TypeScript

TypeScript is a superset of JavaScript, it provides OOPS Concepts like functions , classes and interfaces.

TypeScript has a feature strongly-typed or supports static typing. That means Static typing allows

for checking type correctness at compile time. This is not available in JavaScript.

typescript can not understand by browsers so there is typescript compiler whcih can convert into Javascript

then jvascript can render into browser and provide dynamic functionality.

Typescript has a feature OF ECMAScript standard

TypeScript is a transpiler(It converts one language into another language.)

complie: tsc sample.ts

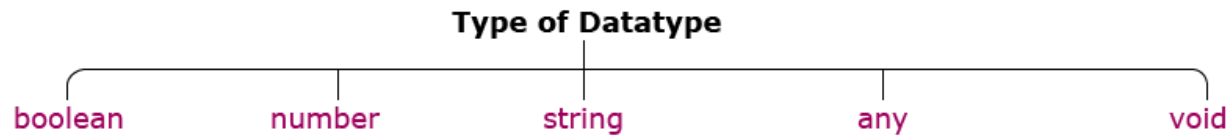
Declaring variables:

Declaring variables using `var` and `let` keywords:

```
1 var productName="Mobile";
```

```
1 let productName = "Mobile";
```

Data types:



let productId:number=1045;

let productDescription:string="16GB, Gold";

let productAvailable: boolean = true;

let manufacturers:any = 20;

const discountPercentage:number=15;

console.log("The type of productId is " + typeof productId);

```
1.  //Declaring variables with basic types using let
2.
3. let productId:number=1045; //Declaring a numeric variable
4.
5. let productDescription:string="16GB, Gold"; //Declaring a string variable
6.
7. let productName:string="Samsung Galaxy J7";
8.
9. let productAvailable: boolean = true;   //Declaring a boolean variable
10.
11. console.log("The type of productId is " + typeof productId);
12.
13. console.log("The type of productAvailable is "+typeof productAvailable);
14.
15. console.log("The type of productName is "+typeof productName);
16.
17. //Declaring variables using const
18.
19. const discountPercentage:number=15;
20.
```

Enum:

it is named integer constants, each and every item in enum should have value.by default it start with 0

enum MobilePrice{Black=25000,Gold,White=30000};

MobilePrice.Gold //25000

MobilePrice["Black"] // 25001

```
Syntax: enum EnumName{property1, property2, property3};
```

TypeScript

Example:

```
enum MobilePrice{Black,Gold,White};
```

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

Array:

array can be used for store same type of data type values.

let manufacturers:any[] = [{ "id": "Samsung", "checked": false }];

let color: string[] = ["red","green","blue"];

let color: Array<string>= ["red","green","blue"];

console.log(color.pop()); ---blue

Tupple:

its a kind of array , but it used to store different type of data type values where array can store same type of data type values ;

let productAvailable: [string, boolean, number]=["Samsung Galaxy J7",true, 20];

functions:

function getMobileByManufacturer(manufacturer: string): string[]{

}

Arrow function:

functions without name called arrow funtions

var products=(productid:number)=> { return "productid" + productid};

here products is the functiona name so products(1) //productid1

TypeScript

```
let manufacturers = [{ "id": "Samsung","price":15000 }];

let test = manufacturers.filter((manufacturer)=>manufacturer.price>=20000);
```

optional and default parameters:

```
function getMobileByManufacturer(manufacturer: string="Samsung",id?:number): string[]{

}

getMobileByManufacturer("Apple");
```

Rest parameters:

Rest Parameter is used to pass multiple values to a single parameter of a function.

It accepts zero or more values for a single parameter.

Rest Parameter should be declared as an array.

rest parameter preceded with triple dots.

should be the last parameter in the function parameter list.

```
function addtoCart(prodid:string, ...productName: string[]): string[]{

}

addtoCart(1,"a","b");


function  add(num1:number,num2?:number) {
   if(num2){
      return num1+num2;
   }
   return num1;
}

console.log(add(3,5)+add(5)+add(3));
```

TypeScript

Interfaces:

It is a collection of properties and method declarations.

Interface keyword is used to declare an interface.

Properties or methods in an interface should not have any access modifiers.

Properties cannot be initialized in a TypeScript interface

methods should not have body only definatins.

```
interface Product{
        productId:number;
        productName:string;
}
function getProductDetails(productobj:Product):string{
   return "The product name is "+productobj.productName;
}
let prodObject={productId:1001,productName:'Mobile'};
let productDetails:string=getProductDetails(prodObject);
console.log(productDetails);
```

Duck typing:

passing extra paramters for interface object type , it will be ignored for type checking for extra paraeters.

```
let prodObject={productId:1001,productName:'Mobile' pcategory:"mobilesdata"};  //ignoe type checking for pcategory:"mobilesdata"
let productDetails:string=getProductDetails(prodObject);
```

TypeScript

```
interface Product{
        productId?:number;
        productName:string;
        Getdetails(productid:number):string;
        Getdetails2:(productid:number)=>string;
}
```

function type of interface:

```
Function CreateCustomerID(name: string, id: number): string{
return "The customer id is "+name + id;
}
interface StringGenerator{
(chars: string, nums: number): string;
}
let IdGenerator: StringGenerator;
IdGenerator= CreateCustomerID;
let id: string = IdGenerator("Infy", 101);
console.log(id);
```

extending interface:

```
interface Category{
categoryName:string;
}
interface Product{
        productName:string;
```

```
        productId:number;
}
interface productList extends Category,Product{
        list:Array<string>;
}
let productDetails:productList={
        categoryName:'Gadget',
        productName:'Mobile',
        productId:1234,
        list:['Samsung','Motorola','LG']
}
let listProduct = productDetails.list;
let pname: string = productDetails.productName;
```

implimenting interface with class:

```
interface Product{
        getProductDetails():string[];
        displayProductName:(prouctId:number)=>string;
}
class Gadget implements Product{
   getProductDetails(): string[]{

   }
   getGadget(): string[] {
```

```
        return ["Mobile","Tablet","iPad","iPod"];
    }
}
let g:Product=new Gadget();
let g:Gadget=new Gadget();
```

class and constrctor:

Use class keyword to create a class.

```
class Product{
            productId:number;
            constructor(productId:number){
                        this.productId=productId;
            }
            getProductId():string{
         return "product id is"+this.productId;
            }
}
var product:Product=new Product(1234);
console.log(product.getProductId());
```

access modifiers and extending class(inheritence):

private , public , protected

```
class Product{
```

```
        private productId:number;
        public productName:string;
        static productPrice: number=15000;
        protected productCategory:string;
        constructor(productId:number,productName,productCategory){
                this.productId=productId;
                this.productName=productName;
                this.productCategory=productCategory;
        }
        getProductId(){
        console.log("The productId is "+this.productId);
        }

}
class Gadget extends Product{
        getProduct():void{
                console.log("ProductCategory: "+ this.productCategory);
        }
}
var g:Gadget=new Gadget(1234,"Mobile","SmartPhone");
g.getProduct();
g.getProductId();
```

TypeScript

```
class Product{
        protected productId:number;
        constructor(productId:number){
                this.productId=productId;
        }
        getProduct():void{
                console.log("ProductID"+ this.productId);
}}
class Gadget extends Product{
        constructor(public productName:string,productId:number){
                super(productId);
        }
        getProduct():void{
                super.getProduct();
                console.log("ProductID"+ this.productId+"ProductName"+this.productName);
        }
}
var g=new Gadget("Tablet",1234);
g.getProduct();
```

accessers in class:

```
let passcode = "secret passcode";
class Product {
   private _productName: string;
   get productName(): string {
```

```
        return this._productName;
    }
    set productName(newName: string) {
        if (passcode && passcode == "secret passcode") {
            this._productName= newName;
        }
        else {
            console.log("Error: Unauthorized update of employee!");
        }
    }
}
let product:Product = new Product();
product.productName = "Fridge";
if (product.productName) {
    console.log(product.productName);
}
```

abstract class , extending abstract class:

```
abstract class Product{
            getFeatures():void{

            }
            abstract getProductName():string;
            }
class Gadget extends Product{
```

```
getProductName():string{
        return "ProductName is Mobile";
        }
}
class Clothing extends Product{
        getProductName():string{
        return "ProductName is Shirt";
        }
}
var g=new Gadget();
console.log(g.getProductName());
var c=new Clothing();
console.log(c.getProductName());
```

namespace:

Namespace is used to group functions, classes or interfaces under a common name(Organizes code).

The content of namespaces are hidden by default unless exported.

We can have nested namespaces if required.

No special loader required

ES2015 does not have Namespace concept. It is mainly used to prevent global namespace pollution.

Suited for small - scale applications

```
namespace Utility {
    export namespace Payment {
    export function CalculateAmount(price: number,quantity:number): number {
```

```
            return price*quantity;
        }
    }
        export function MaxDiscountAllowed(noOfProduct: number): number {
        if (noOfProduct >5) {
            return 40
        } else {
            return 10;
        }
    }
        function privateFunc(): void {
        console.log('This is private...');
    }
}
```

```
/// <reference path="./namespace_demo.ts" />
import util = Utility.Payment;
let paymentAmount = util.CalculateAmount(1255,6);
console.log(`Amount to be paid: ${paymentAmount}`);
let discount=Utility.MaxDiscountAllowed(6);
console.log(`Maximum discount allowed is: ${discount}`);
```

TypeScript

module, default export class of module:

Organizes code

Have native support with Node.js module loader. All modern browsers are supported with module loader.

Supports ES2015 module syntax

Suited for large - scale applications

Modules helps us in grouping a set of functionalities under a common name.

Precede export keyword to the function, class, interface, etc.. which you need to export from a module

Create a file module_demo.ts

```
export function MaxDiscountAllowed(noOfProduct: number): number {
      if (noOfProduct > 5) {
         return 30;
      } else {
         return 10;
      }
   }
class Utility {
CalculateAmount(price: number, quantity: number): number {
           return price * quantity;
}

}
interface Category {
           getCategory(productId: number): string;
}
```

TypeScript

```
 export const productName:string="Mobile";

export {Utility,Category};


export default class{

productName: string="Tablet";

getProductDetails(productId:number):string{

        return "ProductId is "+productId+"ProductName is "+this.productName;

}

}
```

```
import Product,{ Utility as mainUtility,Category,productName,MaxDiscountAllowed } from
"./module_demo";

let util =new mainUtility();

let price=util.CalculateAmount(1350,4);

let discount=MaxDiscountAllowed(2);

console.log(`Maximum discount allowed is: ${discount}`);

console.log(`Amount to be paid: ${price}`);

console.log(`ProductName is: ${productName}`);
```

generics:

Generics is a concept using which we can make the same code work for multiple types.

we can create generic cls , mthod , interface

generics fun:

```
function orderDetails<T>(arg: Array<T>): Array<T> {
```

```
    console.log(arg.length);
    return arg;
}
let orderid:Array<number>=[101,102,103,104];
let ordername:Array<string>=['footwear','dress','cds','toys'];
let idList=orderDetails(orderid);
console.log(idList);
let nameList=orderDetails(ordername);
console.log(nameList);

function printData<T>(data:T):T{
return data;
}
let data:string=printData<string>('Hello Generics');
console.log("String data"+data);
class Product{
        productName:string;
}
let productData:Product={productName:'Tablet'};
let data2:Product=printData<Product>(productData);
console.log("Object data"+data2);
```

generic interfaces :

```
interface Inventory<T> {
addItem: (newItem: T) => void;
```

```
getProductList: () => Array<T>;
}
class Gadget implements Inventory<string>{
addItem(newItem:string):void{
console.log("Item added");
}
productList:Array<string>=["Mobile","Tablet","Ipod"];
getProductList():Array<string>{
return this.productList;
}
}
let productInventory:Inventory<string>=new Gadget();
let allProducts:Array<string>=productInventory.getProductList();
console.log("The available products are:"+allProducts);
class Shipping implements Inventory<number>{
addItem(newItem:number):void{
console.log("Item added");
}
shippingID:Array<number>=[123,234,543];
getProductList():Array<number>{
return this.shippingID;
}
}
let shippingInventory:Inventory<number>=new Shipping();
let shippingIDs:Array<number>=shippingInventory.getProductList();
```

```
console.log("The shipping IDs:"+shippingIDs);
```

generic class:

```
class Gadget <T>{
        productList:Array<T>;
addItems(newItemList:Array<T>):void{
        this.productList=newItemList;
        console.log("Item added");
}
getProductList():Array<T>{
return this.productList;
}
}
let product=new Gadget<string>();
let productList:Array<string>=["Mobile","Tablet","Ipod"];
product.addItems(productList);
let allProducts:Array<string>=product.getProductList();
console.log("The available products"+allProducts);
let product2=new Gadget<number>();
let shippingList:Array<number>=[123,234,543];
product2.addItems(shippingList);
let allItems:Array<number>=product2.getProductList();
console.log("The available shipping ids"+ allItems);
```

generic constraints:

TypeScript

```
interface AddLength{
        length:number;
}
function orderLength<T extends AddLength>(arg:T): T{
   let lengthValue=arg.length;
        console.log("Length is "+lengthValue);
   return arg;
}
class Product implements AddLength{
        length:number=10;
}
let product:Product=new Product();
let product1=orderLength(product);
console.log("Product Length"+product1.length);
let ordername:Array<string>=['footwear','dress','cds','toys'];
let order1=orderLength(ordername);
console.log("Order length is"+order1.length);
```