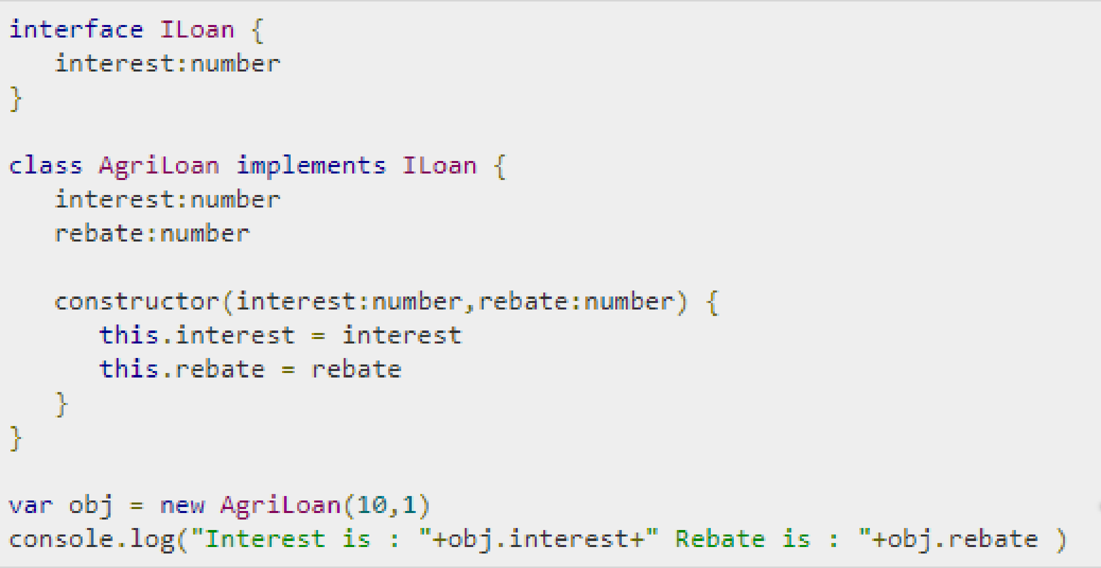
```
class StaticMem {
   static num:number;

   static disp():void {
      console.log("The value of num is"+ StaticMem.num)
   }
}

StaticMem.num = 12     // initialize the static variable
StaticMem.disp()      // invoke the static method
```



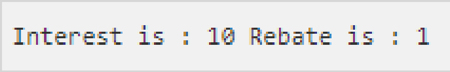
```
The value of num is 12
```

Classes and Interfaces

- Classes can also implement interfaces
- **Example**



```
interface ILoan {
   interest:number
}

class AgriLoan implements ILoan {
   interest:number
   rebate:number

   constructor(interest:number,rebate:number) {
      this.interest = interest
      this.rebate = rebate
   }
}

var obj = new AgriLoan(10,1)
console.log("Interest is : "+obj.interest+" Rebate is : "+obj.rebate )
```



```
Interest is : 10 Rebate is : 1
```

Modules

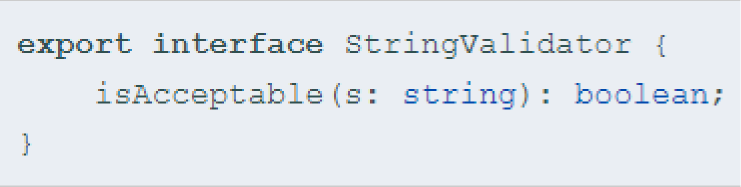
Modules

- Modules are used to logically group classes, interfaces, functions into one unit and can be exported to another unit.
- Modules execute in their own scope i.e., variables, functions, classes declared inside a module are not visible outside the class unless they are exported.
- Modules are declarative and their relationships are specified using import or export at file level.
- Modules import one another using a module loader. At runtime module loader is responsible for locating and executing all the dependencies of a module before executing it.

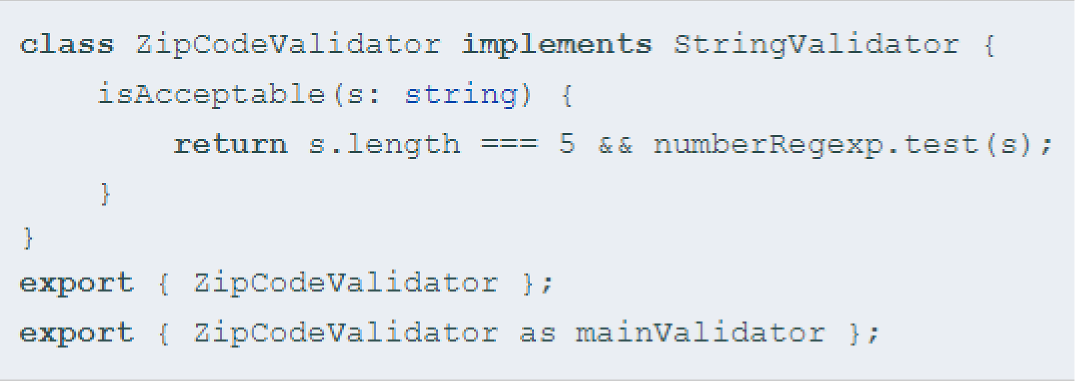
Modules

- Export a Module

  - Any declaration can be exported using ‘export’ keyword.



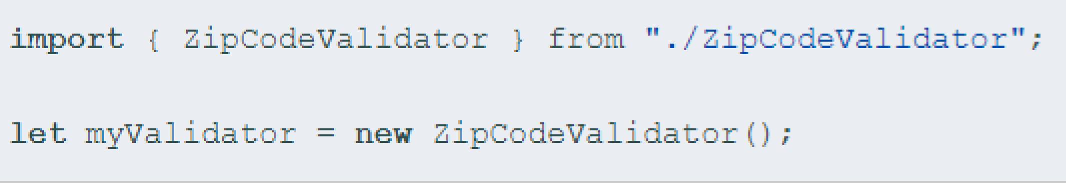
```
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
```



```
class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
export { ZipCodeValidator };
export { ZipCodeValidator as mainValidator };
```

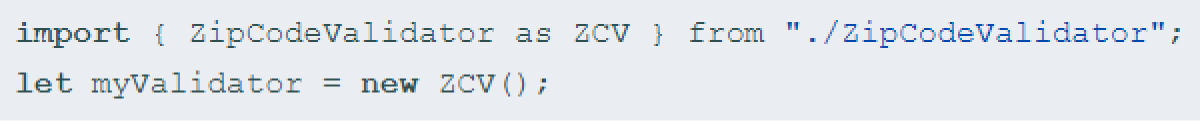
Modules

- Import a module
  - Importing an exported module is done using ‘import’ keyword.



```
import { ZipCodeValidator } from "./ZipCodeValidator";

let myValidator = new ZipCodeValidator();
```

  - Imports can be renamed

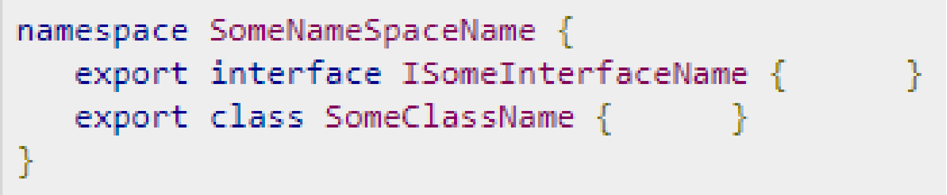


```
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
let myValidator = new ZCV();
```

Namespaces

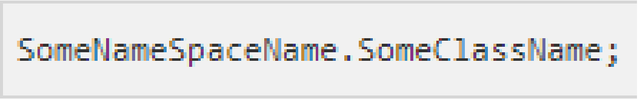
Namespaces

- Namespaces are a way to organize code.
- Internal modules are referred to as “namespaces”.
- “namespace” keyword should be used instead of “module” to declare a internal module
- Defining a namespace



```
namespace SomeNameSpaceName {
   export interface ISomeInterfaceName {      }
   export class SomeClassName {      }
}
```

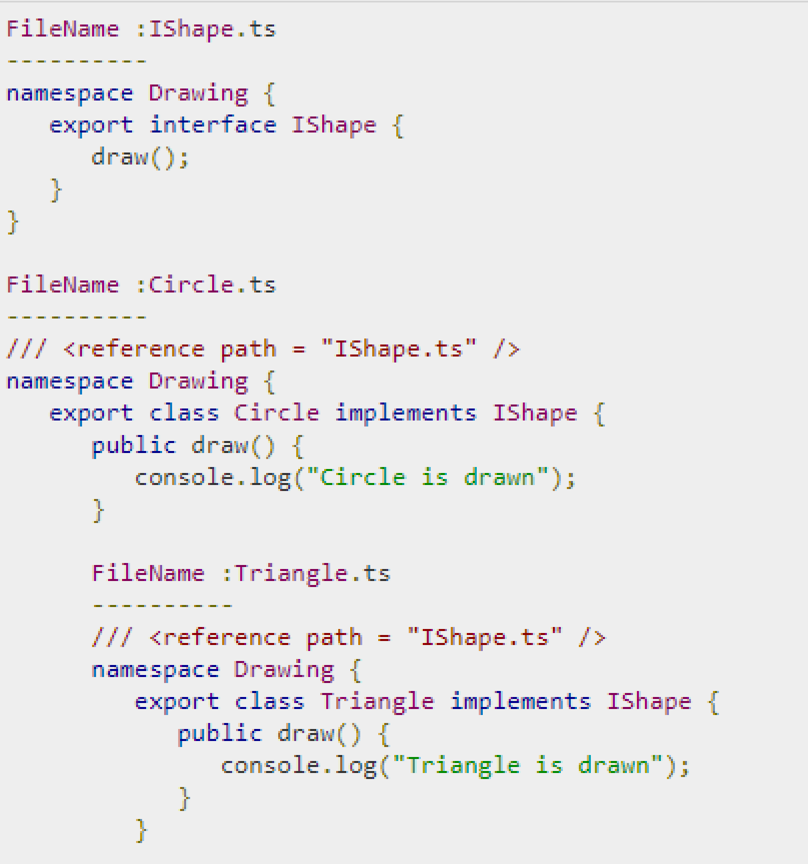
- Accessing a class or namespace in another namespace



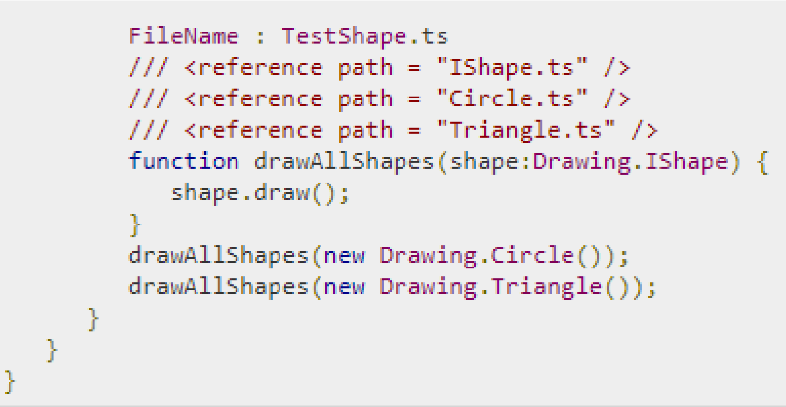
```
SomeNameSpaceName.SomeClassName;
```

Namespaces

- **Example**



```
FileName :IShape.ts
----------
namespace Drawing {
   export interface IShape {
      draw();
   }
}

FileName :Circle.ts
----------
/// <reference path = "IShape.ts" />
namespace Drawing {
   export class Circle implements IShape {
      public draw() {
         console.log("Circle is drawn");
      }

      FileName :Triangle.ts
      ----------
      /// <reference path = "IShape.ts" />
      namespace Drawing {
         export class Triangle implements IShape {
            public draw() {
               console.log("Triangle is drawn");
            }
         }
```



```
         FileName : TestShape.ts
         /// <reference path = "IShape.ts" />
         /// <reference path = "Circle.ts" />
         /// <reference path = "Triangle.ts" />
         function drawAllShapes(shape:Drawing.IShape) {
            shape.draw();
         }
         drawAllShapes(new Drawing.Circle());
         drawAllShapes(new Drawing.Triangle());
      }
   }
}
```

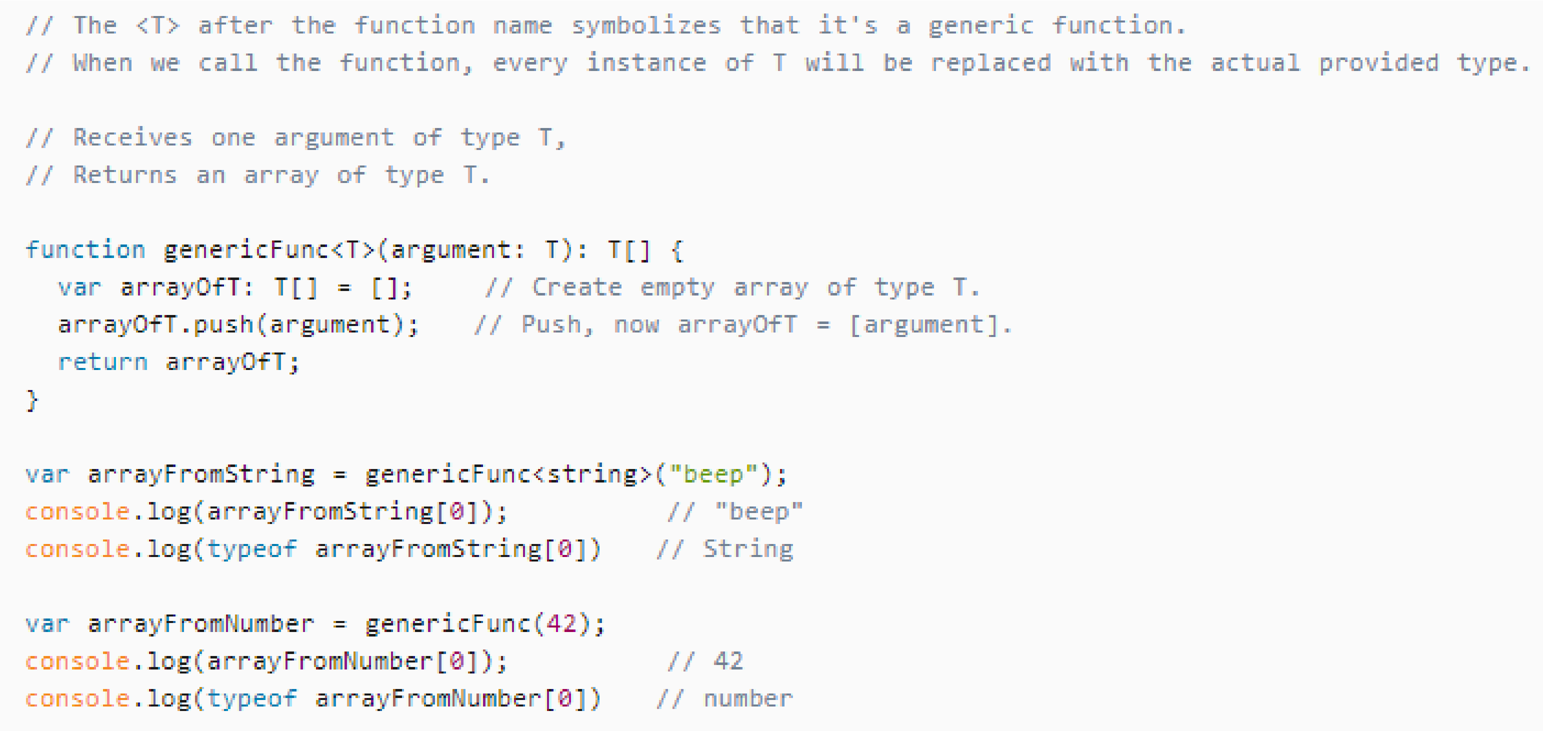
Generics

Generics

- Generics are templates that allow the same function to accept arguments of various different types.
- Creating reusable components using generics is a good practice compared to using the any data type, as generics preserve the types of the variables that go in and out of them.

Generics

- **Example**



```
// The <T> after the function name symbolizes that it's a generic function.
// When we call the function, every instance of T will be replaced with the actual provided type.

// Receives one argument of type T,
// Returns an array of type T.

function genericFunc<T>(argument: T): T[] {
  var arrayOfT: T[] = [];     // Create empty array of type T.
  arrayOfT.push(argument);    // Push, now arrayOfT = [argument].
  return arrayOfT;
}

var arrayFromString = genericFunc<string>("beep");
console.log(arrayFromString[0]);           // "beep"
console.log(typeof arrayFromString[0])    // String

var arrayFromNumber = genericFunc(42);
console.log(arrayFromNumber[0]);           // 42
console.log(typeof arrayFromNumber[0])    // number
```

Decorators

Decorators

- A Decorator is a special kind of declaration that can be attached to a class declaration, method, accessor, property, or parameter. They are used for declarative programming.
- Decorators use the form @expression, where expression must evaluate to a function that will be called at runtime with information about the decorated declaration.
- @component, @inject, @service, @pipe are some of the built in decorators used in Angular to apply metadata on classes to implement different concepts of Angular.
- To use decorator we need to set the experimentalDecorators compiler option either through the command line or in the tsconfig.json file.

Using command line: tsc - -target ES5 –experimentalDecorators

or

Using tsconfig.json: {

```
            “compileroptions”:{
                        “target” : “ES5”,
                        “experimentalDecorators” : true
            }
}
```

Class Decorator

- A class decorator is declared just before a class declaration.
- The class decorator is applied to the constructor of the class and can be used to observe, modify, or replace a class definition.
- The expression for the class decorator will be called as a function at runtime, with the constructor of the decorated class as its only argument.
- If the class decorator returns a value, it will replace the class declaration with the provided constructor function.
- **Syntax**: //Defining decorator function with the constructor of the decorated function as the parameter

```
function decoratorname (constructor : Function){
          . . .
          }
@decoratorname                              //Applying decorator using //decoratorname
class classname { }
```

Class Decorator

Example:

```
//Overriding original constructor with new one and returning new the constructor using logClass constructor
function logClass (constructor: Function) {
        var newconstrcutor : any = function (. . . args){
                this.studentID = 100;
                this.studentName = “Ram”;
                        }
        return newconstructor;        }
@logClass                                        //Applying decorator using @logClass
class Student  {
        public studentID: number;
        public studentName: string;
        constructor(studentID: number, studentName: string){
                this.studentID = studentID;
                this.studentName = studentName;    }
```

Project Configuration

Project Configuration

- Project Configuration in TypeScript is used to set the compiler options and also helps us in specifying the files to be included or excluded while performing the compilation.

tsc	IDE
Build Tool	tsconfig.com

Specifying compiler options

- Compiler option is used to specify configurations like target ES version to be used to compile, module loader to be used and so on.
- There are many compiler options available which you can refer from the TypeScript documentation:

  http://www.typescriptlang.org/docs/handbook/compiler-options.html

Common compiler options

--module
Or
--m

--sourcemap

--target
or --t

--watch
Or –w

--outDir

--outFile

Specifying Compiler Options

Option	Description	Example
- -target	Specify ECMA Script version: ‘es3’(default), ‘es5’, or ‘es6’	tsc - -target ES2015 filename.ts
- -module	Specify module code generation: ‘none’, ‘commonjs’, ‘amd’, ‘system’, ‘umd’, ‘es6’, or ‘es2015’	tsc - -module commonjs filename.ts
- -outdir	Redirect output structure to the directory	tsc - -outDir foldername filename.ts
- -outFile	Concatenate and emit output to single file. Order of concatenation is determines the list of files passed to compiler on command line along with triple-slash references and imports	tsc - -outFile outfilename.js filename1.ts filename2.ts
- - sourcemap	Generates corresponding .map file which is used to perform debugging	tsc - -sourceMap filename.ts
- -watch	Runs compiler in watch mode. Watches input files and trigger recompilation on changes.	tsc - -watch filename.ts

Role and Structure of tsconfig.json

- It is used to provide compiler options to a Typescript project.
- It helps in specifying the files to be included or excluded from the project.
- Once we add tsconfig.json file we can use tsc command to compile the files using the tsconfig.json file

```
Ex:     {
                //Provides compiler options to be configured while compiling .ts file
                "compileroptions": {
                                        "target": "es5",
                                        "outDir": "js";
                                        "module" : "amd",
                                        "outFile" : "moduletest.js"
                                   },
                //Provide file names to be compiled with configured compiler options
                "files" : [
                          "filename1.ts", "filename2.ts"
                     ]
 }
```

Summary

- Getting started with TypeScript
- TypeScript Basics
- Functions
  - Functions in TypeScript vs JavaScript
  - Arrow Functions
  - Optional and Default parameters
  - Rest Parameters
- Interfaces and Classes
- Modules
- Namespaces
- Generics
- Decorators
- Project Configuration

References

- https://www.keycdn.com/blog/typescript-tutorial/
- https://www.tutorialspoint.com/typescript/typescript_overview.htm
- https://tutorialzine.com/2016/07/learn-typescript-in-30-minutes
- https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html
- http://blog.teamtreehouse.com/getting-started-typescript

Thank You

● ● ●

Infosys *be more*